university of groningen

University Medical Center Groningen

University of Groningen

Software product line engineering for consumer electronics

Hartmann, Herman

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Publisher's PDF, also known as Version of record

Publication date:
2015

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Hartmann, H. (2015). *Software product line engineering for consumer electronics: Keeping up with the speed of innovation*. University of Groningen.

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 12-10-2022

Chapter 6. Using MDA for Integration of Heterogeneous Components in Software Supply Chains

This chapter is published as:
H. Hartmann, M. Keren, A. Matsinger, J. Rubin, T. Trew, T.Yatzkar-Haham: "Using MDA for Integration of Heterogeneous Components in Software Supply Chains", Sci. Comput. Program. 78(12): 2313-2330 (2013) DOI 10.1016/j.scico.2012.04.004

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 (partially configured) products to its customers. To satisfy both the variability required by each customer and the variability required to satisfy different customers' needs, it may be necessary for such a company 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 the limitations of the current practice of combining heterogeneous components in a product line and describe the challenges that arise from software supply chains. We introduce a model-driven approach for automating the integration between 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.

6.1 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. The software product line development process is usually separated into two phases: domain and application engineering. During the domain engineering phase, reusable artifacts that contain common and variable parts are created. These artifacts are configured during the application engineering phase to create specific variants. A variability model is used to capture the commonality and variability in a product line and to configure a variant [Pohl 2005A].

In this paper we address products running embedded software, such as those in the automotive, aerospace & defense or consumer electronics domains. As the amount of software in these products grew, an increasing portion of a product line used commercial components [Wallnau 2002]. When such components first began to be incorporated, there was a clear separation of roles, with product integrators selecting components from suppliers. As the complexity of products increased further, a new market for pre-integrated sub-systems emerged. Each sub-system might contain components from several suppliers, resulting in a software supply chain. In a supply chain, participants use components containing variability, combine them with in-house developed components, and deliver components containing variability to the next party in the supply chain [Greenfield 2004, Czarnecki 2005A, Hartmann 2008, Pohl 2005B, Kim 2005].

Many application areas employing embedded software have limited standardization of APIs and software component technology, so different suppliers may choose different mechanisms for interfacing, binding, and implementing variation points in their components [Ommering 2004, Park 2007, Blair 2009]. When a system is formed from components that do not conform to a common architecture, these components are usually referred to as *heterogeneous components* [Yang 1997]. When heterogeneous components have to be integrated, there might 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 implement somewhat different functionality (functional splitting).

Often, for a particular functional area, the variability required to satisfy all customers is too great to be implemented by a component from a single supplier. Therefore, there may be alternative components for each functional area, together with associated glue components [Hartmann 2009]. Where there are alternative components, specific components must be selected, configured and integrated during application engineering.

In this paper we analyze the current practice for integration and configuration of components. We show that the traditional approach, in which the possible glue components are developed during the domain engineering phase, is not a scalable solution. The alternative approach, in which the required glue components are created during application engineering, requires too much throughput time and, finally, creation of a common standard interface would introduce unnecessarily complex glue components. We identify the challenges of developing embedded software that executes on resource-constrained devices, as well as the challenges that arise from developing software in a software supply chain. For resource constrained devices, it is important to minimize the code in the product. Therefore, some component technologies use a reachability analysis on the code to exclude components that are not required in a particular configuration [Ommering 2004]. This need for source code to be available during applications engineering conflicts with the desire of parties earlier in the chain to protect their Intellectual Property (IP). It may also be necessary to protect the IP of customers, e.g. to restrict the visibility of variation points. Furthermore, each specialized component technology may have a customized build environment, hampering building the overall product.

To address the aforementioned challenges, our work introduces a novel model-driven approach for automating the integration of heterogeneous components, covering syntactic mismatches, mismatches related to different component technologies and minor semantic mismatches. Our approach is based on a reference architectural model, from which model-based transformations generate the artifacts required for a specific product variant. The selection of components is driven by a feature-modeling tool and wizards guide the application engineer, who may not be experienced in model-driven technology, through the creation of glue components, for which the code is generated automatically. Using this approach, glue components are generated efficiently and only when they are required. Staged configuration is supported, while we will identify a route by which, given further research, the reachability analysis to minimize code size can be applied across multiple component technologies.

We exercised our approach on a case study that is derived from an existing supply chain and describe the process and roles that are associated with this approach.

The paper addresses the following issues:

1. What are the challenges that arise from combining components with non-matching interfaces and what are the limitations of current practice?
2. What are the challenges that arise from developing a product line in a software supply chain? In particular, what are the challenges for developing resource constrained devices?
3. Can MDA be used to bridge mismatches between components from alternative suppliers?
4. Can support for staged configuration be provided?
5. What is the development process for the proposed approach and what level of MDA expertise is required by the engineers?

The remainder of the paper is organized as follows: Section 6.2 elaborates on the integration and delivery problems, and the current industrial practices. Section 6.3 introduces a case study, while Section 6.4 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 6.5. Section 6.6 contains a discussion and identifies areas for future research. Section 6.7 gives an evaluation of our approach with Section 6.8 making a comparison with related art, followed by our conclusions in Section 6.9.

6.2 Problem Description

6.2.1 Component integration

A product line must be able to satisfy the requirements of its potential customers. The key performance metric for suppliers in a software supply chain is its ability to be responsive to changing product requirements [Xia 2005]. When implementing a functional area using pre-existing components, often no single supplier can cover the full range of variability needed, so it is frequently necessary to use components from several suppliers. This leads to a

product line that contains alternative components, only one of which can be used in a particular implementation [Hartmann 2009], and a large number of glue components.

For example, in the high level component diagram of the car infotainment system shown in Figure 30, the manufacturer uses four alternative suppliers for Navigation (with different displays and for different countries), three alternative suppliers for connectivity (Bluetooth, USB and/or 3G cellular), and three alternative suppliers for audio processing (with different equalizers, and amplifiers). To bridge between the interfaces of these components, up to 33 different glue components (A+B+C combinations) could be needed.

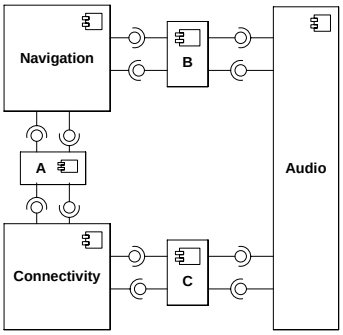

Figure 30 Infotainment high level component diagram.

Figure 31 shows an example of the structure of the glue that is required between a particular pair of components. The left-hand side shows the case where the component interfaces match in all respects and the interfaces can be bound directly. The right-hand side shows the introduction of a glue component, as already discussed, and a wrapper.

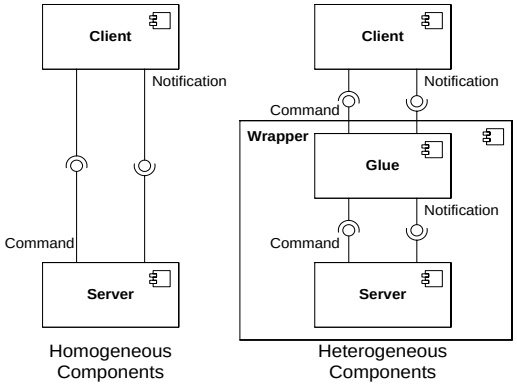

Figure 31 Homogeneous and heterogeneous component integration.

The wrapper is required when the glue component uses a component technology, such as Koala [Ommering 2004] or nesC [Gay 2003] that binds components statically at build time, and requires a top-level component to specify the component composition. Each such

component technology requires a separate wrapper, containing all the components conforming to that component technology. Components of a particular component technology could be scattered over the system.

6.2.2 Current practice for integration and configuration of components

In current practice, a variability management tool is often used for configuring a product line, and creating a particular application out of the base components and glue components that were developed during domain engineering. The variability management tool allows the developer to select the required features, resulting in corresponding settings being made to the variability mechanisms of the development artifacts. Examples of development artifacts are source code and build scripts. Variability management tools include a feature model, which identifies the combinations of features that can be selected, and links to the development artifacts required to support each feature [Pohl 2005A, Xia 2005]. Additionally, these links can provide the rules for configuring the artifacts' variability settings, (e.g., using a dedicated pre-processor). Figure 32 shows a diagram of a feature model, based on the feature-oriented domain analysis (FODA) notation [Kang 1990], with its links to example development artifacts, and the corresponding configuration space of a commercial variability management tool [Purevariants 2009].

The variability management tool can guide an engineer in the selection of the features to be supported by the application, by indicating which selections are no longer available according to constraints defined in the feature model to depict constraints arising from, for example, commercial policies or technical considerations [Hartmann 2008]. In the case of a software supply chain, the feature model can also be used to select the desired suppliers, as described in [Hartmann 2009].

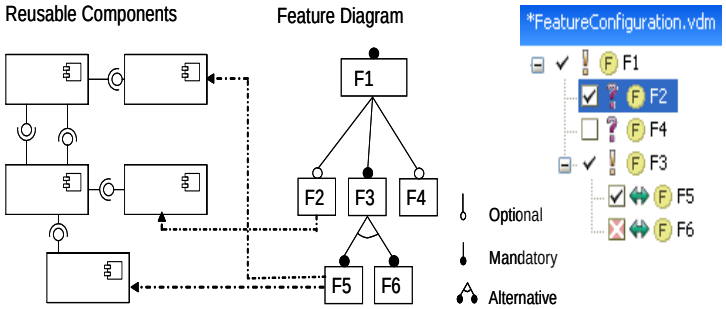

Figure 32 Example feature diagram with links to development artifacts, together with its configuration space.

When considering the integration of glue components, we will describe three of the different approaches that are taken in current practice. In the remainder of this subsection we will describe the characteristics and the limitations of each of these approaches.

Approach 1: Creation of all possible glue components during domain engineering.
In this approach all the possible glue components are created during domain engineering and configured during application engineering. The limitation of this approach is that these artifacts must exist prior to configuration. However, if glue components are only to be used between mismatched interfaces, the need for glue and wrapper components is only known when components from the different suppliers are being selected during application engineering.

Furthermore, the links between the features and the glue components have to be created [Pohl 2005A, Xia 2005]. The presence of each glue component is based on the mismatched interfaces and is therefore based on a unique combination of the selected suppliers for a particular configuration.

The manual creation of glue components and their links to the feature model introduces a substantial development effort into the product line development process, since the number of possible glue components can become considerable, as shown in the example from Section 6.2.1. For a scalable solution it is economically unfeasible to create all glue components and their links speculatively.

Approach 2: Creation of the required glue component during application engineering.
In this approach the variability management tool selects and configures the required base components during application engineering, but the required glue component must be added manually during application engineering. The advantage of this approach is that only the required glue components have to be developed. The disadvantage is that the glue is defined late in the development process and the throughput time during application engineering could become too large. Since the "purpose of application engineering is to explore the space of requirements, and to generate the application very quickly" [Weiss 1999], this approach decreases the benefits of product line engineering.

Approach 3: Use of a common standard interface and component technology
As an alternative to developing glue components wherever mismatched components are to be bound, standard interfaces and component technologies may be defined between each of the components. With this approach, any component not conforming to these interfaces is wrapped to achieve a match. This reduces the number of potential glue components and, since each of them is only associated with a single supplied component, they can be created during domain engineering. In the infotainment example of Figure 30, each glue component would then need two glue layers, but it reduces the total number of possible glue elements to 20. The use of standard interfaces has the important additional benefit that the selection of a glue element is only linked to the selection of the supplier of a single component. Consequently the links between the feature model and the artifacts can be maintained easily and the selection and configuration during application engineering can be done without needing to consider glue components. However, the glue elements that bridge to a standardized interface can become unnecessarily complex when there is a significant mismatch between the standard interfaces and those of the supplied components. This leads to undesirable additional development effort compared to the effort required to glue the two chosen components directly. In addition, different individuals often work on the two halves

of the glue, which can lead to more faults during integration than if a single individual had studied the interfaces of both supplied components.

6.2.3 Delivery of partially configured components

One of the outstanding challenges in a software supply chain is the ability to deliver partially-configured development artifacts to customers. A customer may not be the last participant in the supply chain, so they may leave configuration choices to their own customers, i.e. the next participants in the supply chain. The challenges related to this process have both commercial and technical aspects:

- Some variation points may be implemented in the source code using pre-processor directives. This is a popular mechanism in embedded software because of its ability to minimize deployed software size and avoid performance overheads [Atkinson 2003]. However, to fully protect intellectual property, it may be that a customer should not receive any source code at all [Kim 2005, Sturgeon 2003].
- From a commercial perspective it may be the case that one customer should not be aware of the existence of a variation point that is delivered to another customer, never mind the code that implements that feature. For instance when a particular feature is developed for a customer for whom this feature is a unique selling point, he may not wish to expose it prior to market introduction. There are also situations in which a particular choice of technology may not be revealed to competitors, thereby retaining market advantage.
- Where source code is to be provided, there is a wide variety of build environments, particularly when specialized component technologies are being used. Even when no explicit component technology is employed, there are a variety of approaches used to propagate configuration settings to the variation points of the artifacts. This diversity makes it difficult to create an integrated build environment for a product line containing components from several suppliers.
- Another challenge is to preserve the capabilities of existing technologies to exclude unneeded code. Some component technologies, such as Koala and nesC, target resource-constrained products where hardware costs are critical. These technologies use static binding and employ a reachability analysis prior to compilation, to exclude any code that will not be reachable in that configuration. As will be discussed in Section 6.6, this analysis may be frustrated if multiple component technologies are used in a product.

6.3 ZigBee Case Study

We will demonstrate the applicability of our approach to solving the issues introduced in Section 6.2 using a realistic case study. In order to limit 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 study, we chose a heterogeneous ZigBee stack. ZigBee is a wireless technology for low-cost, low-power wireless sensor and control networks, supporting applications such as home automation, health care and smart metering [ZigBee 2009]. It is designed for applications whose power consumption and hardware cost are lower than, for instance, those supported by Bluetooth. Consequently, many of the standard's features are optional, so that code size can be reduced to the minimum required for any particular application. The ZigBee stack is defined as a layered protocol (see Figure 33).

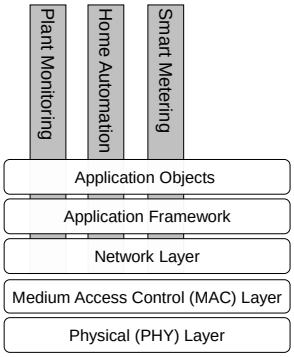

Figure 33 Layers in the ZigBee protocol with pofiles

The standard is platform-independent, being expressed in terms of service primitives, i.e. abstract, implementation-independent representations of interactions between the service-user and the service-provider [ANSI 2003]. In each layer of the ZigBee protocol stack, service primitives are grouped into Service Access Points (SAPs), which expose the layer's services to its clients. This form of specification gives implementers the freedom to make their own choices for the exact form of the APIs between the layers.

In this case study we focus on the lower layers, i.e. Physical, MAC, and Network layers. There is considerable variation between the Network layers for different application profiles, e.g. "Plant Monitoring", "Home Automation" and "Smart Metering", so a particular software supplier 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 appropriate Network layer has to be integrated with the MAC layer for the selected IC.

In this case study, we investigated the source code of three ZigBee stacks. Table 12 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. For instance, the power saving feature together with the beaconing feature allows a battery powered nodes to have a lifetime of 10 years. This is required in advanced e-metering infrastructure [Kistler 2008A] and, consequently, Supplier C is the only suitable supplier for this. Supplier B offers more capabilities in Security and Network configuration, which makes it more suitable for commercial buildings with strict authentication and authorization rules [Kistler 2008B].

Table 12 Feature set of selected suppliers

Feature\ Suppliers	**A**	**B**	**C**
Beaconing	Optional	Not implemented	Optional
Guaranteed Time Slot	Optional	Not implemented	Optional
Security	Not implemented	Optional	Optional
Mesh Configuration	Not implemented	Mandatory	Not implemented
Power Saving	Not implemented	Not implemented	Optional
Message fragmentation	Not implemented	Not implemented	Optional
Implementation Technology	nesC	C	C

Considering the structure of the supply chain, it consists of the IC vendor, alternative Network and MAC layer software suppliers and the final product developer. Because of the constraints on performance and power consumption, the IC vendor is usually responsible for the integration of the ZigBee Network and MAC layers, prior to delivery to the final product developer.

During development, staged configuration may be required for each of the customers. 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.

In the case study, an Open-ZB implementation [Cunha 2007], based on nesC technology [Gay 2003], was used for the MAC layer. To identify the range of capabilities that might be required of glue components, the differences between the technology choices of the Open-ZB implementation and those of the other suppliers were studied. In order to demonstrate how the code for these glue components could be generated, a dummy ZigBee Network layer was created, whose interfaces exhibited the union of the differences that had been found in the investigated software stacks, thereby covering all the observed differences. Here, we chose tmCom as the component technology, this being a precursor of that used in the MPEG Multimedia Middleware standard [ISO 2007B]. The main differences between the technologies used in this dummy Network component and the Open-ZB MAC are summarized in Table 13.

Table 13 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	tmI<port>NXP<interface>[Ntf] <method>	<port>_<interface> .<method>
Calling convention	Referencing structure	Individual parameters
Specific keywords	STDCALL	command, event

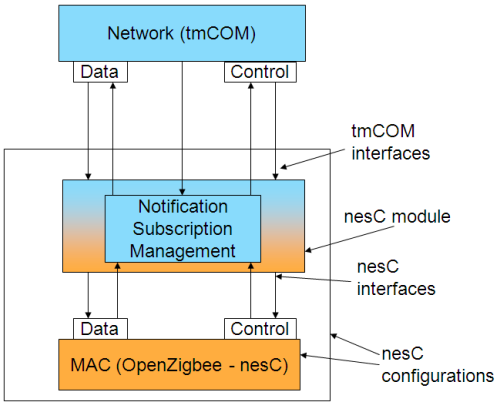

Figure 34 Integration of Network and MAC layers

Both components have two Service Access Points (for data and control), which are connected correspondingly. To bridge the technology differences, an additional glue component must be added, as shown in Figure 34. 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 [ISO 2007A]. 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.

6.4 MDA for the Integration of Heterogeneous Components

The product line architecture contains components from alternative suppliers, where their commonalities are at a higher level of abstraction than the code. We therefore use Model Driven Architecture (MDA) to represent these commonalities since this technology supports abstract modeling and the automatic generation of concrete models and code for specific configurations [Stahl 2005, OMG 2011]. In this section, we first present a conceptual description of our MDA-based approach and the modeling constructs that we introduce. We then describe a high level development process that is to be followed according to our approach, and further elaborate on each step of the process.

6.4.1 Variability pattern and profile

Variability patterns [Gurp 2001] refer to the mechanism of introducing variability into a system, and the process of generating specific system out of many variations. In our model-driven approach, we propose a new product line variability pattern, which extends the common variability patterns such as optional and alternative components, by supporting selection of supplier components and integration of heterogeneous components. Our pattern allows the replacement of a conceptual architecture component, which represents an architectural variation point and has no actual implementation, by one of several alternative supplier components. The selected supplier component substitutes the conceptual component while keeping all its external connections. Moreover, interfaces of the supplier component do not have to exactly match the interfaces of the corresponding conceptual component. This allows us to partially define the interfaces of the conceptual architecture component, according to high level specification or standard, and to refine the interfaces of the concrete supplier components, including method names and signatures. As part of the pattern, automatic detection of mismatched components, and their subsequent integration using the automatic insertion of glue components, is supported.

The reference architectural model contains conceptual components and their composition, which are defined in terms of the dependencies between components' untyped ports. The concrete components, some of which are supplier components and others are in-house components, are modeled with their internal structure and their own variation points. The ports of the concrete components must correspond to those of the conceptual components, but are typed by their actual interfaces. Consequently, the component dependencies can be maintained through their ports. The concrete components are associated with the conceptual components by using variability conditions. An overview of our approach follows, described with respect to the transformations illustrated in Figure 35.

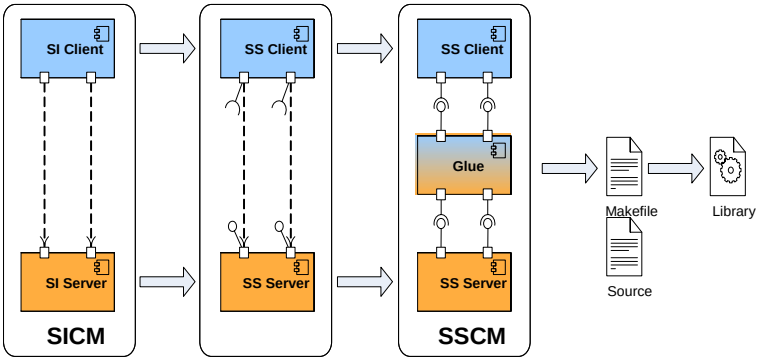

Figure 35 Model transformations for integration and configuration of components

Inspired by the OMG PIM/PSM terminology [OMG 2011], 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 suppliers for the SI components are selected, a model-to-model transformation creates a new model, in which SI components have been substituted by supplier-specific (SS) components that contain interface descriptions for the corresponding development artifacts. Components of the new model are tagged with their component technology.

Now, glue components are inserted by a second model-to-model transformation. This is done wherever a pair of components with connected ports is found that have either different technology tags or incompatible interfaces for these ports. Then, a combination of model-to-code transformations and reusable code snippets is used to generate the required glue code and other auxiliary files, such as build scripts. Generated code can be transferred to the next participant in the supply chain.

The pattern enables the specification of the needed integration at the model level, which allows postponing some variability decisions to a later stage (done in-house or by a downstream customer), and postponing the actual generation of the glue code to the stage when all desired configuration decisions have been made. This also implies that at the feature model level, the downstream customer is only aware of the variability that is available to him and is not aware of any prior variability decisions, such as supplier selection.

6.4.2 Process overview

The process of using this approach is illustrated in Figure 36 and further elaborated in the following subsections. During the domain engineering phase the Feature Model and Reference Architecture are created. The required transformations are implemented as well. During the application engineering phase the suppliers are selected, and then the glue is generated. At the end, the component build is performed, followed by delivery to the next party in the chain (e.g., the customer), who can perform the final configuration.

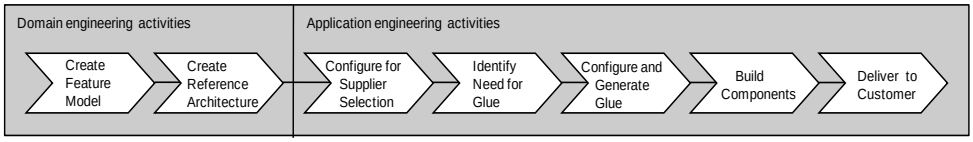

Figure 36 Process overview

To evaluate the approach, tool support for glue specification was implemented as an extension to the IBM Rational toolset [IBM 2011], 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 [Purevariants 2009], for which no extensions where needed. Component modeling was done in the IBM Rational modeling tool.

In what follows, we elaborate on the activities presented in Figure 36 and illustrate their use on the ZigBee case study.

6.4.3 Creation of a feature model and a reference architecture

As the first step in the process, as part of the domain engineering activities, the feature model that represents the product line variability is defined using a variability management tool. Following [Hartmann 2009], a distinction is 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). In the ZigBee case, there are two supplier variation points - one for the MAC layer and one for the Network layer, and a list of functional variation points, a subset of which is shown in Table 1.

Next, 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 Figure 35).

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 *implements* connection with variability conditions is used to connect each SS component to its corresponding SI component, as shown in Figure 37 for the ZigBee case study.

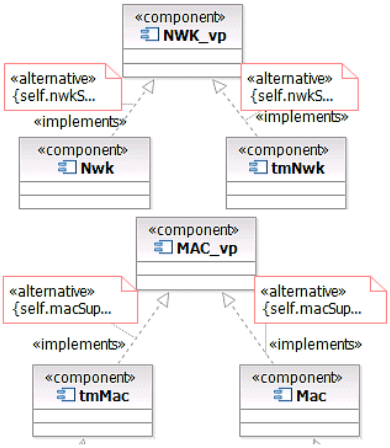

Figure 37 SVP's of the case study

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).

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 (Figure 37).

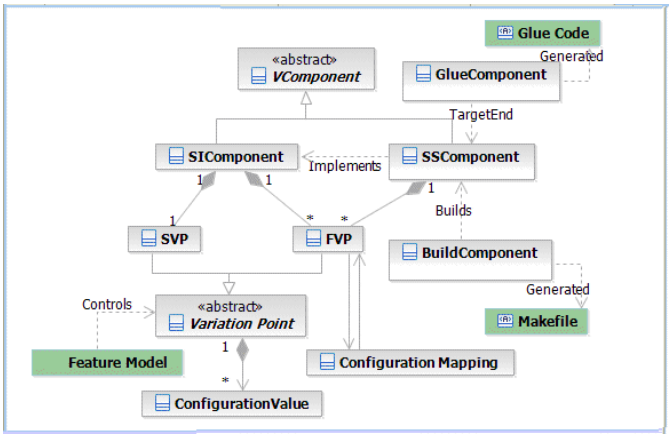

Figure 38 Conceptual model of entities used

The conceptual model of the entities used in this approach is shown in Figure 38. It shows how the SS Components implement 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.

6.4.4 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, as well as non-functional criteria, such as cost. The selection is made using a variability management tool, which assigns values to the supplier variation points (SVP). The assigned values determine, for every supplier independent (SI) component, which supplier specific (SS) component is chosen to implement it. Then, a model-to-model transformation creates a new model in which every SI component with an assigned SVP is replaced by the selected SS component. The process is depicted in Figure 35, where the model with SI components on the left is transformed to the model in the middle, in which SI components are replaced with SS components. These SS components are connected to the rest of the model in the same way that the SI components were previously connected.

The transformation was implemented using the transformation mechanism of IBM's Rational Software Architect (RSA) tool. This allows the creation of a set of mapping declarations that defines relationships between the elements in the models. These mapping declarations include complex expressions to identify elements in the input model and how they are transformed to the output model. The mapping rules are specified by referring to elements of a UML metamodel, extended through the UML profile described in Section 6.4.6. The variability constraints included in the model are evaluated during the transformation, in order to find the attached SS components that satisfy the user's selected suppliers. This model-to-model transformation is generic and can be applied to any supply chain application that has been defined in the tool. The transformation supports staged configuration without any other measures being required to keep the partially configured feature and component models synchronized, allowing suppliers to be selected in several steps, and allowing future configuration of FVPs. In addition, the input model is validated during the transformation against the UML profile.

6.4.5 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, the skeleton of a glue component is created with ports and interfaces that match those of both the SS components. This glue component is inserted between the mismatched components by a second model-to-model transformation. This transformation is depicted in the right hand side of Figure 35, where the above criteria detected that a glue element needed to be added between the two SS components, as illustrated in the middle part of the figure. This is also a generic transformation that can be applied to any supply chain application.

The inserted component provides a skeleton for the glue, with its implementation still to be generated, as will be described in Section 6.4.6. Some component technologies, such as Koala [Ommering 2004] and nesC [Gay 2003], require a wrapper to specify the composition of the components that use that technology, as illustrated in Figure 31. Where required, this top-level component is also generated by the transformation.

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

6.4.6 Configure and generate glue components

To discuss how glue components are configured and generated, we first describe the glue components' meta-model. 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 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 incompatibilities between implementations from different suppliers 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 model is defined in a way that allows the implementation to be fully-generated from it. 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 fragments, called "snippets", created during domain engineering and the model is configured interactively during application engineering using a set of wizards, which we elaborate later in this section.

A meta-model for glue component models is defined in Figure 39. Each glue component is associated with the two SS components to be glued (the "TargetEnd" association); one of them may also be specified as "Wrapped End" for cases where a top-level wrapper component is needed. The glue component contains ports that correspond to the connected

ports of the replaced SI components. For each such port it holds a number of interface maps. In addition to these interface maps, the glue component has indicators for whether subscription management and initialization are to be included. When set, these indicators result in the instantiation of the corresponding snippets to generate the additional code.

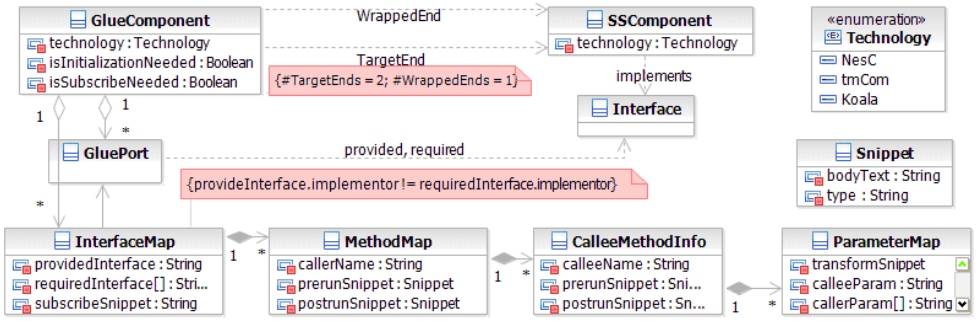

Figure 39 Meta-model for a glue component

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 references to snippets. This arrangement is able to support component integration even when different suppliers group methods into interfaces in different ways.

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.

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, defined by the corresponding snippet. This sequence is surrounded by 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, the wrapper component, and build scripts.

Tool support for glue code generation:

To make the glue specification process faster, tool support for configuring glue, as well as for defining and using glue snippet templates was developed on top of IBM Rational Software Architect (RSA) [IBM 2011]. 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, screenshots of the Interface Map Wizard, applied to the ZigBee case study, are shown in Figure 40.

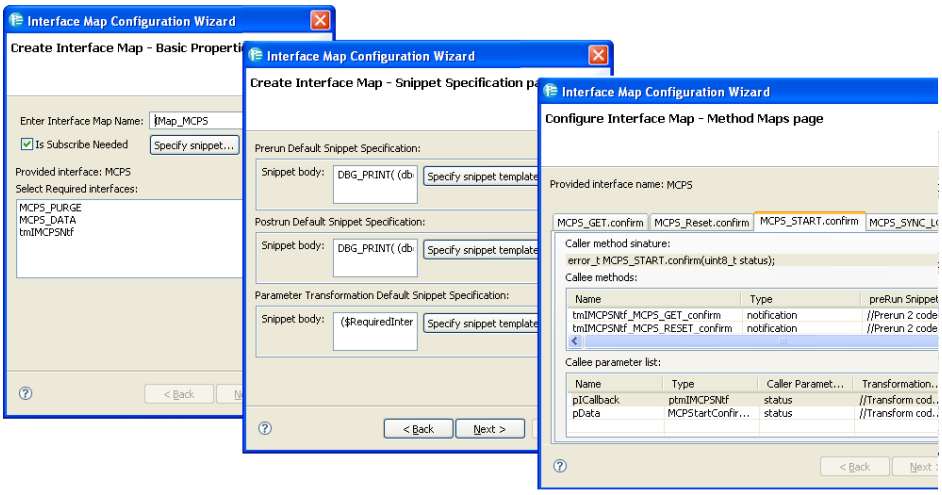

Figure 40 Wizards for the interface map configuration process

The Interface Map Wizard assists the user in specifying the mapping between provided and required inter-faces that need gluing. The central part of the wizard is the Interface Map Editor, which allows the selection of one or more callee methods for each caller method, and to specify the mapping between the parameters of the methods.

The final model-to-text transformation creates glue code that converts between interface methods with similar meanings but of different naming conventions and technologies. Code generation requires the instantiation of parameterized code fragments, in which the parameters are replaced by elements from the model. Conventionally, this is implemented using transformation templates, such as JET used by RSA [Java 2011]. These template languages support constructs for model navigation and conditional code generation. Because hard-coding fragments of glue code in the transformation template would require widespread knowledge of the template language in the development organization, the fragments are stored as snippets. The snippets use a much simpler syntax, only supporting parameter substitution, and are straightforward for developers to create. For example, the debug information that is generated at the beginning of each function is specified by a snippet, rather than being hard-coded in the template. So, we can change the debug information, without changing the template.

The code for a snippet can be entered by the user either explicitly or by selecting a predefined snippet from a snippet library. Figure 41 shows how the snippet library is populated. 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 are referenced by the model-to-text transformation templates, and are automatically substituted by the model attributes

taken from the SSCM. The use of such parameterized code fragments increases range of circumstances in which the templates can be reused unchanged.

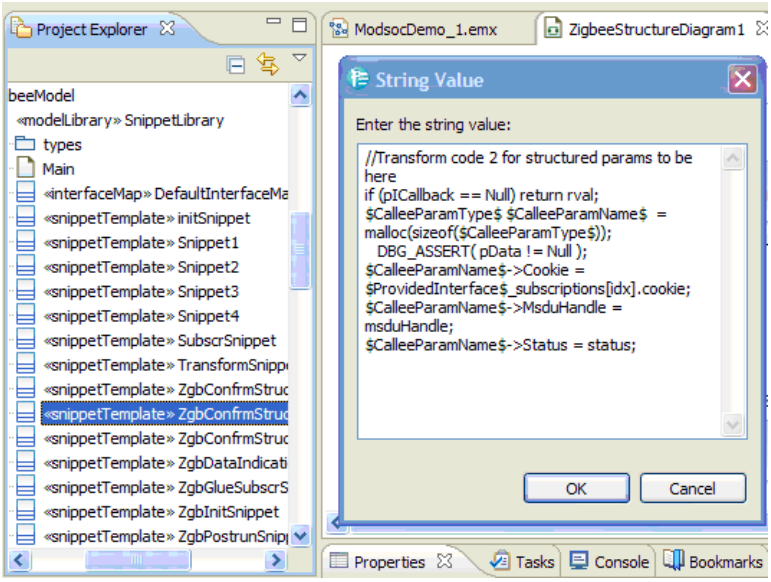

Figure 41 Snippet library

In addition, to reflect technology naming conventions, the tool allows usage of naming hints, which significantly automates 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 6.4.6. 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 Figure 42.

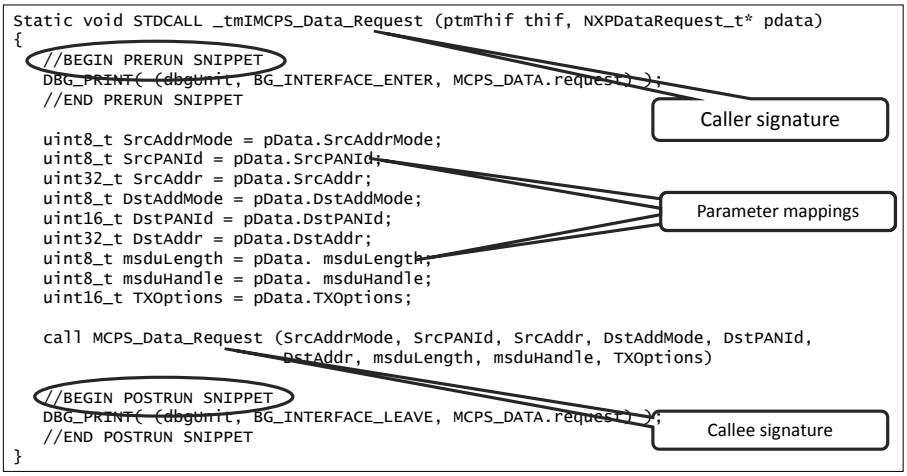

Figure 42 Example of the generated glue code for the ZigBee case

6.4.7 Building the components and delivery to the customer

Prior to the final build of the product, all the SVPs 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 the 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 [Ommering 2004] 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 to protect the suppliers' IP, or where it is not possible to separate the two stages above, the final build is performed remotely on the supplier's site. For instance, by exposing the complete configuration and build process as a web service, the build is performed based on the customer's feature selection.

6.5 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 [Stahl 2005]. This section describes the development roles involved in the approach and the different levels of knowledge that each role requires to perform the activities, as illustrated in Fig. 4 and described in Section 6.4.

Tasks during Domain Engineering:

- 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.

Tasks during Application Engineering:

- 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 final configuration is performed by the *customer*, being the next link in the supply chain. He uses the variability management tool to make the final configuration of the received product artifacts, but is not exposed to any MDA technology.

We recognize additional tasks for our approach. These tasks correlate to the "meta-team", identified by Aagedal and Solhein [Aagedal 2004]:

- The definition of the meta-model and the model-to-model transformations are performed by the *language designer* [Stahl 2005]. 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*.

- The creation of transformations for generating the glue code is performed by the *transformation developer*. The *transformation developer* requires very specific skills related to the transformation tooling. However, this only needs to be undertaken once because these transformations can be reused for any combination of component technologies.
- The *COTS engineer* is responsible for the maintenance of the snippet templates and creation of new snippets, e.g. when a new component technology is used. Should a new combination of component technologies ever be required for which the glue component cannot be generated from the current meta-model, the *language designer* must extend the meta-model and the *transformation developer* must adapt the model-to-text transformations to meet these new requirements. It is the task of the *COTS engineer* to identify these situations and discuss the required changes with the *language designer* and the *transformation developer*.

6.6 Discussion and Further Research

6.6.1 Challenges solved by our approach

The approach described in this paper solves the challenges that are described in Section 6.2. It 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. It also supports staged configuration, where some variation points are resolved by the next participant in the supply chain and the suppliers' IP can be protected through a remote build process.

The approach addresses the application engineering phase of SPLE by supporting the creation of glue components where they are required. It makes glue code 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 next party in the supply chain with a specialized configuration space, in which choices made at earlier stages are no longer visible, although the constraints resulting from these choices are still in effect. Controlling the visibility of variation points protects the IP of other customers.

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 [Pohl 2005A], or model transformations [Voelter 2007], but the choice of the SVP now, *indirectly*, may lead to the generation of a glue component.

6.6.2 Outstanding challenges

While our approach solves the challenge identified in Section 6.2, some other challenges remain for further research, e.g. concerning reachability analysis and bridging of greater semantic differences.

Reachability analysis. One of these challenges is in the ability to deliver to customers partially-configured development artifacts while preserving the full capabilities of component technologies, such as nesC [Gay 2003] and Koala [Ommering 2004], which 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 shielding the customer from multiple build environments. The creation of standard source files uses the native build process for each component technology and their reachability analysis is utilized. However, the build process for some component models, such as nesC, 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.

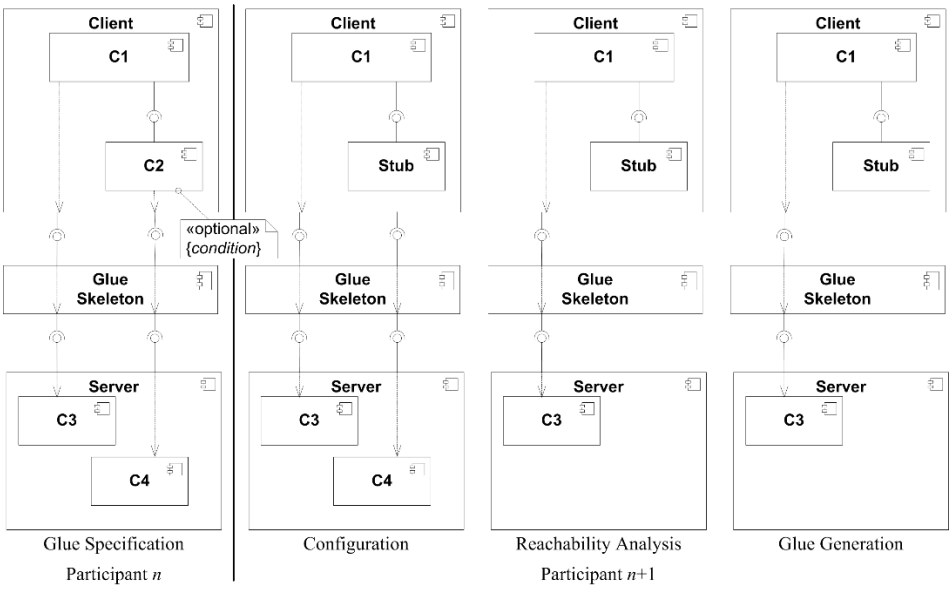

Figure 43 Component composition and configuration with reachability analysis

The current approach does not support the reachability analysis *across* different component technologies whose consequences are illustrated in the following scenario. Figure 43 gives an example of a client and a server with a glue component. The client contains an optional component C2, which requires the server to contain the component C4. If the client and server were implemented in the same technology then, omitting C2 would automatically exclude C4 from the build since it would have no clients. The complicating factor in a software supply chains is that the configuration is done in steps, and the presence of component C2 might only be resolved though a configuration choice of a participant later in the supply chain.

Let's assume that only the server is implemented with the Koala component technology, which uses reachability analysis. The server is to be bound to a client, implemented in some other technology. The diagram begins at the point at which one participant in the supply chain has selected their suppliers and identified the need for glue. One might imagine that the glue could be fully instantiated at this point and delivered to the next participant. Note that only glue components are implemented in two technologies; all other components are implemented in a single technology. In this example, Koala is only being used for the glue component's required interfaces. The Koala reachability analysis will not see any provided interfaces for the sub-components of the glue, since these are implemented in the component technology of the client and the analysis will therefore assume that the glue is the top-level component, and that all its required interfaces are necessary. However, looking downstream, we see that, if C2 is deselected then C4 is no longer required, and should be omitted by the reachability analysis. In other words, if the glue component were to be instantiated as soon as its initial specification is available, then it would appear that all of the server components are always necessary, and the reachability analysis would have no effect. Note that, in a more generic situation, the glue would support both commands and notifications. This would result in a cyclic pattern of dependencies, making the reachability analysis even more complex than is shown here.

The optimal solution would result in a situation in which the reachability analysis is only performed once the final specification is known, allowing all unnecessary code to be removed. Since, at this stage, we have components based on different technologies, together with generated glue components; no existing build environment supports the full analysis. Therefore, technology-neutral interface dependencies should be maintained and new build environments should be developed that are aware of variation points, so that a reachability analysis can be performed across all components. Further research is needed to achieve this optimal solution.

Bridging of greater semantic differences. 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 mismatched components. 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 [Egyed 2006] 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 bookkeeping for shared OS resources. Therefore, a more general model of glue code must be developed for these cases.

6.7 Evaluation

This section discusses our approach using three evaluation criteria: *Quality Improvement*; *Reduction in Development Effort* and *Required Level of Expertise*. These criteria can help practitioners and industrial organizations in estimating the benefits that our approach would provide in their context. Below, we compare our MDA-based approach with the manual creation of glue code.

6.7.1 Quality improvement

In the manual approach, exemplar code is usually developed first. Then, the rest of the glue components are created through copying, pasting and editing this code for each function. When following the manual approach, it is essential that the developer makes all the changes necessary for each copied function, otherwise errors can be introduced, which can be time-consuming to localize, given the limited information that is usually available for COTS components [Wallnau 2002].

In contrast, in the MDA approach, code snippets are created from the exemplar glue component. The snippets fall into two categories: those that will always be instantiated, with their parameters replaced by models elements, and those whose selection requires the developer's judgment. Snippets in the first category are instantiated completely automatically, while the wizard guides the developer through the selection of those in the second category. In addition, in the manual approach the developer has to alternate between selecting the appropriate code fragment to copy and deciding how this fragment should be modified. On the other hand, in the MDA approach, the developer only has to consider those differences between the components that cannot readily be characterized. The wizard prompts the developer for each decision that has to be made. Consequently, the quality of the generated glue components is higher than for those created manually.

6.7.2 Reduction in development effort

The efficiency gain depends on the number of functions for which glue code is required and on the number of decisions that the applications engineer must make when running the code generation wizard (see Section 6.4.6). The use of code snippets allows a fixed set of transformation templates to be reused unchanged because the use of snippets avoids hard-

coding any code fragments within the model-to-text transformations. These snippets can be edited by the COTS engineer without much effort.

The efficiency increases when few decisions have to be made for each function, so that most of the code is generated automatically. Several different scenarios can be considered:

1. In the situation where the mismatches are solely based on the use of different component technologies, the glue components can be generated automatically without human intervention. However this was not the case for the ZigBee case study.
2. For syntactic mismatches, the transformations between the calling and called functions can be captured in the code snippets. This was the case for the ZigBee case study. The applications engineer only has to select the relevant snippets for each function.
3. For simple semantic mismatches, where there is not a 1:1 correspondence between the calling and called functions, the wizard maps a single function call to a sequence of calls. Here, more extensive testing is required to confirm that this mapping is correct, e.g. to verify whether the functions have been invoked in the correct order.

We expect that an additional reduction of development effort will be achieved during maintenance activities, as earlier identified by one of the co-authors [Dubinsky 2011] and by other studies [Mohaghegi 2008]. This is because the structure of the glue code will be more consistent, despite being created and modified by different application engineers, due to the uniformity imposed by the code generator and the reduced number of errors that are expected.

The efficiency gain is further increased if the initial investment can be amortized over many glue code functions.

6.7.3 Required level of expertise

To introduce our approach, a certain amount of training is required to perform the tasks described in Section 6.4. The domain architect needs to have a working knowledge of the UML component model and the use of our UML profile to represent the supplier variation points. The COTS engineer needs to become familiar with the creation and maintenance of snippets, e.g. when new component technologies are used, and the use of the wizards to create the glue components. For both tasks, no specific MDA knowledge is required and the engineer does not have to be familiar with the transformation language itself.

The reference architecture is created when a new product line is started and when additional suppliers are used. This model is easy to develop and maintain, since component dependencies are only represented at the granularity of ports, rather than interfaces. Moreover, the meta-model, the model-to-model and model-to-text transformations, and the wizards, once developed, can be reused between different applications and domains since the meta-model and the transformations cover a very wide range of different component technologies.

6.8 Comparison with Related Art

From the perspective of staged configuration in software supply chains [Czarnecki 2005A], 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. [Reiser 2007]. 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 [ISO 2007A], but that paper did not consider how glue components would be addressed.

Voelter and Groher describe the integration of a variability management tool with a model-based software development environment [Voelter 2007]. However, they address the links to transformations for in-house developed components, rather than the needs of a software supply chain. The Common Variability Language (CVL), described by Fleurey et al. [Fleurey 2009], supports a notation that is compatible with feature models and provides a mechanism for the substitution of model fragments. CVL could support the supplier selection and the first transformation from the SICM, in which supplier independent components are replaced by supplier specific components. However, it is neither able to synthesize the model fragments on-the-fly for the glue components nor to dynamically identify new boundary points where the glue components are to be inserted.

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. [Avgeriou 2005], 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. [Zhao 2004] 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 approach is unacceptable for resource-constrained devices, in which code size and performance remain critical. Smeda et al. [Smeda 2008] 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. [Stahl 2005], Krahn et al. [Krahn 2006] and Aagedal and Solhein [Java 2011] 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.

6.9 Conclusions

In this paper we discussed the implications of developing product lines in a software supply chain. Since software components are developed by different parties in the supply chain, often using different interfaces and different component technologies, an increasing number of glue components are required to bridge those mismatches.

The paper described the problems that arise from combining components with non-matching interfaces. We particularly addressed the case where components from different suppliers are used to implement a single functional area. We showed that the traditional three approaches, that is, the glue components are developed during the domain engineering phase, during the application engineering phase or are mapped to a common standard interface; do not provide a scalable solution.

We introduced a novel model-driven approach for automating the integration of heterogeneous components, 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 [Purevariants 2009], and a prototype was implemented as an extension to an IBM Rational MDA tool [IBM 2011]. We described the process and roles that are associated with our approach.

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; enabling the next party in the chain to do further configuration while IP and commercial interests are protected.
- 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 addressed areas for further research: To support reachability analysis across multiple component technologies to minimize code size, we identified that technology-neutral interface dependencies should be maintained and new build environments should be developed that are aware of variation points. For bridging greater semantic differences we identified that a more general model of glue code is required.

Acknowledgments

The authors would like to thank Uri Avraham, Ofir Brukner, Alan Hartman, Shiri Kremer-Davidson, Itay Maman and Asaf Shaar from IBM Research and Yanja Dajsuren, Jos Hegge and Rob Wieringa from NXP Semiconductors for their contributions to this work.