PMG-Pro: A Model-Driven Development Method of Service-Based Applications

Selo Sulistyo and Andreas Prinz

Faculty of Engineering and Science, University of Agder,
Jon Lilletuns vei 9, N-4879 Grimstad, Norway
{selos,andreas.prinz}@uia.no

Abstract. In the Internet of Things, billions of networked and software-driven devices will be connected to the Internet. They can communicate and cooperate with each other to form a composite system. In this paper, we propose PMG-pro (present, model, generate and provide), a language independent, bottom-up and model-driven method for the development of such composite system. We envision that all devices in the Internet of Things provide their functionalities as services. From a service description, a service presenter generates source code (i.e., for the service invocations) and uses an abstract graphical representation to represent a service. The code is connected to the abstract graphical service representation. A service abstractor constructs the abstract graphical representations even more abstract in hierarchical service taxonomy. Software developers use the abstract graphical service presentations to specify new service-based applications, while the source code is used for the automation of code generation.

1 Introduction

Today's Internet technology is mainly built for information sharing. Information providers, which typically are implemented as servers, provide information in the form of web pages that can be accessed by internet clients. In the *Future Internet* [19], various independent networked computing devices from small devices (mobile devices, embedded systems, etc) to powerful devices (desktops and servers) may be easily connected to the Internet, in a plug and play manner. These devices can communicate and cooperate with each other. The Internet of Things is one of the popular terms illustrating the *Future Internet*.

From the software developer's point of view, the '*Thing*' in the Internet of Things can be seen as all kinds of networked devices that are driven and delivered by (embedded) software. Considering that the device's functionalities are provided as services, we will have billions of services in the Future Internet. Composing these services is a challenge for the development of service-based applications. Unfortunately, traditional software engineering approaches are not fully appropriate for the development of service-based applications. There is an urgent need for developing comprehensive engineering principles, methodologies and tools support for the entire software development lifecycle of service-based

I. Ober and I. Ober (Eds.): SDL 2011, LNCS 7083, pp. 138–153, 2011.

applications [24]. Within the European Community, composing services in the Internet of Things has been proposed as one of the research agendas during 2010-2015 [1].

Model-driven development (MDD) is considered as a promising approach for the development of software systems. With this approach, models are used to specify, design, analyze, and verify software. It is expected that running systems can be obtained from executable models. One approach is to transform models into source code conforming to existing programming languages, such as Java and C++ (i.e., code generation). A more distinct approach is model interpretation. The transformation approach has several benefits, because the resulting code can be easily inspected, debugged, optimized and becomes more efficient. In addition, the generated implementation is easier to understand and can be checked by the compiler.

However, fully-automated code generation is a difficult task. In the context of the Model-driven Architecture (MDA) [16], unclear definitions of platform models and also a big variability of device capabilities and configurations are two reasons why it is difficult. If a fully-automated code generation can be achieved, it is restricted only to a specific domain in the context of domain-specific modeling (DSM). The automation in the DSM is possible because of domain-specificity where both the modeling languages and code generators fit to the requirements of a narrowly defined domain. In this paper we propose PMG-pro, a language-independent, bottom-up and model-driven development method of service-based applications. To achieve a fully-automated code generation, we adapt the DSM concepts, where we construct models as a representation of real things.

The remainder of the paper is organized as follows: In Section 2 we present a development scenario of a service-based application. Then, in Section 3 we present the PMG-pro method. Based on the scenario, we illustrate the use of PMG-pro. Section 4 is devoted to related work. Finally, we draw our conclusions in Section 5.

2 Service-Based Applications: *A Scenario*

A typical example of an environment containing embedded services is a smart home where a residential gateway is controlling and managing home devices with embedded services. A smart home example was also used in [25] and [5]. In this type of dwelling, it is possible to maintain control of doors and window shutters, valves, security and surveillance systems, etc. It also includes the control of multimedia devices that are parts of home entertainment systems. In this scenario, the smart home is containing the following devices:

1. WeatherModules that provide different data collection services (i.e., air temperature, solar radiation, wind speed, and humidity sensors).
2. Lamps that provide on-off and dimmer services.
3. Media Renderers that provide playing of multimedia services.
4. Virtual devices that provide sending e-mail services.

We consider that all the devices have been implemented their functionalities as embedded services. Different open and standardized languages and technologies may have been used to describe the services. For example, Universal Plug and Play (UPnP) is one of the popular standards for describing embedded services of home entertainment devices (e.g., media renderer). Other examples are Web Services Description Language (WSDL) and Device Profile for Web Services (DPWS). Based on these different embedded services, new applications can be promoted. In this scenario, a new application is promoted with the following new functionalities:

- the application is able to send notifications (e-mail) when the air temperature from the WeatherModule is greater than 50°C,
- it will play music/songs on the Media renderer when the light is turned on and the solar radiation is below than 10, and
- the application provides two new UPnP services. The first service is to enable a user to configure the song to be played for a specific weather condition, while the second service is to get the configuration information.

3 The PMG-Pro Method

Different approaches can be used for developing the application scenario mentioned above, for example bottom-up and top-down development. Since embedded services can be abstract (i.e., models) or concrete (i.e., real implementations at run-time), the development of service-based applications can use both approaches. Bottom-up development approaches assume that all abstract services have concrete services, while in top-down approaches abstract services may be services that will be implemented in the future. Since we will use existing (i.e., implemented) embedded services, we consider that the bottom-up development approaches are suitable for the development of service-based applications.

With regard to the software production, there are different approaches, which focus on how to specify, design, implement, test, and deploy software systems. They can be categorized as implementation- and model-oriented approaches. The implementation-oriented approach focuses on the implementation. Although there may exist design descriptions and models, they are not formal and are often incomplete. In contrast, the model-oriented approach focuses on the descriptions of the functionality or the properties of the system. These descriptions constitute more or less formal models of the system and they can be verified, analyzed, and understood independently of the implementation of the system.

PMG-pro combines bottom-up and model-driven development approaches to promote a rapid and automatic development of service-based applications. As it is indicated by the name, PMG-pro (Present, Model, Generate, and provide) has four steps (i.e., presenting, modeling, generating, and providing). PMG-pro combines the generating and providing as one step.

The PMG-pro architecture (see Fig. 1) consists of three main parts: the presenter/abstractor that is used at the presenting step, the modeling editor that

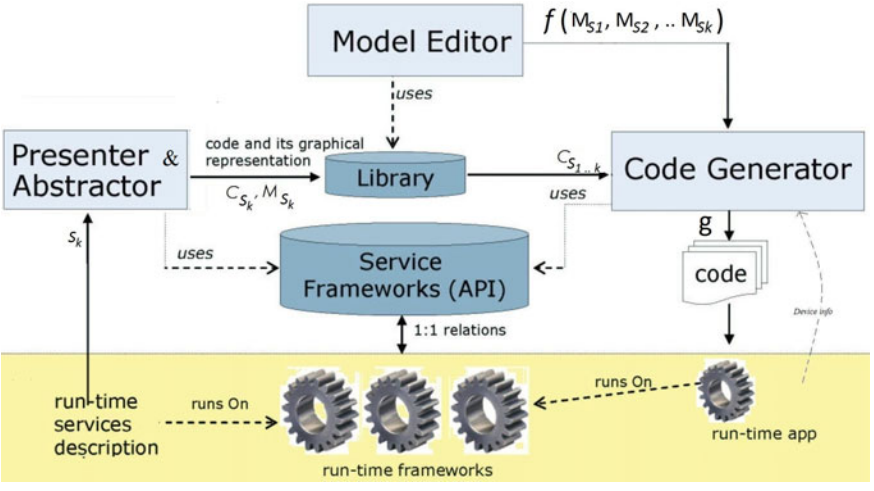

Fig. 1. The PMG-pro Architecture

is used at the modeling step, and the code generator/provider that is used at the generating/providing step. There is also a library that is used to store the generated source code and service models at the presenting step. Each service model (abstract) stored in the library binds to source code (concrete). Within the DSM context, the generated service model represents a real thing as it has been implemented (i.e., embedded services). At the modeling step, based on the stored service models, developers can specify (i.e., produce) new models of service-based applications using selected the modeling editor. At the generating step, referring to the library the models of a service-based application is transformed to source code.

3.1 The Presenting Step

Specifying models of a service-based application is only possible if the models of the included (i.e., existing) services are in place. The presentation step consists of a transformation mechanism of service descriptions into graphical representations and source code. It involves re-engineering or reverse engineering processes. For example, in the context of Web services, we can use WSDL2Java [8] to re-engineer (WSDL) service descriptions into source code. The source code is used as a proxy for service invocations to the concrete services that reside in service providers. The graphical representation will be used by the developers in order to be able to work in a model-driven fashion.

From a service description (s), the abstractor/presenter generates a graphical service model ($\mathbf{M}_s$) conforming to a selected modeling language and source code ($\mathbf{C}_s$) conforming to a selected programming language. To automate the transformation process, existing service frameworks and APIs (e.g., the Web Service framework and API) are used. PMG-pro uses `Cyberlink for Java` for

the UPnP services [12], `WS4d` [6] for the DPWS services, and `WSDL2Java(Axis)` [8] for the Web services. Obviously, this can be extended for other service frameworks and APIs.

Depending on the selected modeling language, different graphical representations (i.e., notations) can be used to represent the existing services. UML classes, CORBA components, Participants in SoaML, or SCA components are among them. However, it must be noted that within the context of domain-specific modeling (DSM), a graphical representation must relate to a real thing which in this case is the implementation of the service. Therefore, it is important to keep the relation (bindings) between graphical representations (i.e. service models) and source code (i.e. implementation for the service invocations). Fig. 2 shows the relation between a service description, its model, and source code.

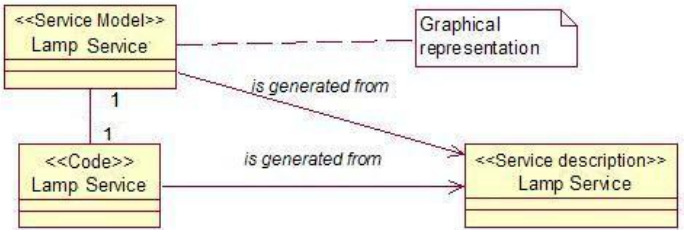

Fig. 2. The relation between a service description, its model and source code

Different service frameworks and APIs have been developed using different programming languages and run on different platforms, helping developers to implement services. For example, in the Web services context there are Apache Axis (Java and C++), appRain (PHP), .NET Framework (C#, VB and .NET), etc. Therefore, a graphical representation of a service may have several implementations (i.e., source code). This source code may also use different programming languages and may run on different platforms. Thus, the `1..1` relation (model-source code) in Fig. 2 can be a `1..*` relation.

A Common Service Model. Different standards and technologies may be used to describe services, although they have similar functionalities. In PMGpro, from a service description pair of source code and graphical representation is generated. For similar services, there will be a common service model. An example of a common service model is shown in Fig. 3. The figure shows three different services that provide similar functionalities, but they use different standards and technologies for describing their services. The UPnP_Lamp service and the UPnP_Light use UPnP schema to describe services while the DPWS_Lamp uses DPWS to describe the service. However, all lamp services provide functionalities to switch ON-OFF the light even though they use different operation names. From these different service models a common model for the Lamp services which has ON and OFF functionalities can be constructed.

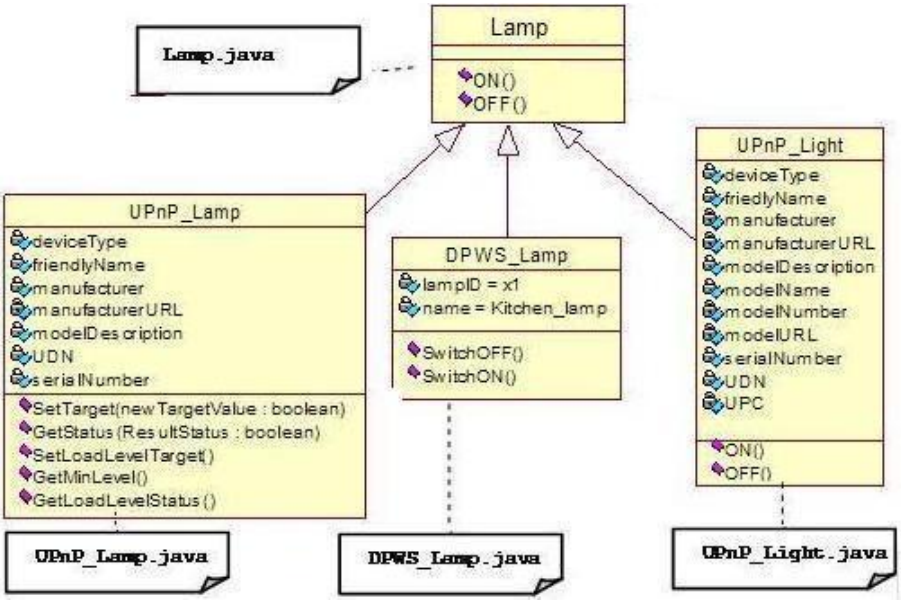

Fig. 3. A platform independent service models and its code

Listing 1.1. Lamp Code

```
public class Lamp {
...
public void Lamp(){
UPnP_Lamp upnp_lamp= new UPnP_Lamp();
UPnP_Light upnp_light=new UPnP_Light();
DPWS_Lamp dpws_lamp = new DPWS_Lamp();
}

public void ON (String selectedService){
  if (selectedService.equalsIgnoreCase("UPnP_Lamp")) {
        upnp_lamp.SetTarget(true);
  } else if  (selectedService.equalsIgnoreCase("UPnP_Light")) {
        upnp_light.ON();
  } else if (selectedService.equalsIgnoreCase("DPWS_Lamp")) {
        dpws_lamp.SwitchON();
  }

public void OFF (String selectedService){
   if (selectedService.equalsIgnoreCase("UPnP_Lamp")) {
        upnp_lamp.setTarget(false);

.........
}
```

A common service model can be seen as a platform-independent model. In PMG-pro, a common service model binds also to source code or a script. The script provides information about what possible source code can be used at the code generation step. Listing 1.1 code above illustrates an example of source code of the common service model for the Lamp service. This is just an example to illustrate the idea of platform-independent service models. The use of scripts will facilitate the realization of the idea. Moreover, at the moment, the example is a manual implementation of an old feature provided by object languages, the polymorphism.

Service Taxonomy and Ontology. In general, taxonomy can be defined as a classification of things, as well as the principles underlying such a classification. A hierarchical taxonomy is a tree structure of classifications for a given set of things. On the other hand, ontology is a data model that represents a set of concepts within a domain and the relationships among the concepts. A common service ontology and taxonomy enables us to identify the common characteristics of services that fall into a particular category. In the context of service-oriented architectures (SOA), the service taxonomy introduced in [4] is an example service categorization. In [4], services are categorized as *Bus Services* (Communication Services and Utility Services) and *Application Services* (Entity Services, Capability Services and Activity Services, and Process Services).

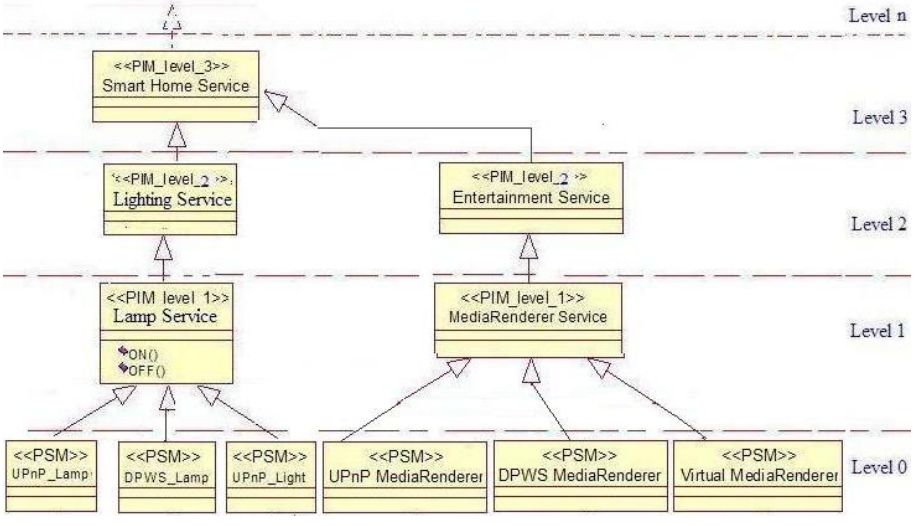

Fig. 4. Taxonomy of service models

In PMG-pro, the hierarchy of service models is stored (at the moment manually) in the service library. For automatic generation of the hierarchy the service description must contain enough information for the service categorization.

Categorization enables with the manageability of services, which can help with the discoverability for code generation.

Fig. 4 shows an example of service taxonomy in a smart home domain, where different implementations of `MediaRenderer services` (e.g. UPnP and DPWS) are categorized under the `MediaRenderer services`, `Entertainment Services`, and `Smart home services`, respectively. All `MediaRenderer` services at level 0 of the hierarchy have graphical representation (with `<<PSM>>` stereotype) and source code (with `<<code>>` stereotype). MediaRenderer services at level 1 of the hierarchy are defined as a common model to represent all media renderer services at level 0. Accordingly, the `Lamp services` have a similar hierarchy. It should be noted that all the service models at any level of the hierarchy finally relate to source code.

At level 1 and above, there is a similar relation between model and code. The service models at level 1 and above are platform-independent models and denoted with `<<PIM_level_n>>`. At a very high level of the hierarchy it would be kind of a script that contains information about possible target platforms that can be used in the code generation step.

An Example: *Arctis Building Blocks.* In this case study we use Arctis. Arctis [13] is a modeling editor that uses a building block as a basic entity. Obviously, different modeling editor and tools can be used. For this purpose, we have developed a service abstractor/presenter that is able to transform UPnP service descriptions into Arctis building blocks and its connected source code (Java class). An Arctis building block is considered as an encapsulation of activities which can be accessed through its ports. In this case, one building block can be used to represents several services.

Table 1. Transformation rules between UPnP and Arctis Building Block

UPnP	**Arctis Building Block**
Device name	Building block name
Action	Parameter input
Action argument	Parameter type (input)
State variable	Parameter output
Type of state variable	Parameter type

A UPnP device has two kinds of descriptions; device description and service description. A UPnP device can have several services that are in a UPnP service description called Actions. To automate this step we use transformation rules. Table 1 shows transformation rules to transform different properties in a UPnP service description into properties in an Arctis building block. To construct the transformation rules, both Arctis and UPnP meta-models are required. However, the rules are very simple. For example, to present the name of the building block, we use the name of the UPnP device. Obviously, other XML-based service descriptions (e.g., WSDL, DPWS) will use a similar process.

In Fig. 5, three building blocks (i.e., WeatherModule, Lamp and MediaRenderer) representing services are shown. Each of these building blocks has source code (a Java class for service invocations). Obviously, other programming languages can also be used. After all service descriptions are transformed into graphical service models the modeling step can be started.

3.2 The Modeling Step

In software development, models are used to describe the structures and the behaviors of the software systems. The structure specifies what the instances of the model are; it identifies the meaningful components of the model construct and relates them to each other. The behavioral model emphasizes the dynamic behavior of a system, which can essentially be expressed in three different types of behavioral models; interaction diagrams, activity diagrams, and state machine diagrams. We use interaction diagrams to specify in which manner the instances of model elements interact with each other(roles).

Even though for model-driven development, state-machine diagrams are considered as the most executable models, we are still interested in using UML activity diagram and collaboration diagram. The reason is that from activity diagrams we can generate state machine diagrams [14]. The UML activity diagrams are used mostly to model data/object flow systems that are a common pattern in models. The activity diagram is also good to model the behavior of systems which do not heavily depend on external events.

PMG-pro is a language-independent method. It is possible to use different existing modeling languages and different modeling editors. This is done by developing and implementing different service abstractors/presenters. The requirement is that the presenter must generate notations (i.e., abstract service models) that conform to the chosen modeling languages. Using the abstract service models $(\mathbf{M}_{s1}, \mathbf{M}_{s2}, ..., \mathbf{M}_{sk})$, a service-based application can be expressed in a composition function $f\{\mathbf{M}_{s1}, \mathbf{M}_{s2}, ..., \mathbf{M}_{sk}\}$, where $s1..sk$ are the included services in the service-based application.

To demostrate the language-independent feature, we have developed prototypes (i.e., service abstractor/presenter) that supports Arctis [13] and Rational Rose [20]. Using Arctis, behaviors of a service-based application are modeled using collaboration activity diagrams while using UML (Rational Rose), the behaviors are modeled using sequence diagrams. Obviously, the semantics of the language follows the chosen modeling languages.

An Example: *Collaboration Activities in Arctis.* Using the service taxonomy (see Fig. 4), it would be possible for the modelers to model new service-based applications using service models at any level of hierarchy. The higher the level of hierarchy the more platform-independent the models would be. By high-level models, we mean models of service-based applications that are built using platform-independent service models at level 1 and above of the hierarchy in the service taxonomy. From high-level models of service-based applications different

source code can be generated. Obviously, it will be limited by the number of source code binds to the service models that are stored in the service library.

Using Arctis, a service-based application can be specified as a building block diagram. The behavior of the new service-based application is defined by interactions between building blocks, which are in this case defined using activity nodes defined in UML 2.0. Accordingly, the semantic follows the semantic of UML 2.0 activity diagrams. Fig. 5 shows an Arctis model of the service-based application defined in the scenario. There are four building blocks that represent different existing services mentioned in the scenario. In this model, the service models are taken from the level 1 of the service hierarchy (see Fig. 4). This means that they may have several different implementations.

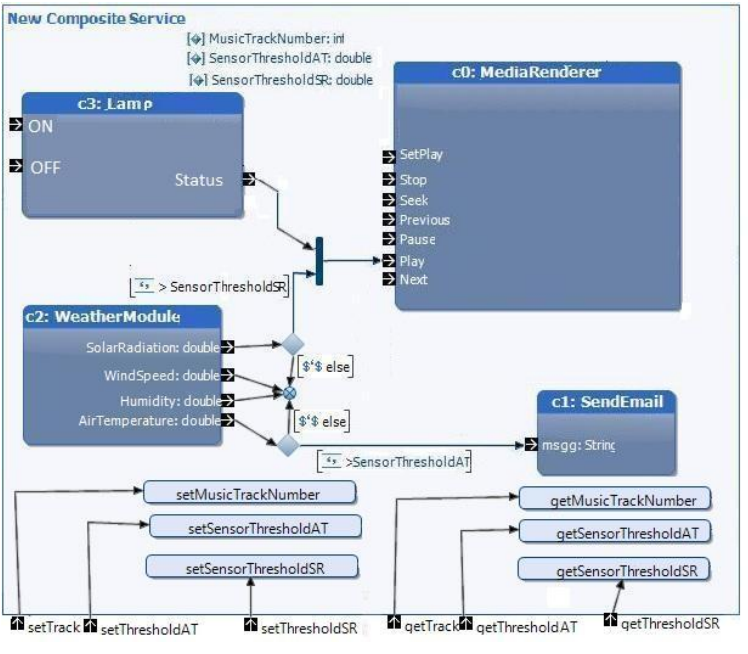

Fig. 5. The Arctis model of the new application

In Arctis, a service-based application can be, again, encapsulated and presented as a new building block. This allows us to model the new provided services defined in the scenario. As defined in the scenario the new service-based application will provide two new services (i.e., SettingService and GetConfiguration Service). In Arctis, this can be specified by defining new parameters (i.e., ports). Obviously, this new building block can be used to build a new building block diagram as new specification of a software system. This is the way how an incremental development of a large application can be done in Arctis.

3.3 The Automated Code Generation Step

For model execution, we use code generation approaches instead of model interpretations. For this, we did not use any transformation language to generate code, but a Java program to transform models into texts (i.e., source code). At the moment, the program can only read collaboration activity diagrams (i.e., Arctis diagrams) and sequence diagrams (i.e., UML sequence diagrams). The potential code generation from activity diagrams was studied in [3] and [7]. For a tool, Enterprise Architect from Sparx Systems [21] is an example for modeling tools that support code generation from activity diagrams.

The code generation process of a service-based application can be expressed as a generation function $g[f\{\mathbf{M}_{s1}, \mathbf{M}_{s2}, .., \mathbf{M}_{sk}\}, \mathbf{C}_{s1..sk}, dev\_info] \Rightarrow code$, where $f\{\mathbf{M}_{s1}, \mathbf{M}_{s2}, .., \mathbf{M}_{sk}\}$ is the model of the service-based application, $s1, s2$ .. sk are the included services, $\mathbf{C}_{s1..sk}$ are the connected code of the used service models ($\mathbf{M}_{s1..sk}$), and dev_info is the given device information (i.e., the capability and configuration information).

Configuration and Capability Matching. In the context of MDA, different target platforms require different tailored code. In the context of embedded systems, the terms of device capability and configuration are often used instead of platforms. Device capability and configuration introduce problems for code generation even within the same device capability. For example, it can occur the development of mobile applications. In this case, it can happen that a working source code for one specific mobile phone will not work on another one of the same type. For instance, this can be caused by a difference in the user configurations (e.g. memory size) or/and in the device capabilities (e.g., the resolution of camera).

Services in a service-based application environment are considered being provided by third parties; therefore they can easily come and go. It would be very possible that included services are not available when a service-based application is implemented, deployed, and run. To solve the problem, two solutions are possible. The first is to generate code for possible aggregated services in the ontology. All possible device configurations would also support run-time adaptation in case of services are removed or new services appear.

The second possible solution is to generate code only for the present (specific) services. This solution requires information about what devices are available at runtime. This means code cannot be generated before the platform is used, which essentially means on-demand code generation.

An Example: *From Arctis models to Code.* To illustrate the code generation in PMG-pro, we use the Java programming language. To generate code from the structure the block diagram and building blocks are used. From a building block diagram the main application (class) is generated. The name of the class is defined using the name of the building block diagram.

From each building block, one object is instantiated. Since the building blocks in this scenario are platform independent, the objects to be instantiated are depending on the platform selection. Listing 1.2 shows an example of the generated code when the UPnP platform is selected at the generating step. Only classes that implement UPnP services are instantiated. `Get` and `Set` methods are generated for all the introduced variables.

Listing 1.2. New Composite Service

```
public class New_Composite_Service {

private UPnP_MediaRenderer c0;
private SendMail c1;
private UPnPWeatherModule c2;
private UPnP_Lamp c3;
MusicTrackNumber int;
SensorTresholdAT double;
SensorTresholdSR double;

public New_Composite_Service() {
    // main behavior
 }

public void setMusicTrackNumber (int number){
    this.MusicTrackNumber=number;
 }

public void setSensorTresholdAT(double sensorTresholdAT){
    this.SensorTresholdAT=sensorTresholdAT;
 }

public void setSensorTresholdSR (double sensorTresholdSR){
    this.SensorTresholdSR=sensorTresholdSR;
 }

public int getMusicTrackNumber (){
return this.MusicTrackNumber;
 }

.....

public static void main(String [] orgs){
  new New_Composite_Service();
 }
}
```

Code from the behavior parts is taken from the activity nodes. For this we adapt the generation method presented in [3]. With regard to their method, an Arctis building block can be considered as an entity that executes an external action. For example, for the decision node (i.e., the decision node with the `airtemperature` input) the following code is produces.

Listing 1.3. An example of code for the behavior part

```
if (c2.airtemperature >= SensorTresholdAT) {
    c2.msgg=SetMessage();
   }
   else
      break;
```

For the scenario example, two new services are provided as UPnP services. For this, UPnP code must be added. We have implemented a code generator to generate code for the application scenario. Fig. 6 is a screenshot of the running UPnP services. We use deviceSpy software provided by Intel Tool for UPnP technology [11]. It can be seen that the new application (composite service) provides two UPnP services: `GetConfiguration()` and `SettingService()`.

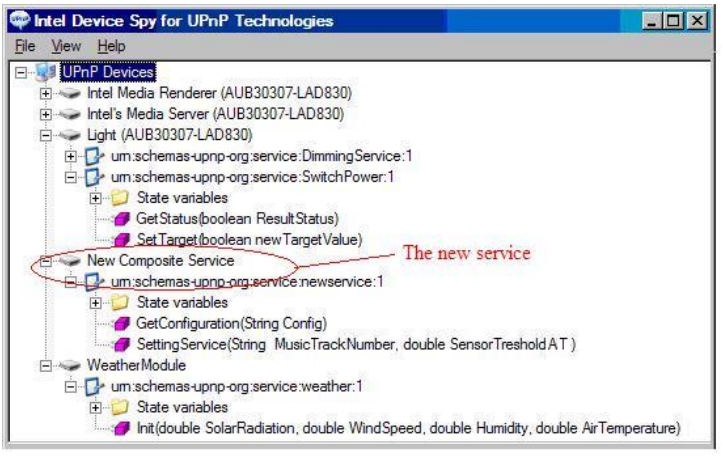

Fig. 6. The new composite service at run-time

4 Related Work

Service composition is gaining importance, as it can produce composite services with features not present in the individual basic services. Several research projects for promoting tools and methods have been conducted, for example the SeCSE project [22]. One of the SeCSE goals is to create methods, tools, and techniques to enable service integrators to develop service-centric applications. The SODIUM [23] project focuses on the need for standards-based integration of heterogeneous services. One thing that makes PMG-pro different from the projects mentioned above is that, while they proposed new modeling languages for the development of service-based applications, PMG-pro uses existing technologies (e.g., modeling languages). PMG-pro focuses on an automated abstraction/presentation of existing services. Depending on the abstractor/presenter,

any graphical presentation conforming to a modeling language can be used to present a service. We construct platform information by maintaining the relation between service models and their code (i.e., their real implementations).

The fact that different perspectives may have different definitions of a service, the definition of service composition may also be different. PMG-pro considers that a service is just a kind of software component model that has evolved from the older software component models (i.e., modules, objects, and components). Here, we use a definition of software component models, which is different from the definition of standardized component models such as CORBA and DCOM. With this definition, a services composition can be done in a similar way as a composition of software units that normally is done at design-time using bottom-up approaches. Thus, PMG-pro focuses on design-time compositions. However, the PMG-pro method can be extended to support run-time compositions. Using the PMG-pro method it would be possible to generate graphical service representation that can be used by end-users (i.e., run-time composition). In the ISIS project for example [26], ICE, an end-user composition, has been developed. A service in ICE is presented as a puzzle with either one input (trigger) or one output (action). By developing a service abstractor/presenter, ICE puzzles for end-users and source code for implementing service invocations can be generated. The ICE puzzles then can be used by end-users to model the composition (at run-time) while the source code is used by ICE to execute the composition.

For the composition of service component models (i.e., software units), composition techniques, and composition languages are required [2]. For the composition languages, there is no particular language that is supposed to be a language for the service compositions. In Web service context, the Web Service Business Process Execution Language (WS-BPEL) [18] and the Web Services Choreography Description Language (WS-CDL) [9] can be considered as a composition language. Within the OMG context, the Service-oriented architecture Modeling Language (SoaML) [17] is another example of composition languages. Also in the Web services context, services orchestration and choreography are well-known service composition techniques. In the context of Service Component Architecture (SCA) [10], wiring can also be considered as a type of composition techniques. For this reason, PMG-pro is language-independent. To show this feature, we support Arctis [13] by developing a presenter to present sevices using building blocks. We also have developed a service presenter to present sevices using UML classes in Rational Rose 2000.

Presenting software functionality into abstract graphical representation has also been studied by other researchers. For example, in [15] UML is used to model Web services. The main contributions of their method are conversion rules between UML and web services described by WSDL. However, their work focused only on Web services and did not think about how automated code generation can be achieved. In [27], software components are visualized using graphical notations that developers can easily understand. They use a picture of

a real device to present a software component. The integration is done by simply connecting components graphically. Obviously, the approach is only applicable for a specific domain. In contrast, PMG-pro is domain-independent.

5 Conclusion

In this paper, we propose PMG-pro, a language independent, bottom-up, and model-driven method for the development of service-based applications. To enable the auotmation, the method adapts the concept of domain-specific languages. Based on the existing service frameworks, APIs, and service descriptions, a service presenter and service abstractor captures platform knowledge statically and presents services using abstract graphical representations. Different notations conforming to modeling languages can be used to represent the services. Each of the generated abstract service model is connected to the code for the service invocations. We use these pairs of code - abstract graphical as a platform-specific model. Ontology is used to construct more abstract the platform models. Service developers can use the abstract graphical service representation to model new service-based applications.

For the code generation, information about the actual targeted platform is required. We have shown two ways of using the constructed platform models. Firstly, based on the information of the targeted platform, the code generator uses the platform models to generate code tailored for the selected target platform. Alternatively, the generated code includes code that contains a detection mechanism to do necessary adaption at run-time. The fully automated code generation is possible, since the service models are connected to code for implementing the service invocations. Just like in domain-specific languages (DSM), the service models are constructed using concepts that represent real things in the application domain.

References

1. Rezafard, A., Vilmos, A., et al.: Internet of things: Strategic research roadmap (2009)
2. Assmann, U.: Invasive software composition. Springer, Heidelberg (2003)
3. Bhattacharjee, A.K., Shyamasundar, R.K.: Validated code generation for activity diagrams. In: Chakraborty, G. (ed.) ICDCIT 2005. LNCS, vol. 3816, pp. 508–521. Springer, Heidelberg (2005)
4. Cohen, S.: Ontology and taxonomy of services in a service-oriented architecture. MSDN Libary Infrastructure Architectures 11(11) (2007)
5. Coyle, L., Neely, S., Stevenson, G., Sullivan, M., Dobson, S., Nixon, P.: Sensor fusion-based middleware for smart homes. International Journal of Assistive Robotics and Mechatronics 8(2), 53–60 (2007)
6. Zeeb, E., Bobek, A., et al.: WS4D: SOA-Toolkits making embedded systems ready for web services. In: Proceedings of Second International Workshop on Open Source Software and Product Lines. ITEA, Limerick (2007)

7. Eshuis, R., Wieringa, R.: A formal semantics for uml activity diagrams - formalising workflow models (2001)
8. Goodwill, J.: Apache Axis Live: A Web Services Tutorial, Sourcebeat (December 2004)
9. Diaz, G., Pardo, J.-J., Cambronero, M.-E., Valero, V., Cuartero, F.: Automatic translation of ws-cdl choreographies to timed automata. In: Bravetti, M., Kloul, L., Tennenholtz, M. (eds.) EPEW/WS-EM 2005. LNCS, vol. 3670, pp. 230–242. Springer, Heidelberg (2005)
10. IBM. Service component architecture (November 2006), http://www.ibm.com/developerworks/library/specification/ws-sca/
11. Jeronimo, M., Weast, J.: UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play. Intel Press, Hillsboro (2003)
12. Konno, S.: Cyberlink for java programming guide v.1.3 (2005)
13. Kræmer, F.A.: Arctis and Ramses: Tool suites for rapid service engineering. In: Proceedings of NIK 2007 (Norsk informatikkonferanse). Tapir Akademisk Forlag, Oslo (2007)
14. Kræmer, F.A.: Engineering Reactive Systems: A Compositional and Model-Driven Method Based on Collaborative Building Blocks. PhD thesis, Norwegian University of Science and Technology, Trondheim (August 2008)
15. Grønmo, R., Skogan, D., Solheim, I., Oldevik, J.: Model-Driven web services development. In: Proceedings of International Conference on e-Technology, e-Commerce, and e-Services, pp. 42–45. IEEE Computer Society, USA (2004)
16. OMG. Model driven architecture guide, version 1.0.1, omg/03-06-01 (June 2003)
17. OMG. Service oriented architecture modeling language (SoaML): Specification for the UML profile and metamodel for services, UPMS (2009)
18. Ouyang, C., Verbeek, E., van der Aalst, W.M.P., Breutel, S., Dumas, M., ter Hofstede, A.H.M.: Formal semantics and analysis of control flow in WS-BPEL. Science of Computer Programming 67(2-3), 162–198 (2007)
19. Papadimitriou, D.: Future internet: The cross-etp vision document. Technical Report Version 1.0, European Future Internet Assembly, FIA (2009)
20. Quatrani, T.: Visual modeling with Rational Rose 2000 and UML, 2nd edn. Addison-Wesley Longman Ltd., Essex (2000)
21. Sparx Systems. Enterprise architect, http://www.sparxsystems.com/products/ea/index.html
22. The SeCSE Team: Designing and deploying service-centric systems: the secse way. In: Proceedings of the Service Oriented Computing: a look at the Inside (SOC @Inside 20 (2007)
23. Topouzidou, S.: Service oriented development in a unified framework (sodium). Deliverable CD-JRA-1.1.2, SODIUM Consortium (May 2007)
24. van den Heuvel, W.-J., Zimmermann, O., et al.: Software service engineering: Tenets and challenges. In: Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems, PESOS 2009, pp. 26–33. IEEE Computer Society, Washington, DC (2009)
25. Wu, C.-L., Liao, C.-F., Fu, L.-C.: Service-oriented smart-home architecture based on OSGi and mobile-agent technology. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews, 193–205 (2007)
26. Su, X., Svendsen, R., et al.: Description of the ISIS Ecosystem Towards an Integrated Solution to Internet of Things. Telenor Group Corporate Development (2010)
27. Yermashov, K.: Software Composition with Templates. PhD Thesis, De Montfort University, UK (2008)