Software Development Methods in the Internet of Things

Selo Sulistyo

Software Development Methods in the Internet of Things

Selo

e-Systems Lab
Department of Electrical Engineering and Information Technology
Gadjah Mada University
Jl. Grafika No. 2, Bulaksumur, Yogyakarta, Indonesia 55281
selo@ugm.ac.id

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 function as a composite system. This paper proposes the AMG (abstract, model and generate) method for the development of such composite systems. With AMG, the development of software application can be done in an automatic manner, and therefore reducing the cost and develpment time. The method has been prototyped and tested with use cases.

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* [16], 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. 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 Internet (i.e., the Internet of Services). A typical example of an environment containing several (embedded) services is a smart home where a residential gateway is controlling and managing home devices with embedded services. 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 multi-media devices that are parts of home entertainment systems. In this scenario, the smart home is containing 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*, and 3*) Media Renderers that provide playing of multimedia services*, see Figure 1 below.

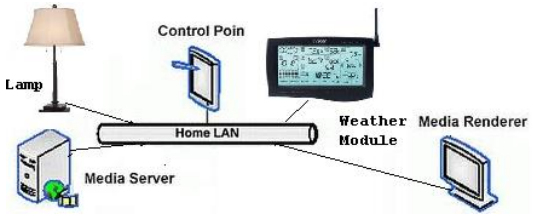

Fig. 1. Smart home services: A Scenario

Combining these independent services (i.e., embedded software) and promoting as a new application is a challenge. 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 applications [19].

This paper proposes a software development method in the Internet of Things. The remainder of the paper is organized as follows: In Section 2 we present terms and definitions of a service and a service-based application. Then, in Section 3 we present the AMG method. Section 4 is devoted to related work. Finally, we draw our conclusions in Section 5.

2 Background

This section gives a background for the paper. It discusses the definition of a service and a service-based application.

What is a Service? A service can be defined in different ways. In [17] for instance, a service is defined as *asset of functions provided by a (server) software or system to client software or system, usually accessible through an application programming interface.* Similar definitions as the one above appear in the context of middleware technologies such as Jini [3], .NET [9], or JXTA [21]. These definitions recognize services as a central element in the system implementation.

In this paper, referring to [1] and [10] a service is defined as a model of software unit. To get an overview of this definition, we have to look at the history of managing the complexity of software systems where a component-oriented architecture is considered as the solution for the complexity problem. Figure 2 illustrates a historical perspective of the use of different models to represent a software unit.

The idea of using software component as defined in [10], can be considered as the birth of today's software component and can be seen as an architectural approach of building software systems. When Assembler was the only available programming language, a routine was considered the first model of software

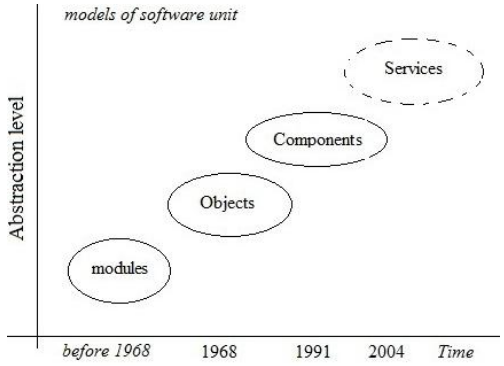

Fig. 2. The historical perspective of the use of different models of a software unit.

units. As the complexity of software systems was increasing a new model of a software unit called a module was introduced. A module is a simple model. However, a module is more abstract than a routine. Accordingly, objects, components and services can also be seen as models of a software unit. The differences between them are their abstraction levels, means of encapsulation and ownership. Software abstraction, encapsulation and reuse are the key points.

With software component-orientation in mind, a single software component might not work as an application. Therefore a composition system is needed. According to [1], a software composition system has three aspects: component models, composition techniques and composition languages. Depending on the models of software unit, see Figure 2, different composition techniques and languages are required. These two aspects have influenced the development approaches and paradigms. For example, when we use objects as a model of a software unit to build a software system we call the paradigm object-oriented development. Accordingly, we have component-oriented development for component oriented-systems and service-oriented development for service-oriented systems. A general concept of service-oriented systems is well-known as Service-Oriented Architectures (SOA)[4].

Service-based Applications Using the SOA concept, architecturally, software applications are built from compound, heterogeneous, autonomous software units called services. If it is the case, service compositions will be a common approach for the development of software applications in the Internet of Services. Software systems and applications are becoming service-based. We call this a service-based application.

A conceptual model of a service-based application is presented in Figure 3. It can be seen that a service-based application may use more than one service. It is shown also that a service can be classified either as `Simple` or `Composite`. In this paper, all services that will be composed are considered as a simple service. As

mentioned earlier, a composite service is a type of a service-based application. It composes services and provides the combined functionality as new services.

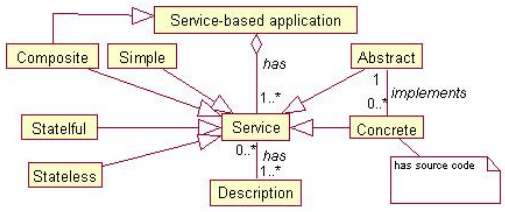

Fig. 3. A conceptual model of service-based applications.

A service can also be classified either as `Abstract` or `Concrete`. An abstract service may have one or more concrete services or it may mean that the service will be implemented in the future. But for the composition of services, this paper considers only Concrete services (run-time services). Furthermore, services can also be classified either as a `State-less` or `State-ful` service. Web services can be considered as an example of stateless services, while UPnP services [8] can be considered as a kind semi state-ful services. UPnP devices use state variables to store the states of specific variables and inform those state changes to other UPnP devices.

3 The AMG Method

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. AMG is a model-oriented approach. Figure 4 shows the idea of the AMG-method.

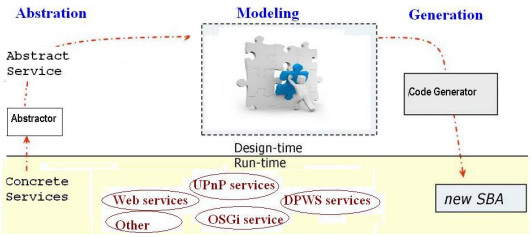

Fig. 4. The AMG (abstract, model, and generate) method.

3.1 The Abstraction Step

Specifying models of a service-based application is only possible if the models of the included (i.e., existing) services are in place. For this an abstraction process is required. The abstraction step consists of a transformation mechanism of service descriptions into graphical representations and source code.

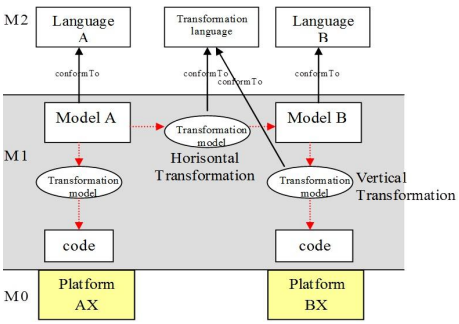

Fig. 5. A conceptual model of model transformation.

The abstration step uses the concept of model transformations [13]. Figure 5 illustrates the concept of model transformation. As shown in the figure, two main types of transformation exists; vertical transformation and horizontal transformation. In the vertical transformation, a source model is at a different level of abstraction than a target model. Examples of vertical transformation are refinement (specialization), PIM-to-PSM transformations and abstraction (generalization). Generalization could mean also an abstraction of platforms or a transformation from code into models.

In tge horizontal transformation, the source model has the same level of abstraction as the target model. Examples of the horizontal transformation are refactoring and merging. In this type of transformation one or more source models are transformed into one or more target models, based on the languages (meta-model) of each of these models. In other words the instance of one meta-model is transformed into instances of another meta-model. So, in this step, we apply the vertical transformation.—

Models can be presented in two forms; graphical (models) or textual. Depending on the relation between these form, there are four different model transformations; M2M (Model-to-Model) transformation, T2M (Text-to-Model) transformation, M2T (Model-to-Text) transformation and T2T (Text-to-Text) transformation. The T2T transformation is often used for the processing of the M2M, M2T and T2M.

Using the illustrated concept earlier, from a service description (s), the abstractor produces 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.

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. 6 shows the relation between a service description, its model, and source code.

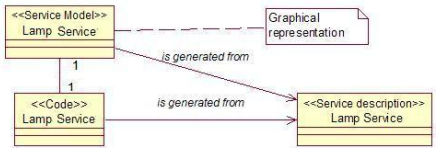

Fig. 6. The relation between a service description, its model and source code

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.

An Example: *UML classes.* 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 the abstraction step we use transformation rules. Table below shows transformation rules to transform different properties in a UPnP service description into properties in an UML class. To construct the transformation rules, both UML and UPnP meta-models are required. However, the rules are very simple. For example, to present the class name, we use the name of the UPnP device. Obviously, other XML-based service descriptions (e.g., WSDL, DPWS) will use the similar process.

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 [11]. 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.

AMG 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 $\mathrm{f}\{\mathbf{M}_{s1}, \mathbf{M}_{s2}, ..., \mathbf{M}_{sk}\}$, where $s1..sk$ are the included services in the service-based application.

An Example: *UML Sequence Diagram.* Using UML classes, the structure of a service-based application can be specified as a class diagram, while the behavior part can be defined using sequence diagram. Accordingly, the semantic follows the semantic of UML 2.0. Fig. 7 shows a UML model of the service-based application defined in the scenario. There are four UML classes that represent different existing services mentioned in the scenario.

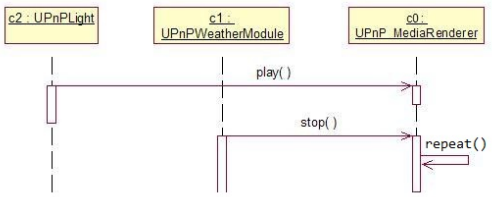

Fig. 7. A service-based application model specified is using a sequence diagram. The composed application will play music when a UPnPLight service is invoked, and will stop it on a certain value of weather parameter.

3.3 The 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). The potential code generation from activity diagrams was studied in [2] and [5]. For a tool, Enterprise Architect from Sparx Systems [18] 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).

Code from the behavior parts is taken from the activity nodes. For this we adapt the generation method presented in [2]. With regard to their method, an UML class 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. To generate code from the structure, from each classes, one object is instantiated. Since the UML classes in this scenario are platform independent, the objects to be instantiated are depending on the platform selection.

4 Discussion

Service composition is gaining importance, as it can produce composite services with features not present in the individual basic services. However, the fact that different perspectives may have different definitions of a service, the definition of service composition may also be different. AMG considers 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). 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.

AMG focuses on service composition at design-time. However, the abstraction step in the AMG can be extended to support run-time compositions. Conceptually, 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 [20], 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). A composition is done by connecting puzzles. Using a specific service abstractor, ICE puzzles for end-users and its source code for implementing service invocations can be generated.

For the composition of service component models (i.e., software units), composition techniques, and composition languages are required [1]. In Web service context, the Web Service Business Process Execution Language (WS-BPEL) [15]

and the Web Services Choreography Description Language (WS-CDL) [6] can be considered as a composition language. Within the OMG context, the Service-oriented architecture Modeling Language (SoaML) [14] 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) [7], wiring can also be considered as a type of composition techniques. For this reason, AMG method does not depend to any particular languages. This can be done by developing different abstractor for different target languages.

Abstracting software functionality into abstract graphical representation has also been studied by other researchers. For example, in [12] UML classes is used to abstract Web services. However, their work focused only on Web services and did not think other possible services. In [22], 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, the AMG method is domain-independent.

5 Conclusion

With regard to the complexity of software systems, the aims for both software composition and model-driven development (e.g. MDA) are similar in which they are used for managing the complexity of software systems and their development. Having benefited from these approaches, this paper propose the use of model-driven development for the development of software applications in the Internet of Things. With this idea, the composition of services can be done using models at different abstraction levels, while the executable composite services can be generated automatically.

References

1. Uwe Assmann. *Invasive software composition.* Springer, April 2003.
2. A.K. Bhattacharjee and R.K. Shyamasundar. Validated code generation for activity diagrams. In *Proceedings of Distributed Computing and Internet Technology*, volume 3816/2005 of *LNCS*, pages 508–521. Springer Berlin / Heidelberg, 2005.
3. Franco Cicirelli, Angelo Furfaro, and Libero Nigro. Integration and interoperability between jini services andWeb services. In *IEEE International Conference on Services Computing*, volume 0, pages 278–285, Los Alamitos, CA, USA, 2007. IEEE Computer Society.
4. Thomas Erl. *Service-Oriented Architecture (SOA): Concepts, Technology, and Design.* Prentice Hall PTR, August 2005.
5. Rik Eshuis and Roel Wieringa. A formal semantics for uml activity diagrams - formalising workflow models, 2001.
6. Juan Jos Pardo et.al Gregorio Daz. Automatic translation of ws-cdl choreographies to timed automata. In Mario Bravetti, Lela Kloul, and Gianluigi Zavattaro, editors, *EPEW/WS-FM*, volume 3670 of *LNCS*, pages 230–242. Springer, 2005.

7. IBM. Service component architecture. http://www.ibm.com/developerworks/library/specification/ws-sca/, November 2006.
8. Michael Jeronimo and Jack Weast. *UPnP Design by Example: A Software Developer's Guide to Universal Plug and Play*. Intel Press, May 2003.
9. Li JiZhe and Yuan YongJun. Research and implementation of lightweight esb with microsoft .net. *Japan-China Joint Workshop on Frontier of Computer Science and Technology*, 0:455–459, 2009.
10. D McIlroy. Mass-Produced software components. In *Proceedings of the 1st International Conference on Software Engineering*, pages 98, 88, 1968.
11. Frank Alexander Kræmer. *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.
12. Roy Grønmo, David Skogan, Ida Solheim, and Jon Oldevik. Model-Driven web services development. In *Proceedings of International Conference on e-Technology, e-Commerce, and e-Services*, volume 0, pages 42–45, Los Alamitos, CA, USA, 2004. IEEE Computer Society.
13. OMG. Meta object facility (MOF) 2.0 Query/View/Transformation specification final adopted specification ptc/05-11-01, 2005. Available at http://www.omg.org/docs/ptc/05-11-01.pdf.
14. OMG. Service oriented architecture modeling language (SoaML) : Specification for the UML profile and metamodel for services (UPMS), 2009.
15. Chun Ouyang, Eric Verbeek, Wil M. P van der Aalst, Stephan Breutel, Marlon Dumas, and Arthur H. M ter Hofstede. Formal semantics and analysis of control flow in WS-BPEL. *Science of Computer Programming*, 67(2-3):162198, 2007.
16. Dimitri Papadimitriou. Future internet: The cross-etp vision document. Technical Report Version 1.0, European Future Internet Assembly, FIA, 2009.
17. Sancho. Definition for the term (software) service, sector abbreviations and definitions for a telecommunications thesaurus oriented database, itu-t, 2009.
18. Sparx Systems. Enterprise architect. http://www.sparxsystems.com/products/ea/index.html.
19. Willem-Jan van den Heuvel, Olaf Zimmermann, and et.al. Software service engineering: Tenets and challenges. In *Proceedings of the 2009 ICSE Workshop on Principles of Engineering Service Oriented Systems*, PESOS '09, pages 26–33, Washington, DC, USA, 2009. IEEE Computer Society.
20. Xiaomeng Su, Reidar Martin Svendsen, et.al . *Description of the ISIS Ecosystem Towards an Integrated Solution to Internet of Things*. Telenor Group Corporate Development, 2010.
21. William Yeager and Joseph Williams. Secure peer-to-peer networking: The jxta example. *IT Professional*, 4:53–57, 2002.
22. Kostyantyn Yermashov. *Software Composition with Templates*. PhD Thesis, De Montfort University, UK, 2008.