

Development tool for service-oriented applications in smart homes

Jianqi Yu, Philippe Lalanda and Stéphanie Chollet
Laboratoire Informatique de Grenoble
F-38041, Grenoble cedex 9, France
Jian-qi.Yu@imag.fr
{firstname.lastname}@imag.fr

Abstract

In this paper, we present a model-driven, domain-centric tool allowing the specification of service-oriented applications through abstract services composition. Executable applications, made of concrete services, are automatically generated and deployed. This tool can be customized with domain-specific knowledge. Our contributions in this paper are to slightly extend the tool to be used in the demanding home computing domain. An approach for modelling and implementing dynamic service composition is provided for satisfying the dynamic home environment. This work has been carried out in collaboration with Schneider Electric.

Key words: Development environment, smart home.

1. Introduction

Software systems pervade every aspects of our life and, as a consequence, place very demanding requirements on new development projects. In domains like home computing, unprecedented pressure is put on cost and time-to-market. Pervasive applications are also characterized by complex requirements in terms of dynamism, context awareness and autonomy. New development paradigms have been proposed to face these new demanding conditions. Product-line approaches have been defined to decrease time-to-market and cost while increasing quality of repetitive products. Service-oriented computing has been proposed to better deal with frequently changing environments. Dedicated tools are naturally expected by the industry to support these paradigms.

Service-Oriented Computing (SOC) is a paradigm that uses services as fundamental elements for developing applications [8]. Services are self described computational elements that can be dynamically used by third parties.

Services are expected to support the rapid, inexpensive development of distributed applications. Since each element can be developed and evolved separately, the resulting application is more flexible than monolithic applications. Services can be simple services or composite [8] that is made as an assembly of existing services. Composing services to form an application is a complex, error-prone task for several reasons. First, there are many technologies for describing, publishing and composing services, such as OSGi, Web Service and UPnP. Different protocols and mechanisms can be used to implement a service-oriented architecture. In addition, the correctness of a composition is hard to prove. Syntactic correctness requires services to be connected through compatible interfaces. Semantic correctness, harder to express and to obtain, refers to services that behave as expected by other services. Currently, technologies capable of verifying the correctness of service compositions are only emerging. An interesting approach to compose services effectively is to rely on domain information and on a product-line approach. A software product line is a set of software-intensive systems sharing a common, managed set of features satisfying the specific needs of a particular market segment and that are developed from a common set of core assets in a prescribed way [11]. Typically, an instance product is created by taking reusable components from the base of common assets and then tailoring them in order to satisfy a specific requirement. Each product architectural instance has to conform to the product line architecture defined in a specific domain.

We believe that SOC and software product line have some common ground. The business process or product line architecture within a specific context is a key factor for verifying application correctness in a particular context. They utilize reusable and existing elements for improving application development efficiency. Especially, the software product line instantiates products through

managing variability within existing assets. Through the architectural view, both approaches provide the possibility of considering the product line architecture as a mechanism for verifying the service composition correctness at an abstract level.

This paper presents a tool allowing the specification of service-oriented applications through abstract services composition. Executable applications, made of concrete services, are automatically generated and deployed. This tool can be customized with domain-specific knowledge. In this paper, we focus on home computing applications. The work presented hereafter has been carried out in collaboration with Schneider Electric, within the ANSO¹ European project. The remainder of this paper is organized as follows. In Section 2 we discuss the major challenges in the smart home computing and modelling new domain difficulties in using the previous work. Section 3 presents the main contributions of this paper. Section 4 details the implementation of our propositions and presents an approach based on model transformations. Section 5 provides an illustration in the home domain. Conclusions and outline of further work are presented in Section 6.

2. Background

2.1 Home computing

Digital technologies have been rapidly evolving and are today pervading many domains, including home computing. Many buildings and houses are already covered in wireless or wired network technologies. A number of heterogeneous networks and devices like computers, smart-phones, set-top-boxes, TVs, DVDs, and intelligent digital appliances constitute the so-called home computing environment. The main stake today is to develop, deploy and maintain services based on the devices and networks available in such buildings.

This vision is however hard to realize. Many technical challenges have to be met:

- Heterogeneity. Today, numerous manufacturers are working on new, innovative electronic devices and home applications on top of different platforms. More than 50 candidate protocols, working groups and standard specifications for home networking already exist [4]. The seamless integration of heterogeneous communication is a major concern.

- Scalability. Two aspects have to be considered regarding this aspect. One is related to the number of devices connected in the house. The other is related to the numbers of houses that can be connected to a service management center.
- Security. The main purpose of a home application is to provide comfort and security for the residents. Home devices and associated services have to be secured. More precisely, the external access to the home platform should be secured for maintaining some private properties and transporting sensitive data as well as the internal communication among devices and applications.
- Dynamicity. Home networks are made of electronic devices that are allowed to leave or join the network at any time. This may result in losses of data or even in fatal errors of the execution platform. Therefore, verification of device or/and service availability including failure detection, intermittence of connection, or new installations is an important concern.
- Autonomy. The autonomy of the platform is a crucial issue. In particular, the platform should continue to function in spite of some different devices appearing or disappearing dynamically.

SOC provides many properties that deal with such complex requirements in this domain. Indeed, UPnP, OSGi or Web Services have been recently used in home related applications. We are actually investigating architectures including these three technologies. However, in this paper, we focus on building Internet gateways and connected UPnP devices. As illustrated by Figure 1, we are working on innovative architectures where devices are represented as service providers and requesters. Devices are structured into 3 categories:

- Electronic devices incorporated into buildings (controllable shutters or lights for instance) providing basic services through UPnP to sense and act on the environment,
- Gateways providing computing resources allowing to run high level services that aggregate basic services provided by the previous type of devices,
- Rendering devices (TVs, smart phones or PDAs for instance) allowing users to interact with the home services and, possibly, to administrate them.

In our architecture shown in figure 1, gateways run service-oriented applications connecting devices and also orchestrating the devices' actions. Using a service-oriented framework allows developers to implement highly flexible applications where devices and applications can change over time. Unfortunately deep technical knowledge is needed to design, implement,

¹ This work is partially supported by the French Ministry of Industry under the ITEA European program.

assemble, deploy and maintain such service-oriented applications. Thus, most programmers do not take full advantage of the SOC capabilities.

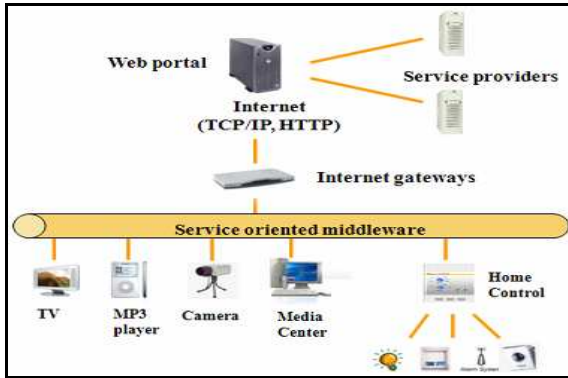


Figure 1. Building computing architecture

2.2 DoCoSOC

We have implemented a domain-oriented generative approach [17] where applications are generated from service compositions. We have also added domain specialization capabilities to the tool in order to facilitate the developers' work. This additional feature is not presented in this paper. More information can be found in [1].

We have defined the notions of abstract and concrete services. An *abstract service* is a service specification independent of any given implementation technology. It retains the major service-oriented features and ignores low-level technological aspects. In our model, an abstract service is defined in the following terms:

- Functional interfaces specifying the provided capabilities.
- Properties including service properties and configuration properties representing the data needed to configure a service before using it.

A *concrete service* is an implementation of an abstract service in a given service technology. A single abstract service may be implemented by multiple concrete services.

An *application* is formed by a set of abstract services that are connected through abstract connectors. Once a composition is specified, appropriate concrete services have to be selected in order to constitute executable application. Although connectors themselves can be implemented as services, some glue code is generally necessary to allow the interaction between potentially heterogeneous services. We readily acknowledge that executable applications cannot be automatically generated in the general case. However, we strongly believe that this becomes possible as soon as this approach is applied in a restrained area, i.e. a domain.

The presented tool has the following components:

- An abstract service repository,

- A composition editor,
- A concrete service repository.

Both repositories (abstract and concrete) are structured elements that can be browsed or searched. They are managed by the tool administrator who is in charge of the creation, modification, suppression of services, which may be abstract or concrete.

The abstract service repository is directly used by the application assembler through the composition editor. The editor allows the assembler to select abstract services and to link them via connectors.

The concrete service repository is invisible for the application assembler. It is used by the deployment manager whose responsibility is first to select concrete services implementing an abstract composition configured in the previous composition editor. In addition, the deployment manager has to generate the glue code necessary to run the concrete composition.

3. A meta-model approach

DoCoSOC has been developed according to a Model Driven Engineering approach. More precisely, we have built a meta-model defining in a technological agnostic way all the aspects of service-oriented computing at the different development phases. Then, we have used advanced tools from the MDE community (see the EMF Eclipse Plug-in at www.eclipse.org/emf/ and GMF Eclipse Plug-in at www.eclipse.org/gmf/) to generate the skeleton and the graphical part of our development tool. Relying on a meta-model, presented in detail in [16], and turned out to be very beneficial. First, it made important concepts explicit and forced us to abstract away the essence of the SOC approach. Second, manipulating such explicit knowledge favors evolutions, which are unavoidable in long-term collaborative projects. Smart use of EMF and GMF allows the generation of new tool versions, following changes in the meta-models, in short delays.

3.1 Concepts of the SOC meta-models

In order to tackle home applications, we had to evolve DoCoSOC. The SOC meta-model and the domain modeling meta-model have been slightly extended. This is due to the fact that home applications require to be configured dynamically especially at runtime due to home networking characteristics. Also, we have added new target platforms in order to conform to the technologies that are used in home networking.

As presented by figure 2, we have decided to add a *Service Description* concept. This concept contains the functional interfaces and the relevant properties of the service as well as non-functional aspects (log and security

for instance). An *Abstract Service* represents the functional interfaces and the tightly-coupled functional interfaces required. The main benefit of separating service properties from the *Abstract Service* concept is to help the domain model expert always concentrate on the business logic of a specific domain rather than consider an implementation technology at the same time. Properties, including service properties and configurable properties, are also attached to *Service Description* features. In this way, several service descriptions can be differentiated depending on the different properties whereas they describe the same abstract service. In addition, a *Service Description* can be a composition of other service descriptions to form a new composite service. *Service Description Repository* thus becomes a service provider by publishing the composite service description. This evolution enables the *Application* to constitute service compositions in a simpler way, using simple services and composite services collected by the service description repository.

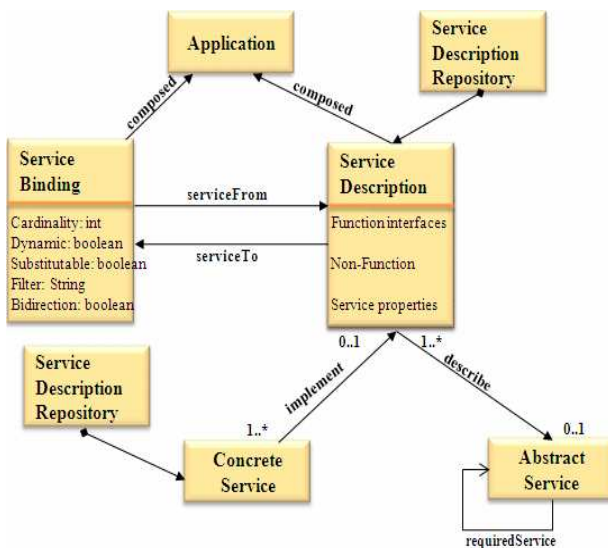


Figure 2. **Extended SOC meta-model**

As in the Service Component Architecture (see SCA at www.osoa.org), we have added a dependency relation called “Required Service” to the *Abstract Service*. The “Required Service” specifies the component interfaces that the abstract service needs in order to provide its advertised functions. This dependency offers the necessary information when verifying the syntax correctness of a service composition within a specific domain. For creating a new service, the “Required Services” also provides a reference towards the implementation library where the required services may be found.

An *application* specifies a reference architecture (in product-line terms) [11] of a specific domain. It determines the application structure, which comprises

Service Descriptions and *Service Bindings* (relationship among *Service Descriptions*). All configured service compositions, which are instances of *application*, have to conform to the reference architecture. Moreover, each service composition configuration determines a binding time(s) for composed elements. The binding time is the point in time where the decision for a composition of particular elements is made [10]. Examples of binding times include development time, packaging time, deployment time, and runtime. In a previous project, called PISE [5], applications had to be modeled statically before deployment time. It is not the case anymore in home computing environments.

3.2 Dynamic features

Dynamic service composition configuration and implementation is related to the following different binding times [10]:

- before deployment time. New methods or fields are added in a functional interface of the existing abstract service described in the specific domain model. This existing abstract service should become a new service type. This results in new relationships, such as inheritance and dependencies between existing services in the domain model.
- at deployment time. A number of services’ instances required by the deploying composition have to be available on the target execution platform. It has also to consider some properties of the platform like performance.
- at execution time – runtime. Slots of a service instance once appear or disappear on the platform at runtime.

Regarding binding times, a crucial concept-“Service Binding”- is defined for describing relationships between two services of a service composition. Otherwise, in order to model dynamic composition, we have defined the following dynamic attributes within this concept:

- Cardinality. This attribute specifies a number of service instances that can be linked to a given service instance at run time. Cardinality can be defined as an interval, from zero to a given value for instance. If this last value is unknown at the time of application modeling, it should be defined as a variable number. At execution time, the number of instances of connected services is often variable. Depending on the value of this attribute, an application may be stopped, uninstalled or kept running when the numbers of connected instances vary. However, it is not easy to model the three sorts of cardinalities via a single attribute in an EMF model. Therefore, a

dynamic attribute has been created to differentiate the three different cardinalities in the model.

- **Dynamic.** When set to true, this attribute permits a service instance to be linked to several instances, even if this Service Binding has affected a value to the “Cardinality” feature between the two services. Before deployment time, an application has to verify the availabilities of the services making up the application. This feature is capable of making decision for deploying an executable application. At runtime, with the feature cardinality, it can update the status of an execution application like stopping, executing or uninstalled when some service instances of the application have dynamically disappeared.
- **Substitutable.** This attribute allows a given service to connect a set of various services called “service variant(s)”, which provide similar functions. However, at binding time just a single service variant can be selected from the set of various services for communicating with the given service. This feature is used at three binding times. Before deployment time, it is used to verify the semantic correctness of a service composition configuration in terms of the defined domain reference architecture. At deployment time, it permits to validate a service composition deployment. The given service instance may function as long as only one service variant instance of the set of various services defined within the *Service Binding* is available on the targeted platform. At runtime, a service composition configuration consisting of the given service and one service variant can continuously function on the platform when the service variant instance is unavailable due to the “Substitutable” attribute specification. This feature supports to automatic discovery of an alternative available service variant instance from the set of various services.

Furthermore, Service Binding has some features such as *Filter* for describing certain connecting conditions and *Bidirectional* for specifying the direction between service descriptions.

3.3 Code generation

Administrators can customize the Service Description Repository for a specific domain in order to publish, create and discover available services. Dynamic service composition implementations absolutely require a dynamic *Concrete service repository*. This new Concrete service repository corresponding to the service description repository is able to update the status of the target platform at runtime due to the integration of the UPnP technology [7]. A service description can be described by several concrete services depending on different implementation technologies and some service

configuration properties. For developing a dynamic service composition, a service implementation selection mechanism is necessary, so as to produce the functional code and certain glue code of service interaction during the code generation.

The general implementation selection strategy consists of selecting an implementation that matches the required functional interfaces with the latest version of the implementation corresponding to the different technologies. The new implementation selection strategy has to take into account these dynamic features of an abstract application. The following section will express the complete phrase of a home application implementation from the domain model specification to the application deployment on a target platform with our tools.

4. Implementation

In this paper, the target platform of a home application deployment is the OSGi and iPOJO. An MDA approach is used to automate the application development in terms of different target platforms.

4.1 OSGi

OSGi (www.osgi.org) defines a framework to dynamically deploy services in a centralized environment. This technology allows for automatic discovery, installation and management of devices and nodes in networks via plug-and-play capabilities. It also provides for integration of existing wired backbones and new wireless solutions that offer quality of service managed throughput [19]. Furthermore, the OSGi specification supports some technical services like application security and remote management.

4.2 iPOJO

iPOJO (felix.apache.org/site/ipojo.html) is a service oriented component framework based on the OSGi technology. It offers a specific mechanism to extend the service component model that allows for adding technical services not supported by the OSGi platform, such as persistence, configuration, services dependency management and scheduling automatically, at runtime. This extension mechanism allows developers to concentrate on implementing the business functions of a service composition. It hides most of the complexity of integrating several technologies and automatically resolves the service dependences of the service composition.

4.3 Model Transformation

Our propositions are realized by applying a Model Driven Architecture (MDA) approach [16] to automate

certain stages of the environment and application development. The MDA approach supports the separate description of key concerns at several abstract levels through different models definition. In our case, we express all the concerns at three abstract levels. According to the Software Product Line, each application is typically developed in a two-stage process, *i.e.* a domain engineering stage and a concurrently running application engineering stage [14]. At the domain engineering stage, the domain model specification tool allows for the definition of a set of reusable components and of a reference architecture for a specific domain with SOC concepts. Figure 2 shows the three different modeling levels. Two successive levels are related through model transformations. The specific domain model is an instance of the extended meta-model and is independent of any target platform. However, a domain application developer normally concentrates on the domain business logic implementation especially when they may lack sufficient SOC paradigm knowledge. In order to make technical paradigms transparent for the domain developers or technicians, a specific domain model is transformed into two different models. This separate model transformation successfully extracts the services notions from a domain component service defined in the specific domain model. The model transforms a domain service element as a domain component as well as the service binding between two service descriptions as the connector between two domain components. It includes only the domain components specifications in terms of the product line approach. Another implicit model is formed of the hidden information related to service concepts corresponding to each domain component and their relationships. The first model transformation has automated the generation of the application development environment.

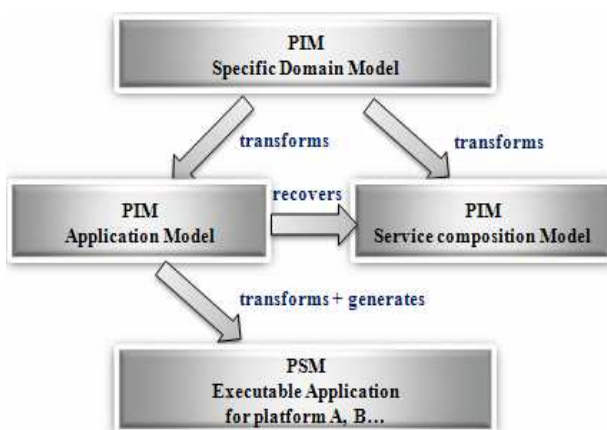


Figure 3. **Modeling levels and model transformations.**

Domain technicians can assemble all reusable domain components to an application model in terms of the

domain composition architecture defined at the first abstract level in the figure 3. On top of an application model specification, the executable application has to be built upon a generic code generator related to a target platform and a domain customized concrete service repository. In addition, an application model can be parameterized and evolved to adapt new individual requirements before the code generation. The application developer needn't reconfigure the application model according to the different target platforms due to the second abstraction level still independent and an implementation selection mechanism. However, an application model defined within the specific editor does not necessarily conform to the specific domain model in SOC because the application elements have lost indispensable information referring to service notions. Hence, we added an implicit inverse model transformation before the code generation in order to recover these missing service concepts and combine them to produce a complete application model conforming to the specific domain model. The second model transformation should be invisible for the domain technicians and/or application developers. An executable application can be built automatically from the complete application model with an implementation selection mechanism related to the target platform, ex. iPOJO and OSGi. In the next section, we will propose an example to express the details of the two model transformations and the two stages process of application development.

5. Example

We have successfully used the presented environment to describe the ANSO domain (Autonomic networks for SOHO) and to automate home application developments. According to the meta-model, the domain-specialization tool permits to explicitly express the dynamic characteristics of the ANSO domain in the domain model. Figure 4 presents part of the specification of the ANSO domain at the domain engineering stage.

The domain expert using the domain-specialization tool describes the domain concepts in terms of SOC concepts. First, expert has to specify the domain-specific data types, including some Java data types for the other domain concept definitions, for example Recipient. Then, the existing functionality concepts of the domain, for instance "MessageSenderService," are described as abstract service concepts. Each abstract service defines a set of available operations. In terms of SOC concepts, when a service provider publishes a service, a corresponding service description is sent into a service repository plus some properties, in our case, it's the service description repository. The IT expert successively describes the service descriptions based on abstract services. Each

service description describes only one abstract service and some properties (Service Property and Service Configuration Property). According to the different properties definitions, the service descriptions are not the same element of the application whereas several service descriptions describe the same abstract service. For instance, “MailSenderService” and “SMSSenderService” describe the same abstract service “MessageSenderService”, the two service descriptions as different elements of the domain application because of a specific “Service configuration property smtpServer” within “MailSenderService”. The “Service Property” value is assigned by the expert; The “Service configuration property” is specified by the expert but its value will be assigned at design time. In addition, the concrete service repository collecting existing service implementations is defined by the domain expert and the platform status is updated dynamically. Finally, the domain expert has to specify the reference architecture of the domain application in terms of product line architecture. The interactions (Service Binding) between service descriptions are the important elements of the service composition. For example, in our case, the domain expert specifies a Service Binding “AlarmManagerToPopupSender”. It presents a connection from the service description AlarmManager to the service description PopupSender. Its attributes “Dynamic = true”, “Optional = false” and cardinalities are assigned.

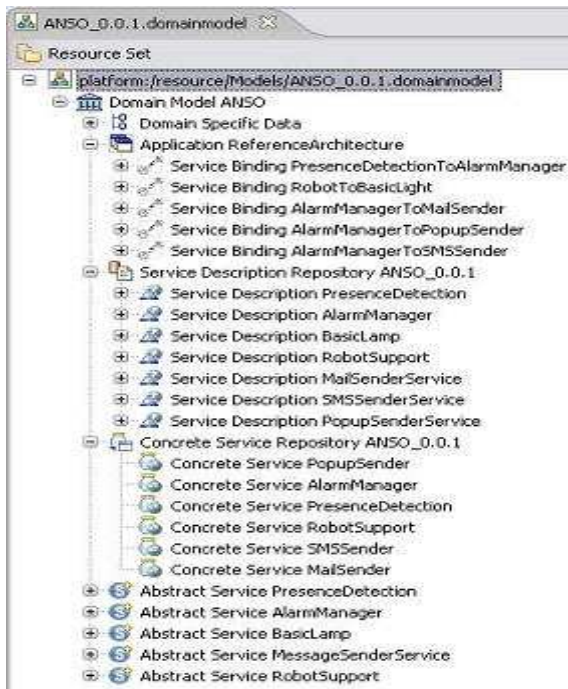
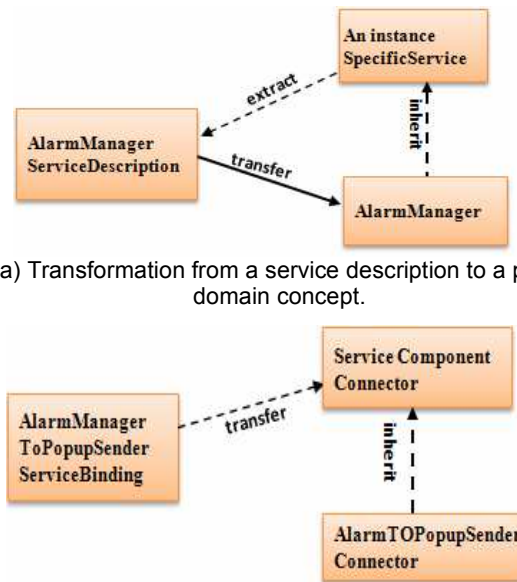


Figure 4. Autonomic networks for SOHO services example

This description indicates that an AlarmManager service can dynamically interact with zero or multiples instances of the PopupSender service by means of a Client/Server communication model at runtime.

From this ANSO domain model specification, we automatically obtained an application development tool by applying the MDA approach via the first model transformation presented in the previous section. The AlarmManager Service Description is extracted to form a specific service and then transformed as an AlarmManager component inheriting the Specific Service. The example shows the relation of the transformed model and the specific domain model in the figure 5(a). A service binding between two service descriptions is transformed in a similar manner as shown in the figure 5(b). The concepts of Specific Service and Service Component Connector are included in the specific service model and forced to be implicit for the domain application developer.

The second model transformation from a home application model to a dynamic service composition model is an inverse direction transformation relative to the first model transformation. It enables an incomplete application model to implicitly recover the missing service information and then automatically produce an executable application with a code generator for the target platform iPOJO and OSGi of the domestic domain. The code generator is capable of packaging the generated application and then to deploy it.



a) Transformation from a service description to a pure domain concept.

b) Transformation from a service binding to the components connector.

Figure 5. Transformation from domain model to application model.

6. Conclusion

The main contributions of this paper are first to show how a model-based, domain-centric development environment reduces the difficulties related to service composition. Today, few modeling tools support modeling dynamic service composition configurations. As a result of a SOC meta-model extension, the presented tool allows the explicit modeling certain service composition characteristics in a dynamic environment context like home computing. A service composition configuration can decide on service implementations selections at any binding time. The tool also supports structural modifications of a service composition configuration at runtime. In addition, the status of the targeted platform is dynamically updated by the concrete service repository, which improved the flexibility and dynamicity of this development environment. The MDA approach accelerated our tools development and evolution. The model transformations automated the implementation of an abstract service composition as well as the environment implementation.

In this paper, we already have a good experience in applying the Software Product Line approach into the service orientation technology for the verification of the service composition correctness within a given context. However, variability and its management are an important part of the software product line engineering because the variability is the ability of a system to be efficiently extended, changed, customised or configured for use in a particular context [18]. Therefore, we propose as our future work a pure domain meta-model separated from the SOC domain concepts. Furthermore, we attempt to import a variability management modelling into this pure domain model. Then, it enables the domain expert to always concentrate on the domain business logic. In addition, this pure domain model with variability concepts might be transformed in order to conform to several technological platforms but not only to service-based platforms. This would include the Fractal technology that is a general software composition framework supporting component-based programming.

7. References

- [1] P. Lalanda and C. Marin, "Domain-configurable development environment for service-oriented applications", IEEE Software, 2007.
- [2] André Bottaro, Anne Géroddolle, Philippe Lalanda "Pervasive Service Composition in the Home Network", AINA 2007:596-603
- [3] Clément Escoffier, Richard S. Hall, Philippe Lalanda: iPOJO: an Extensible Service-Oriented Component Framework. IEEE SCC 2007:474-481
- [4] J.Bourcier, C.Escoffier and P.Lalanda, "Implementing home-control applications on service platform", 4th IEEE Consumer Communications and Networking Conference (CCNC'07), January 2007.
- [5] C. Marin, Ph. Lalanda, D. Donsez: "A MDE Approach for Power Distribution Service Development". ICSOC 2005: 552-557.
- [6] Clément Escoffier, Johann Bourcier, Philippe Lalanda, Jianqi Yu "Towards a home application server" IEEE CCNC 2008
- [7] Jianqi Yu, Philippe Lalanda "Integrating UPnP in a development environment for service-oriented applications" IEEE ICIT 2008
- [8] M. Papazoglou and D. Georgakopoulos. Guest editor introduction: Service oriented computing. ACM SIGSOFT Software Engineering Notes, 46(10):24–28, 2003.
- [9] Papazoglou, M.P., Paolo Traverso, Schahram Dustdar and Frank Leymann "Service-Oriented Computing Research Roadmap"
- [10] Uwe Zdun and Mark Strembeck "Modeling Composition in Dynamic Programming Environments with Model Transformations" Software Composition 2006:178-193
- [11] P. Clements, L. Northrop. "Software Product Lines: Practices and Patterns" Addison-Wesley Professional, 2001.
- [12] André Bottaro, Johann Bourcier, Clément Escoffier and Philippe Lalanda "Autonomic Context-Aware Service Composition"
- [13] Mikko Raatikainen, Katrine Jokinen, Paavo Kotinurmi, Varvana Myllarniemi "Service Composition Using Product Configuration Technology"
- [14] Timo Asikainen, Tomi Mannisto, Timo Soinen "Kumbang: A domain ontology for modelling variability in software product families" Advanced Engineering Informatics 21 (2007) 23–40
- [15] Hiroshi Wada, Junichi Suzuki and Katsuya Oba "Modeling Non-functional Aspects in Service Oriented Architecture"
- [16] C. Atkinson and T. Kuhne. "Model-driven Development: A Metamodeling Foundation." IEEE Software, P36–41, 2003.
- [17] K. Czarnecki. "Overview of Generative Software Development", UPP, Mont Saint-Michel, France, LNCS 3566, pp. 313–328, 2005.
- [18] M. Papazoglou. Service-oriented computing: concepts, characteristics and directions. Fourth International Conference on Web Information Systems Engineering, 2003
- [19] Challenges for Today's Consumer Electronics and White Goods Manufacturers www.osgi.org/Markets/SmartHome