Using MDA for Integration of Heterogeneous Components in Software Supply Chains

Herman Hartmann[1], Mila Keren[2], Aart Matsinger[1],
Julia Rubin[2], Tim Trew[1], and Tali Yatzkar-Haham[2]

[1] Virage Logic, Eindhoven, The Netherlands
{herman.hartmann,aart.matsinger,tim.trew}@viragelogic.com
[2] IBM Research, Haifa, Israel
{Keren,mjulia,tali}@il.ibm.com

Abstract. Software product lines are increasingly built using components from specialized suppliers. A company that is in the middle of a supply chain has to integrate components from its suppliers and offer (partly configured) products to its customers. To cover the whole product line, it may be necessary for integrators to use components from different suppliers, partly offering the same feature set. This leads to a product line with alternative components, possibly using different mechanisms for interfacing, binding and variability, which commonly occurs in embedded software development.

In this paper, we describe a model-driven approach for automating the integration between various components that can generate a partially or fully configured variant, including glue between mismatched components. We analyze the consequences of using this approach in an industrial context, using a case study derived from an existing supply chain and describe the process and roles associated with this approach.

1 Introduction

Software product line engineering (SPLE) aims to create a portfolio of similar software systems in an efficient manner by using a shared set of software artifacts. SPLE is usually separated into two phases: domain engineering and application engineering, and a variability model is used to capture the commonality and variability and to configure a variant [1]. When implementing SPLE using model-driven architectures (MDA), it is conventional to create a domain-specific language (DSL) during domain engineering. During application engineering this DSL is used to create specific applications [2]. In component-based software development, a domain-specific component technology is used to create reusable artifacts, and a particular application is created by selecting components and binding variation points [3,4].

Due to the growing influence of software supply chains, an increasing portion of a product line is developed using commercial components [5]. In a supply chain, each of the participants uses components containing variability, combines them with in-house developed components, and delivers components containing variability to the next party in the supply chain [6,7].

J. Bosch and J. Lee (Eds.): SPLC 2010, LNCS 6287, pp. 361–376, 2010.

In an earlier paper we analyzed the consequences of integrating heterogeneous components in a software supply chain for resource constrained devices [8]. When software components from different suppliers have to be integrated, there may be mismatches between their interfaces, which have to be bridged by glue code. For instance, a set of interfaces might contain different numbers of methods (interface splitting), method parameters can be passed in different forms, e.g., as a struct vs. a list of separate parameters, methods having the same name might have different functionality implemented (functional splitting), etc.

A product line must be able to satisfy the requirements of its potential customers. Often, no single supplier can cover the full range of variability needed to achieve this, so it is frequently necessary to use components from several suppliers for a particular functional area. This leads to a product line that contains alternative components, only one of which can be used in a particular implementation [9], and a large number of glue components. In current practice there are three different approaches for integration and configuration of components, each with their limitations, as described below.

1. The possible glue components are created during domain engineering and configured during application engineering using a variability management tool. However, the manual creation of, possibly, a large number of glue components will require an unacceptably high development effort.
2. The required glue component is created during application engineering, at the moment that the specification of that glue component is known. This introduces an increase of throughput time which would be unacceptable in many situations.
3. A common standard interface and component technology is defined for a set of non-matching components. Glue components are created for each component to match these interfaces. Many components will therefore be bound through two glue components. This approach has the drawback that the glue components might become unnecessary complex if the standard interfaces have to cater for the interactions between any combination of components, thereby leading to additional development effort in comparison with the creation of custom glue.

To solve this problem we exploit the power of model-driven code generation to create custom glue between the combinations of supplied components that are actually used.

For resource constrained devices, some component technologies use static binding, rather than dynamic binding, and reachability analysis to exclude unnecessary code [3] and to create optimal system performance. The challenge that arises from a software supply chain is the ability to deliver a partly configured product to the next party in the supply chain. Components that have to be bridged may contain optional subcomponents. The glue components should only bridge between the components that are actually present so glue components should only be generated when the presence of those optional components is known. This can only be determined when the final configuration choices are known. In a supply chain, these final configuration choices could be made by a downstream participant. Furthermore, each supplier and the receiving parties may all use different build environments, which complicates the creation of the complete software stack. Other challenges from a supply chain relate to the protection of Intellectual Property and commercial interest, which means that, in many cases, the customer should neither receive source code, nor be aware of variation points that are offered to other customers.

We therefore address the following **research questions:**

1. Can MDA be used to bridge mismatches between components from alternative suppliers and can this method be used to support staged configuration? The set of mismatches we address in this paper is given in section 2.
2. What is the development process that is associated with this approach and what level of MDA expertise is required by the engineers?

Paper overview: In the paper, section 2 introduces a case study, Section 3 describes our approach and how it is supported with MDA and variability management tools. The management of the expertise expected of engineers is presented in section 4. Section 5 contains a discussion and identifies areas for future research, with Section 6 making a comparison with related art, followed by our conclusions in Section 7.

2 ZigBee Case Study

We demonstrate the applicability our approach by a realistic case study. In order to restrict the complexity of the first evaluation, the capabilities of the glue code in our tools are restricted to adapting between syntactic differences and the differences between component technologies. This caters for cases in which the semantics of interfaces are standardized, independently of the technology that may be used to implement them. As a case, we chose a heterogeneous ZigBee stack. ZigBee is a specification for a suite of high level communication protocols using small, low-power digital radios for wireless personal area networks [10]. The ZigBee stack is defined as a layered protocol (see Fig. 1). The standard is platform-independent, so implementers make their own choices for the exact form of the APIs between the layers. We focus on the lower layers, i.e. Physical, MAC, and Network. There is considerable variation between the network layers for different application profiles, e.g. "Plant Monitoring", "Home Automation" and "Smart Metering", so software suppliers usually only support a few such profiles. The MAC layer is independent of the application profiles, but has to be configured for the particular integrated circuit (IC). Therefore, in order to serve a range of customers, an IC vendor has to integrate ZigBee implementations from alternative suppliers, each of whom made their own software implementation technology choices (in our case nesC [11] and tmCom, a precursor to that used in the MPEG Multimedia Middleware standard [12]).

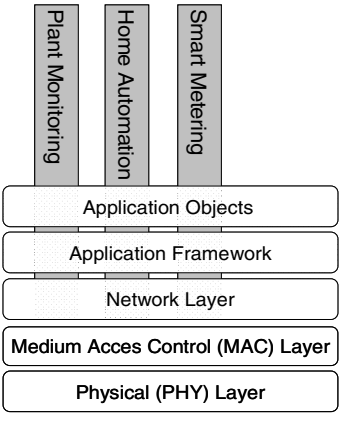

Fig. 1. Layers in the ZigBee protocol stack with profiles

The supply chain consists of IC vendors, software vendors, and the manufacturers that create the final product. In the supply chain, we found multiple parties for each link (typically more than five), each offering different sub-sets and extensions of the ZigBee standard. For each layer, a set of required, alternative and optional features is

specified, together with the dependencies between them. Examples of optional features are: power-saving, guaranteed time slot, and security mechanisms.

In this case study, we take the position of an IC vendor that is using software from specialized suppliers. We investigated the source code of three ZigBee stacks. Table 1 gives a subset of the features of these stacks, whose suppliers we will identify as A, B, and C. For a particular customer, the IC vendor selects the most suitable supplier and creates a partly configured product. For instance, a product can be configured with the components of Supplier C, where "Beaconing" and "Guaranteed Time Slot" are preconfigured, leaving "Power Saving" and "Security" to be configured by the customer.

Table 1. Feature set of selected suppliers

Feature\ Suppliers	**A**	**B**	**C**
Network layer	No	Opt	Opt
Beaconing	Opt	No	Opt
Guaranteed Time Slot	Opt	No	Opt
Security	No	Opt	Opt
Mesh-configuration	No	Man	No
Power Saving	No	No	Opt
Impl. Technology	nesC	C	C

In Table 1, *Man* stands for mandatory, *Opt* stands for optional, and *No* stands for not supported. We also listed the implementation technology. For the MAC layer, an Open-ZB implementation was used [13], based on nesC technology [11]. The differences between the technology choices of the Open-ZB implementation and those of the other suppliers were studied. In order to demonstrate the requirements for glue code generation, a dummy ZigBee Network layer was created, whose interfaces exhibited the union of the differences that had been found, thereby covering all the differences that we encountered in practice. The main differences between the technologies used in the Network component and the Open-ZB MAC are summarized in the Table 2.

Table 2. Differences between the technologies used in the ZigBee case study

	Network layer	Open-ZB MAC layer
Component technology	tmCom	nesC
Binding	Dynamic	Static
Interfaces	Uni-directional	Bi-directional
Naming convention	*tml*<port>*NXP*<interface>[*Ntf*] <method>	<port>_<interface> .<method>
Calling convention	Referencing structure	Individual parameters
Specific keywords	*STDCALL*	*command*, *event*

Both components have two ports (for data and control), which are connected correspondingly. To bridge the technology difference, an additional glue component must be added, as shown in Fig.2. The role of this glue component is to exhibit tmCom style interfaces toward the Network component and nesC style interfaces toward the MAC component. The glue component translates each down-call (*command*) that it receives from the Network component into a corresponding call to the MAC component. In this translation, it deals with naming and parameter-passing conventions. For the up-calls (*events*) originating from the MAC component, nesC uses static binding, based on a configuration description, whereas the tmCom Network layer employs

run-time binding, with a subscription pattern at the granularity of its interfaces [14]. Therefore the glue component implements the notification subscription management, which uses a table to map between the nesC functions called by the MAC and the tmCom functions in the Network layer. The glue code must provide the additional subscription management functions.

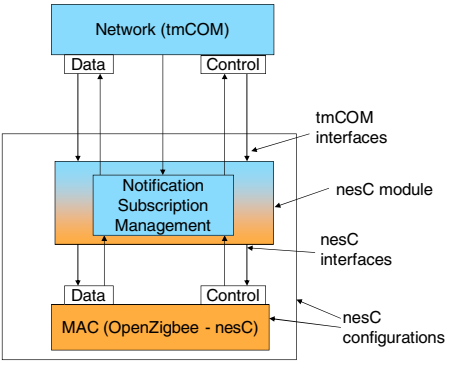

Fig. 2. Integration of Network and MAC layers

During development, staged configuration may be required. Once the supplier has been selected, the remaining choices for optional features shown in the table can either be made in-house or by the customer. For example, when Supplier C is selected, the optional feature "Power Saving" can be selected in-house and the choice for "Security" can be handed over to the customer. In this case, the customer receives code in which "Security" is still a variation point but "Power Saving" is already configured.

3 MDA for the Integration of Heterogeneous Components

This section begins with an overview of our approach and then elaborates on each step. The overview of our approach is described with respect to the transformations illustrated in Fig. 3.

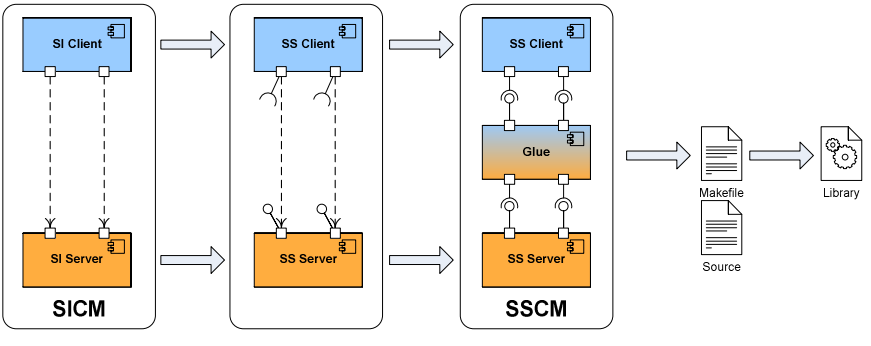

Fig. 3. Model transformations for integration and configuration of components

The presented model-driven approach uses the new variability pattern introduced in [8]. In this approach, an initial reference architectural model is defined containing conceptual components and their composition. These conceptual components represent architectural variation points, each of which describes the alternative suppliers of

that component. Later, this model is transformed into one in which the conceptual components are substituted by models of components from the selected suppliers.

We term the initial model a *supplier-independent component model* (SICM). This model, which is represented in terms of a new UML profile, consists of supplier-independent (SI) components and their dependencies. When the application engineer selects suppliers for the SI components, model-to-model transformations create a new model. In this new model, the SI components have been substituted by supplier-specific (SS) components that contain the interface descriptions for the corresponding development artifacts, which are tagged with their component technology.

Now, glue components are inserted by a second model-to-model transformation wherever a pair of incompatible interfaces is found. Then, a combination of model-to-code transformations and reusable code snippets is used to create the required glue code and other auxiliary files, such as build scripts, which can be transferred to the next participant in the supply chain.

To evaluate this approach, tool support for glue modeling was implemented as an extension to the IBM Rational toolset [15], including a new UML profile for modeling the supplier variability and glue specification, and model-to-code transformations for generating code artifacts. For feature modeling, a commercially available variability management tool was used [16], for which no extensions where needed. The process of using this approach, is illustrated in Fig. 4, and is further elaborated in the following subsections.

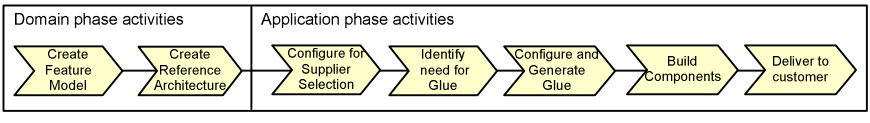

Fig. 4. Process description

3.1 Creation of a Feature Model and a Reference Architecture

As the first step in the process, the domain engineer defines the feature model that represents the product line variability, using a variability management tool. A distinction can be made between variation points that relate to product features, denoted as *functional variation points* (FVP), and variation points that describe alternative suppliers, denoted as *supplier variation points* (SVP), according to [9].

Then the SICM is created using the new UML profile. The SICM contains the SI components with their ports and variation points, as well as the dependencies between these components (left hand side of Fig. 3).

As potential component suppliers are identified during domain engineering, models of the SS components are defined. These components contain the interface descriptions for the supplied development artifacts. Subsequently, an *implement* connection with variability conditions is used to connect the SS components to their corresponding SI component, as shown in Fig. 5 for the ZigBee case study.

To complete the product line domain definition, the domain engineer activates validation rules to check that the SICM is legal and complete (e.g., each SI component must have a corresponding SS component to implement it).

The conceptual model of the entities used in this approach is shown in Fig. 6. It shows how the SS Components implements the SI components and their relation to the glue components. It also shows the relations between the components and the variation points (FVP, SVP). The different variability conditions for a particular SI component represent the SVP and are linked to the feature model in the variability management tool. The feature model is also linked to the FVPs of the SI components. Additionally, since different suppliers can implement the same FVP differently, and with different binding times, the SS components contain the configuration mappings of their FVPs to the FVPs of the SI components that they implement.

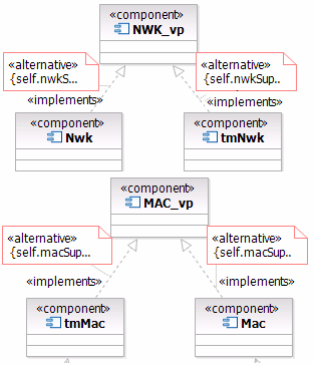

Fig. 5. SVP's of the case study

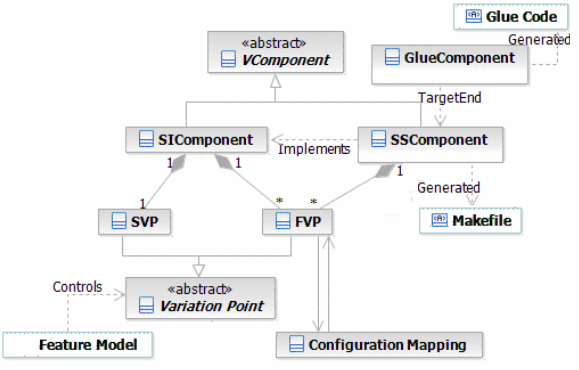

Fig. 6. Conceptual model of entities used

In the Zigbee case study the domain engineer modeled a subset of the ZigBee protocol architecture with two SI components, *NWK_vp* and *MAC_vp*, connected by two ports. Two alternative SS components are modeled for each SI component, *Nwk* and *tmNwk* for the *NWK_vp* SI component, and *tmMac* and *Mac* for the *MAC_vp* SI component (Fig. 5).

3.2 Configuration for Supplier Selection

During application engineering, an early step is the selection of the suppliers for each component, based on the different features that each supplier provides, together with non-functional criteria, such as cost. This choice is made using a variability management tool, which assigns values to the SVPs. A model-to-model transformation creates a new model in which any SI components with assigned SVPs are replaced by the selected SS component, corresponding to the middle model in Fig. 3. This transformation naturally supports staged configuration without any other measures being required to keep the partially configured feature and component models synchronized.

3.3 Identification of the Need for Glue

When resolving supplier variation points, the conceptual components of the initial model are replaced by the selected specific components, possibly from different suppliers. Here we consider the case in which alternative SS components associated with the same SI

component are implemented differently. Yet they should have similar functionality and equivalent ports. Glue components are required wherever connected ports have mismatched interfaces or where they have been implemented in different technologies. These conditions are detected automatically by validation of the UML model, based on the following criteria:

1. For each pair of SS components with connected ports, a *required* interface of an SS component does not match a *provided* interface, or any interface from which it inherits. Here, interface matching relates to the interfaces names and their method signatures (i.e. method names and the number, order and type of their parameters).
2. The components use different component technologies.

The model component elements are checked for the conditions above. Wherever the validation fails, a *glue* component is inserted between mismatched components by a model-to-model transformation, resulting in the right hand model in Fig. 3. At this point, the inserted component provides a specification for the glue, with its implementation still to be generated, as described in Section 3.4. Some component technologies, such as Koala [3] and nesC [11], require a top-level component to specify the composition of the components that use that technology, as illustrated in Fig. 2. Where required, the top-level component is also generated at this point.

When all SVPs have been resolved and all glue components have been added, we obtain the final, *supplier-specific component model* (SSCM).

3.4 Configure and Generate Glue Components

This section starts with a description of the meta-model for glue components. We then proceed with a description of tool support for the application engineer, who can interactively generate the glue code without being exposed to mechanisms of code generation, i.e. the model to text transformations.

Modeling of Glue Components

A glue component should resolve mismatches between interfaces, methods, method parameters and other mismatches between the glued components. Furthermore, it may also supply additional functionality that some component technologies may require. For example, in the case study, the nesC MAC expects that its notification interfaces will be bound statically at build time, whereas the tmCom NWK expects that the server provides a dynamic subscription management facility, which must now be provided by the glue. Other inconsistencies of supplier's implementation that must be resolved within the glue component are initialization, debugging, logging, and power management.

In order to create the glue components most efficiently during application engineering, their implementation is fully-generated from a model. This is in contrast to generating the skeleton of the code and completing it manually. The model combines information from the SSCM with parameterized code snippets created during domain engineering, and is configured interactively during application engineering using a set of wizards, which will be elaborated in the remainder of this section.

A meta-model for glue component models is defined in Fig. 7. Each glue component is associated with two SS components to be glued (the “TargetEnd” association), one of them may also be specified as "WrappedEnd' for cases where a top-level wrapper component is needed. The glue component contains ports according to the connected ports of the replaced SI components. For each such port it holds a number of interface maps. The glue component also has indicators for whether initialization and subscription management are to be included.

The relationships between the *provided* and *required* interfaces of the glue components are addressed at three levels of component integration: interface maps, method maps and parameter maps. Each kind of map has its specific attributes and snippets. This arrangement is able to support component integration even when different suppliers group methods into interfaces in different ways.

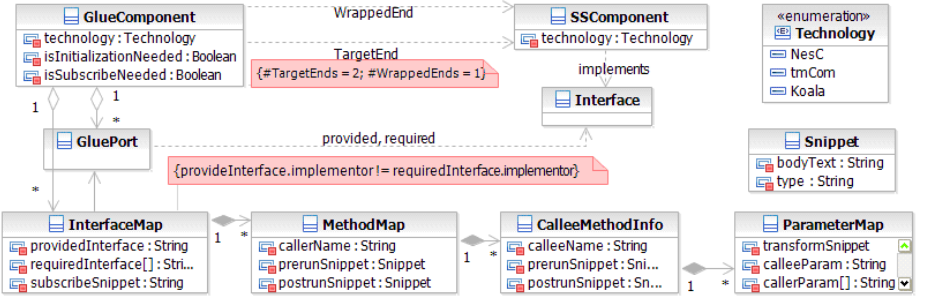

Fig. 7. Meta-model for a Glue Component

These relationships between interfaces are represented in the meta-model as follows. It contains an *InterfaceMap* for each interface provided by the glue. Each interface map contains a set of method maps that have one *caller* entity and a number of *callee* entities. This 1:n relationship caters for cases where the methods of the client and server are not matched, so that a single client call must result in a sequence of calls to the server. Each *callee* entity contains a set of parameter maps, which are used to control the transformations between those parameters that are passed in different forms by the *caller* and *callee* methods. Parameterized code snippet templates, stored in a library, are used for generating snippet instances inside of all these maps; these snippets are essential part of the generating glue code.

The code that is generated from the glue model consists of a set of methods for each *provided* interface of the glue component. The core of the *caller* method's body is a sequence of calls to the methods in the *CalleeMethodsInfo* list, together with the necessary parameter transformations. This sequence is contained within *prerun* and *postrun* code snippets, which can support other functionalities, such as memory management for temporary parameter structures or logging. When the application engineer has populated all the maps, a model-to-code transformation is used to generate the software artifacts, such as glue code, wrapper component, and build scripts.

Tool Support for Glue Code Generation:
To make the glue specification process faster with an effective use of the snippet templates tool support was developed on top of IBM RSA [15]. The implemented tool includes a set of wizards and dialogs to support the application engineer during glue configuration. These wizards hide the mechanics of code generation from the application engineer. For illustration, screen shots of the Interface Map Wizard, applied to the ZigBee case study, are shown in Fig. 8.

The Interface Map Wizard assists the user in specifying the mapping between provided and required interfaces that need gluing. Central part of the wizard is the Interface Map Editor, which allows to select one or more callee methods for each caller method, and to specify the transformations between the parameters of the methods. These transformations are captured as code snippets. The code for a snippet can be entered by the user either explicitly or by selecting a predefined snippet from a snippet library. The snippet library allows snippets to be reused across different interface maps and glue components. The snippet text can also be parameterized by predefined parameters such as interface name, callee or caller method name, etc. Parameter values can automatically be substituted by the model attributes taken from SSCM.

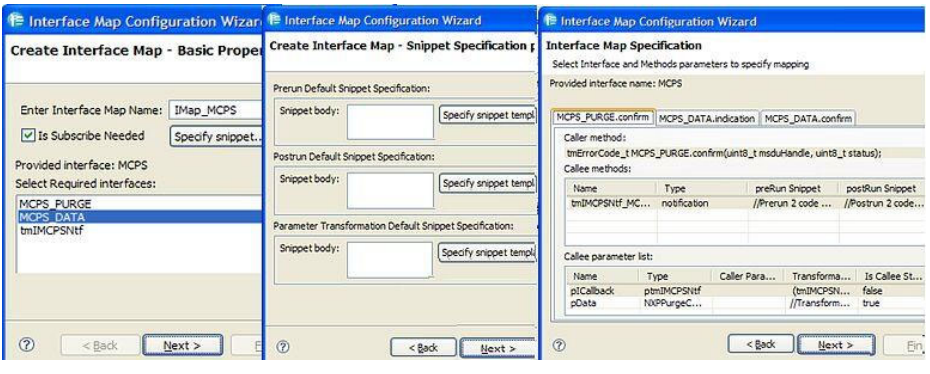

Fig. 8. Screenshots of the wizard for the Interface Map Configuration process

In addition, to reflect technology naming conventions, the tool allows usage of naming hints, which may significantly automate the method and parameter name matching process. We characterize the name's structure as a combination of the part that identifies the specific method or parameter, and additional parts (prefix, suffix and delimiters), e.g. the identifier of the interface, that together comprise the full identifier. In this way the additional parts may be stored for each supplier or technology, and used during the glue configuration process.

For the ZigBee case study we first created examples of glue methods manually, from which a library of parameterized code snippets was extracted. They are used later to configure numerous glue component maps in conformance with the meta-model described in Section 3.4. Code generation was implemented using the extensible JET-based Rational Software Architect transformation framework. A fragment of the code generated by our prototype is shown in Fig. 9.

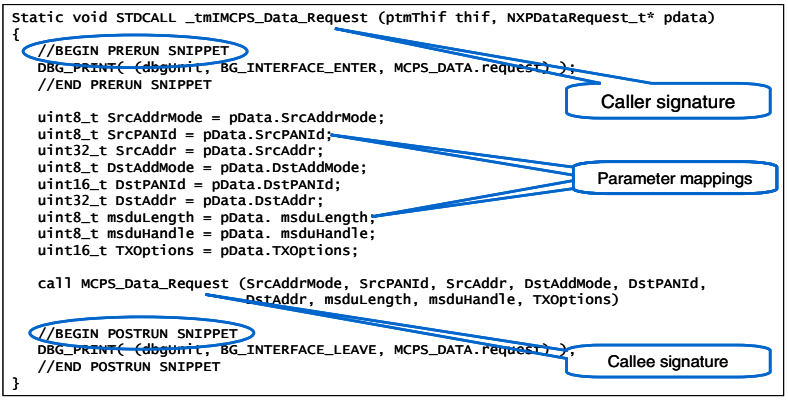

Fig. 9. Example of the generated glue code for the ZigBee case

3.5 Building the Components and Delivery to the Customer

Prior to the final build of the product, all the FVPs must have been configured, but we require flexibility in the configuration time for any FVP. The integrator makes an initial configuration, e.g. to protect intellectual property of other customers and suppliers and each customer receives a specialized configuration space containing only the remaining unconfigured variation points.

At the point that the code is validated and delivered, the mappings from the SI components' unconfigured FVPs to the corresponding variation points in the development artifacts are added to the generated build script. Subsequently, this mapping is used to translate the customer's configuration description. To address these two issues, we adopt a two-stage approach. For each programming language:

1. The components are passed through the early stages of the build process for their respective component technologies, to the point where standard language source and header files are generated. For example, Koala [3] identifies which source files will be required and generates macros to rename functions to permit static binding.
2. Having transformed all components to a standard form of source file for their language, build scripts are generated. These files include the FVP settings, such as the definition of pre-processor symbols used in the realization of variation points. Additional build scripts are generated for each glue component and a further, top-level build script identifies all the required components and validation is performed.

Where the customer only receives binary code, or where it is not possible to separate the two stages above, the final build is performed remotely on the supplier's site (e.g., by exposing the complete configuration and build process as a web service).

4 Development Roles

Given the small proportion of software developers who have experience with MDA technology, to be deployable in the short term, it is essential that only a few developers need to be familiar with the more esoteric aspects of MDA, such as defining UML

profiles and model transformations [2]. This section describes the development roles involved in the approach and the different levels of knowledge that each role requires in performing the activities, as illustrated in Fig. 4 and described in section 3.

The task Create Feature Model is performed by the *requirements manager* and requires a working knowledge of feature modeling. The task Create Reference Architecture is performed by the *domain architect*, who defines the product line architecture, represented by the SICM, and who also identifies the potential suppliers and creates the SS component models. This role requires a working knowledge of MDA.

The tasks Configure for Supplier Selection, Identify need for Glue and Configure and Generate Glue are performed by the *COTS engineer*. The *COTS engineer* is responsible for the integration of components from different suppliers and creating glue components. The *COTS engineer* should be familiar with the component technology and development environments used by the suppliers but he does not need specific knowledge of MDA since his tasks are assisted by the set of wizards that hide underlying MDA complexities. The tasks Build Components and Deliver to Customer are performed by the *customer support engineer*. The *customer support engineer* liaises with customers and determines what configuration is required prior to delivery. He will use the variability management tool to define a specific product configuration and uses the MDA tool to create the SSCM and to perform the final export. These tasks do not require knowledge of MDA principles. The *customer*, being the next link in the supply chain, requires no specific technical skills. He uses the variability management tool to make the final configuration of the received product artifacts, but is not exposed to any MDA technology.

As part of domain engineering team we recognize additional tasks, not described in Fig 4, to support the *domain architect*. The *COTS engineer* provides the requirements to the *transformation developer,* e.g. when a new component technology is used, who defines the transformations to generate the glue code and related artifacts. Here, the *transformation developer* requires very specific skills related to the transformation tooling. The *COTS engineer* has tasks during domain engineering as well as application engineering.

Finally, we identified the *Language Designer* [2] who is responsible for the definition of the meta-model and the model-to-model transformations. This work requires an in-depth knowledge of MDA. Since the meta-model and transformation can be reused for any component composition, these activities would typically be done by the MDA tool vendor.

5 Discussion and Further Research

The approach described in this paper allows components from different suppliers to be integrated, despite syntactic differences in interfaces and semantic differences related to component technologies that are based on a common programming language and variability mechanisms. It also supports staged configuration, where some variation points are resolved by the next participant in the supply chain.

The approach addresses the application engineering phase of SPLE, abstracting from the variability mechanisms used by each supplier and supporting the creation of glue components where they are required. It aims to make that glue generation as efficient as

possible, without making speculative investments during domain engineering. Once the glue snippet templates have been created for a few exemplars they are reused for numerous glue components.

The approach recognizes the limited experience of MDA available in the industry and restricts the number of development roles that need to be familiar with it. This is achieved by using a balance of reusable model-to-code transformations and parameterized code snippets, with a development environment that provides guidance to the applications engineer. Furthermore, our approach supports the export of standard programming languages and makefile technology, thereby avoiding exposure of the customer to unfamiliar technology.

The approach retains the sophisticated feature modeling techniques and supporting variability management tools developed for SPLE. For instance, these support staged configuration through the ability to present the engineer with a specialized configuration space, in which choices made at earlier stages are no longer accessible, although the constraints resulting from these choices are still in effect. However, the links to the development artifacts and, in particular, the mapping to the different variability mechanisms used by different suppliers, is now passed to the component models. The variability management tool is no longer required to determine where glue components will be required; this is now determined by a single model validation rule, providing a scalable solution. Hence, the variability model is now not *directly* connected to development artifacts [1], or model transformations [17], but the choice of the SVP now, *indirectly*, may lead to the generation of a glue component.

One of the challenges for further research is in the ability to deliver to customers partially-configured development artifacts while preserving the full capabilities of component technologies, such as nesC [11] and Koala [3], that minimize the memory requirements of the code through their reachability analyses of their components. The current approach converts all components into standard source files and generates a uniform style of build script, which propagates the remaining FVPs, thereby avoiding the customer from being confronted with multiple build environments. Currently, the creation of standard source files uses the native build process for each component technology. However, the build process for some component models, such as nesC [11], only creates conventional C files once the C pre-processor has been run, by which time all variation points with design-time binding will have been instantiated. However, by developing new build environments that are aware of variation points, both staged configuration and reachability analysis can be supported for these models.

A principal area for further research is the bridging of greater semantic gaps between components. This would allow the approach to be applicable to a much larger range of glue code generation. The current approach supports moderate mismatches because of the ability to map one *caller* function to a sequence of *callee* functions. This is sufficient for the ZigBee case, whose standard defines the semantics of messages *within* the communication stack, but there are many standards for embedded products that only consider the interaction between the product and its environment, with no consideration for the APIs of the software within the product. Egyed and Balzer [18] have proposed a reference architecture for stateful glue components, which may form the basis of more capable glue components for COTS integration. Here further research is required to extend the automated support to be able to guide creation of this style of wrapper. A related issue is that the current approach only inserts glue

components between pairs of components. However, there are cases, such as the generation of code to map from the operating system abstraction layers (OSAL) of each supplier to the actual OS, that cannot be done in a pairwise manner, because of the need for common book-keeping for shared OS resources. Therefore, a more general model of glue code must be developed for these cases.

6 Comparison with Related Art

From the perspective of staged configuration in software supply chains [6], we address organizations in the middle of the chain, which must both integrate components from different sources and pass partially-configured artifacts on to downstream customers. The problem of the use of feature models for coordinating the configuration of artifacts using different variability mechanisms is addressed by Reiser *et al.* [19]. However, they do not consider the creation of glue components. We have previously discussed merging feature models from alternative suppliers for a particular feature area [14], but that paper did not consider how glue components would be addressed.

Gomaa [20] addressed the use of UML to represent feature models. While doing so would have resulted in only a single tool being used, UML must be heavily profiled for this purpose.

Voelter and Groher describe the integration of a variability management tool with a model-based software development environment [17]. However, they address the links to transformations for in-house developed components, rather than the needs of a software supply chain.

The definition of the SICM for the ZigBee case study was straightforward, given the reference model in the standard. Where there is no pre-existing reference mode, the *architectural reconciliation* approach, proposed by Avgeriou *et al.* [21], to defining a COTS-based architecture can be used. However, while their approach aims to avoid architectures that require excessive amounts of glue code, they do not address how the essential glue code would be created efficiently.

Zhao *et al.* [22] address the combinatorial explosion of potential glue components when bridging between different component technologies. They use a generic grammar to specify the implementation of glue, but they avoid having to handle hybrid build processes by using SOAP as a common communication format between all technology types. This is approach is unacceptable for resource-constrained devices, in which code size and performance remain critical. Smeda *et al.* [23] address the creation of the specification of glue components from the composition of parameterized templates in the context of an architectural description language and address the creation of a modeling tool for this language. However, they do not address how automated support could be given to developers to assist in template composition.

Stahl *et al.* [2] and Krahn *et al* [24] describe the different roles needed in MDA and the skills required. Where they provide a classification for the roles during domain engineering, we additionally provide the different roles and skills, associated with our approach, during application engineering and for the customer's organization.

7 Conclusions

In this paper, we presented a model-driven approach for automating the integration of heterogeneous components from different suppliers, covering syntactic mismatches and semantic mismatches related to different component technologies. We exercised our approach on a case study that is derived from an existing supply chain, for which we used a commercially available variability management tool [16], and a prototype was implemented as an extension to IBM Rational MDA tool [15]. We described the process and roles that are associated with our approach.

In this paper we showed that the approach has the following benefits compared to prior art and current practice:

- Glue components are generated efficiently only when they are required, thereby avoiding unnecessary development effort during domain engineering.
- Staged configuration is supported; offering the next party in the chain to do the final configuration, while providing a route to preserving the capabilities of component technologies in this domain to minimize code size.
- The additional skills required to deploy MDA are localized in the organization by providing tool support for configuration and glue code generation, which ensures that only a limited group of developers are exposed to unfamiliar technology.

Finally, we identified how our approach can be extended to support specialized component technologies and to bridge greater semantic differences.

References

1. Pohl, K., Bockle, G., van der Linden, F.: Software Product Line Engineering. Springer, Heidelberg (2005)
2. Stahl, T., Voelter, M.: Model-Driven Software Development. Wiley, Chichester (2005)
3. van Ommering, R.: Building Product Populations with Software Components. PhD. Rijksuniversiteit Groningen (2004)
4. Atkinson, C., et al.: Component Based Product Line Engineering with UML. Addison-Wesley, Reading (2002)
5. Wallnau, K., Hissam, S., Seacord, R.: Building Systems from Commercial Components. Addison-Wesley, Reading (2002)
6. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. Software Process Improvement and Practice 10, 143–169 (2005)
7. Hartmann, H., Trew, T.: Using Feature Diagrams with Context Variability to Model Multiple Product Lines for Software Supply Chains. In: 12th International Software Product Line Conference (2008)
8. Hartmann, H., Keren, M., Matsinger, A., Rubin, J., Trew, T., Yatzkar-Haham, T.: Integrating Heterogenous Components in Software Supply Chains. To be published in 1st ICSE workshop on Product Line Approaches in Software Engineering (2010)
9. Hartmann, H., Trew, T., Matsinger, A.: Supplier Independent Feature Modeling. In: 13th International Software Product Line Conference (2009)
10. ZigBee Alliance, `http://www.zigbee.org/`

11. Gay, D., Levis, P., van Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC Language: A Holistic Approach to Networked Embedded Systems. In: Conference on Programming Language Design and Implementation ACM 2003 (2003)
12. ISO/IEC 23004-3:2007, Information Technology – Multimedia Middleware – Part 3: Component Model. International Organization for Standardization (2007)
13. Cunha, A., Koubaa, A., Severino, R., Alves, M.: An Open-Source Implementation of the IEEE 802.15.4/ZigBee Protocol Stack on TinyOS. Polytechnic Institute of Porto (2007)
14. ISO/IEC 23004-1:2007, Information Technology – Multimedia Middleware – Part 1: Architecture. International Organization for Standardization (2007)
15. IBM Rational Software Architect for WebSphere software, `http://www-01.ibm.com/software/awdtools/swarchitect/websphere/`
16. Pure::Variants, Variability Management Tool, `http://www.pure-systems.com`
17. Voelter, M., Groher, I.: Handling Variability in Model Transformations and Generators. In: 7th OOPSLA Workshop on Domain-Specific Modeling (2007)
18. Egyed, A., Balzer, R.: Integrating COTS Software into Systems through Instrumentation and Reasoning. Automated Software Engineering 13, 41–64 (2006)
19. Reiser, M., Tavakoli Kolagari, R., Weber, M.: Unified Feature Modeling as a Basis for Managing Complex System Families. In: 1st International Workshop on Variability Modeling of Software-intensive Systems (2007)
20. Gomaa, H.: Designing Software Product Lines with UML. Addison-Wesley, Reading (2005)
21. Avergiou, P., Guelfi, N.: Resolving Architectural Mismatches of COTS through Architectural Reconciliation. In: Franch, X., Port, D. (eds.) ICCBSS 2005. LNCS, vol. 3412, pp. 248–257. Springer, Heidelberg (2005)
22. Zhao, W., Bryant, B., Burt, C., Raje, R., Olson, A., Auguston, M.: Automated Glue/Wrapper Code Generation in Integration of Distributed and Heterogeneous Software Components. In: 8th IEEE International Enterprise Distributed Object Computing Conference (2004)
23. Smeda, A., Oussalah, M., ElHouni, A., Fgee, E.-B.: COSABuilder: an Extensible Tool for Architectural Description. In: 3rd International Conference on Information and Communication Technologies (2008)
24. Krahn, H., Rumpe, B., Völkel, S.: Roles in Software Development using Domain Specific Modeling. In: 6th OOPSLA Workshop on Domain-Specific Modeling (2006)