A Feature-Oriented Approach to Developing Dynamically Reconfigurable Products in Product Line Engineering

Jaejoon Lee  
Fraunhofer Institute for Experimental Software Engineering (IESE), Fraunhofer Platz 1, 67663 Kaiserslautern, Germany  
jaejoon.lee@iese.fraunhofer.de

Kyo C. Kang  
Pohang University of Science and Technology (POSTECH), San 31, Hyoja-dong, Pohang, 790-784, R.O.Korea,  
kck@postech.ac.kr

Abstract

Dynamic product reconfiguration refers to making changes to a deployed product configuration while a system is running. Recently, there have been increasing demands for dynamic product reconfiguration in various application areas (e.g., ubiquitous computing, self-healing systems, etc.); however, most product line engineering methods in the literature have focused on the development of reusable core assets for statically configured products. In this paper, we propose a feature-oriented approach to developing dynamically reconfigurable core assets. This approach takes feature binding analysis results as a key design driver for identifying and managing variation points of dynamically reconfigurable products. We also provide a conceptual model for a reconfigurator, which monitors and manages product reconfiguration at run time. The method is illustrated with a home service robot product line example.

1. Introduction

Product line engineering is a paradigm of software reuse and it aims at developing a family of products with reduced time-to-market and improved quality [1,2,3]. Most approaches in product line engineering, however, have focused on the development of statically configured products using core assets with variation points [4,5]. That is, all variations are instantiated before a product is delivered to customers and, once the decisions are made, they are hard to be altered by the users.

Dynamic product reconfiguration refers to making changes to a deployed product configuration at run time. Dynamic addition, deletion, or modification of product features, or dynamic changes of architectural structures [5,6] are some examples of dynamic reconfiguration. Dynamic product reconfiguration has been studied in various research areas such as self-healing systems [7,8,9], context-aware computing [10,11], software component deployment [12,13,14], and ubiquitous computing [15,16]. When a change in the operational context is detected, it may trigger product reconfiguration to provide context-relevant services or to meet quality requirements (e.g., performance). Dynamic reconfiguration approaches in the literature, however, have focused on reconfiguration of a single product, not on a family of products. That is, accommodation of product-specific dynamic reconfiguration needs that may differ from one product to other has not been considered in engineering software assets.

Different from statically configured products, dynamically reconfigurable products should be able to:
- monitor the current situation (i.e., the operational context) of a product,
- validate a reconfiguration request with consideration of change impacts and available resources,
- determine strategies to handle currently active services during reconfiguration, and
- perform dynamic reconfiguration while maintaining system integrity.

In product line engineering, the capability of dynamic management of product line variations is also required. Suppose, for example, that service features of a home service robot product line should be dynamically reconfigured to provide context relevant services with optimal performance. Moreover, a security monitoring feature, which has high cost implications, should always be available for high-end products, while its availability is determined at run time for low-end products. Then, this feature may be bound with the high-end products at the development time, while it may be bound with the low-end products at run time. Therefore, the challenges of developing core assets that support dynamic reconfiguration lie in the systematic management of dynamic variations of the product execution context as well as context-sensitive product services.

In this paper, we propose a systematic approach to developing dynamically reconfigurable core assets and a reconfigurator that monitors and manages product
configuration at run time. The method first analyzes a product line in terms of features and their binding time. Then, core assets are developed with the analysis results as a key design driver. (See the engineering activities for dynamic reconfiguration in Figure 1.) Next, a reconfigurator is developed considering reconfiguration contexts (when to reconfigure), reconfiguration strategies (how to reconfigure), and reconfiguration actions (what to do to reconfigure). (See the engineering activities of dynamic reconfiguration in Figure 1.)

2. Engineering for dynamic reconfiguration

In this section, activities of product line core asset development for dynamic reconfiguration are introduced.

2.1 Feature and feature binding analyses

Feature analysis is the activity of identifying externally visible characteristics of products in a product line and organizing them into a model called feature model [3]. (Figure 3 shows a part of the feature model for the HSR product line.) Once we have a feature model, it is further refined through feature binding analysis [18]. Feature binding analysis consists of two activities: feature binding unit identification and feature binding time determination. Feature binding unit identification starts with identification of service features. A service feature represents a major functionality of a system and may be added or removed as a unit. In HSR, CC, UF, SM, and TP features (See Table 1 for the description of the abbreviations.) are examples of service features.

A set of features that should be included in a feature binding unit are identified by traversing the feature model along feature relationships. For example, Go Forward (Distance), Caller Localizing, Measure Distance, Rotate (Degree), and Eight Channel Based Audio Source Locating Method belong to the CALL AND COME feature binding unit. (See Figure 3 for feature binding unit identification.) Note that RECOGNIZE FACE is identified as a separate feature binding unit, because the Recognize Face feature is related to both the Caller Localizing feature and the User Localizing feature through "composed-of" relationship. Identification of such "shared" features is important because modification of components (e.g., bug patch, performance improvement, etc.) that implement the shared features may impact all related features. Therefore, by managing shared features as separate binding units, we work.

Table 1. Product features of HSR product line

<table>
<thead>
<tr>
<th>Product Feature</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call and Come (CC)</td>
<td>The robot detects the direction of a sound source by comparing the strength of sound captured by microphones. Then, the robot rotates in the direction of the sound source and tries to recognize a human face by analyzing video data captured through the front camera. If the caller's face is detected, the robot moves forward until it reaches within 1 meter from the caller.</td>
</tr>
<tr>
<td>User Following (UF)</td>
<td>Once UF is triggered, the robot constantly checks the vision data and sensor data from the structured light sensor to locate the user. The robot keeps following the user within 1 meter range. A &quot;Stop&quot; command makes the robot stop.</td>
</tr>
<tr>
<td>Security Monitoring (SM)</td>
<td>The robot performs surveillance patrols around a house using a map. This service is automatically triggered when the robot is left alone in the dark. Intrusion or accidents are defined as patterns recognizable from vision and sound data. Once such an event is detected, the robot captures and saves the image. Optionally, it reports the event to a user by sending a message.</td>
</tr>
<tr>
<td>Tele-Presence (TP)</td>
<td>A remote user can control the robot using a PDA. The user can command the robot to move to a specific position on the map displayed on the PDA.</td>
</tr>
</tbody>
</table>
can localize change impacts.

<table>
<thead>
<tr>
<th>Product Lifecycle View</th>
<th>Security Monitoring, Tele-Presence, User Notify, Capture Photo Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>TELE-PRESENCE, USER NOTIFY, CAPTURE PHOT IMG</td>
</tr>
<tr>
<td>Pre-Operation (installation)</td>
<td>SECURITY MONITORING, USER FOLLOWING, SHAPE-BASED FACE RECOGNITION</td>
</tr>
<tr>
<td>Product Development</td>
<td>SECURITY MONITORING, USER FOLLOWING, SHAPE-BASED FACE RECOGNITION</td>
</tr>
<tr>
<td>Asset Development</td>
<td>CALL AND COME, DETECT AUDIO COMMAND, RECOGNIZE FACE</td>
</tr>
</tbody>
</table>

• CALL AND COME has a higher priority than USER FOLLOWING.
• TELE-PRESENCE has the highest priority among other services.
• SECURITY MONITORING has the lowest priority than other services.

Figure 2. Binding time of HSR binding units

Once features are grouped into feature binding units, their binding times are determined. In our approach, feature binding time is analyzed based on two viewpoints: the product lifecycle view, in which the focus is given to the lifecycle phase in which a feature is incorporated into a product, and the binding state view, in which the focus is given to represent the inclusion, availability, and activation states of features. (Note that a feature may not be available for use even if it is physically included in a product.) For example, inclusion of TP to a product is determined at the pre-operation (installation) time and its availability is determined at the operation time (run time), as shown in Figure 2. Activation rules provide information on concurrency/mutual exclusion of feature binding unit activation and they are defined in terms of mutual exclusion, dependency, and priority schemes. As shown in the third column of Figure 2, the activation rules of HSR define priorities among the feature binding units. For instance, TP has the highest priority and SM has the lowest priority.

Through feature binding analysis, we can identify explicitly what features should be bound into a product at run time. Next section illustrates service and reconfiguration behavior specifications.

2.2 Behavior and functional specifications of feature binding units

Unlike statically configured products, a dynamically reconfigurable product should be able to handle requests to change its current configuration. In this paper, we propose a Statechart template that defines common behavior required to handle dynamic reconfiguration requests. (Figure 4 shows the template.) The top level Statechart specification has three states: **Initialization**, **Termination**, and **In-Service**. The **Initialization** and **Termination** states are for the specification of tasks that are required at the initialization and termination time of a binding unit (e.g., device initialization and shutting-down). The **In-Service** state is further refined into the **Normal**, **Suspending**, and **Reconfiguring** states. The **Normal** state is to specify the service behavior of a
binding unit. The Suspending state includes necessary actions before starting dynamic reconfiguration (e.g., current state saving, active service completion, etc). Reconfiguration actions of a feature binding unit are performed in the Reconfiguring state (e.g., changing service behaviors).

Figure 4. A Statechart template for behavior specificiations of a feature binding unit

This Statechart template is refined for each feature binding unit. For example, the Normal state of SM has two variants: SM with or without an optional binding unit, USER NOTIFY. Depending on the selection of USER NOTIFY, one of the two Statecharts is bound to the Normal state during reconfiguration.

Of the states in the template, the Suspending state defines the strategy on how to handle currently active services. The strategy can be one of: 1) suspending immediately without saving current states, 2) suspending after saving current states for state recovery, 3) suspending after finishing current active services, or 4) no suspending (i.e., seamless provision of services during reconfiguration). For example, UF is specified to keep providing the user following service during product reconfiguration (strategy 4) by replicating its service behavior into Suspending and Reconfiguring states.

Once we specified behaviors of feature binding units, functionalities of each binding unit are analyzed and specified. In our approach, the functionality of a feature binding unit is specified using data flow diagrams (DFD). DFD shows data and control flows between processes, and processes in DFD are controlled by a Statechart specification. For example, the processes in Figure 5 are controlled by the SM Service Manager Statechart.

To present variants in a DFD, a stereotype, <<v>> is used. For example, the SM Service Manager controller in Figure 5 has a Boolean guard (\(\text{USER NOTIFY} = \text{True}\)), which means that all elements of this DFD are selected when the USER NOTIFY binding unit is selected. If it is not selected, the variant process SM Messenger and its data/control flows are removed from this DFD.

Through functional analysis, we also need to identify shared processes and data between feature binding units. For example, the external process Tele-Presence Path Planner in Figure 5 shares the Front Camera Controller process. The information of common functionalities is an important input for the design of product line architecture and components: dynamic removal or addition of a feature binding unit that has shared processes and/or data may affect other feature binding units. For example, a dynamic reconfiguration for the removal of SM should not remove the Front Camera Controller process as it is also used by the processes related to the TP binding unit. Decisions on how to handle these shared processes and data are made in the design phase with consideration of resources usage, concurrency, etc.

Figure 5. DFD specification for SM

In the next section, the design of a product line architecture and components using the feature binding units as a key driver is introduced.

2.3 Product line architecture and component design

2.3.1 Engineering guidelines for dynamically reconfigurable architecture design. In this section, we propose five engineering guidelines for architecture design. The guidelines are to enhance visibility and traceability of dynamic reconfiguration using feature binding analysis results as a key design driver. In the following, proposed architecture design guidelines are explained:

**Guideline one: Separation of control components from data components.** In the product line analysis phase, we specify behaviors and data flows of each
the related architectural components should be examined for further decomposition, refinement, or restructuring.

**Guideline four: Separation of management policies from functional components.** As discussed in Section 2.2, common functionalities between different feature binding units should be managed carefully to maintain system integrity before/after a reconfiguration. Therefore, we propose to introduce Quality of Service (QoS) components in the data plane for common functions so that we can specify explicitly the management policy of them. The policy should include a priority scheme and instantiation/termination conditions of common data/computational components.

**Guideline five: Separation of context from content.** As pointed out in [5] and [6], separation of product service concerns (content) from product reconfiguration concerns (context) is important to reduce the complexity of dynamic reconfiguration. That is, the product service concern should focus on interactions and computations of components to provide services, while the product reconfiguration concern should focus on monitoring current product situations and reconfiguration transactions. Thus, the separation of these two concerns can clarify responsibilities of architectural components.

In the following section, how these guidelines can be used to design a dynamically reconfigurable product line architecture is demonstrated with HSR.

### 2.3.2 Architecture model of HSR product line

For a case study, we adopted the C2 architecture style [19,20] for the development of HSR architecture model as shown in Figure 6. The C2 style provides flexibility through its layered structure and modular components, which are called "bricks." A brick can send/receive messages to/from other bricks through its top and bottom ports, and bus-style connectors connecting ports.

Though the C2 style is flexible for changes, the style itself does not provide design guidelines such as how one can identify bricks and layers, and how functionalities should be allocated to them. In this section, we explain how the style is refined and extended to accommodate run time flexibility. As we apply the proposed engineering guidelines, we divide the C2 style into different planes. First, the architecture model is divided into configuration and product planes, and the product plane is further refined into control and data planes. In the following, each of these planes is explained.

**Control plane of product plane (related guidelines: one, two, and three):** Components in the control plane control behaviors of the system. Each of local behavior control components (e.g., CC, UF, SM, and TP) defines behavior of a feature binding unit and it can be
executed and tested independently from other control components. The global behavior control component (e.g., HSR Mode Manager) defines system modes (e.g., initialization, termination, and power saving modes) and an interaction policy (e.g., priority, concurrency) of local control components.

Data plane of product plane (related guidelines: one and four): The data plane consists of computational bricks, which read input data from sensors and process them to make outputs such as events and temporary data. Event data are sent to HSR Mode Manager to determine global states of a system. Temporary data are sent to other computational bricks as inputs.

Functionallities that are allocated to the bricks in the data plane can be found in the DFD specifications of feature binding units. This means that an explicit mapping between feature binding units and bricks in the data plane can also be established. Therefore, change effects from addition or removal of a feature binding unit can also be traced clearly in the data plane. For instance, dynamic removal of SM should also remove the User Message Manager brick, as the brick is used only by SM.

As we pointed out in the guideline four, we should be careful about common functionalities between feature binding units. For example, the functionality of Front Camera Controller is allocated to the Front Camera brick but this brick is also used by the Tele-Presence control brick, as we had identified in the DFD in Figure 5. For the separation of management policy of such a shared functionality, we added a FC QoS Manager component inside the computational brick: it specifies a priority scheme that determines which one of the control bricks will receive its computation results at a certain point in time.

Configuration plane (related guideline: five): The configuration plane is in charge of detecting contextual changes, determining and validating a reconfiguration strategy, and executing reconfiguration. The plane consists of two types of components: Master Configurator and Local Configurator. Master Configurator collects information from Local Configurators and/or external probes to detect contextual changes. If a contextual change that requires product reconfiguration is detected, Master Configurator processes a relevant reconfiguration transaction to change the current product configuration. Each Local Configurator is connected to a connector, and monitors the product by inspecting messages between bricks.

As we applied the guidelines for architecture design, mappings between feature binding units and architectural components could be established easily and clearly, and interactions of feature binding units became visible and manageable. Also, separation of reconfiguration concerns from product service concerns could alleviate complexity of component behavior specifications, as the role of each component became simple and clear.

Figure 6. Architecture Model for HIS Product Line

In the next section, product line component development is explained.

2.3.3 Product line component development. The primary input to product line component development includes a feature model, feature binding units and their binding time, architecture models, and a design object model. For dynamic reconfiguration of feature binding units, variation points corresponding to each binding unit should be identified in the design object model and implemented with appropriate binding techniques. Dynamic binding of objects, menus, and plug-ins are techniques that support dynamic binding of components.

We also need to analyze the change impact of a reconfiguration carefully. For example, behavior of HSR Mode Manager should be changed for a new product configuration. As shown in Figure 7, we applied the Template Method pattern to dynamically change the behavior of HSR Mode Manager (Figure 7 shows the component specification): it has four different behavior specifications that cover combinations of optional service features (e.g., a selection of TP and/or SM). After Master Configurator determines a product configuration at run time, an appropriate behavior control component is bound to HSR Mode Manager to manage interactions among service features.

In this section, we have illustrated how a product line is analyzed and core assets are developed through the product line asset engineering process. In the next
section, activities required to develop a reconfigurator are explained.

![Diagram](https://via.placeholder.com/150)

Figure 7. Mode manager component specification

3. Engineering of dynamic reconfiguration

A system must maintain integrity during reconfiguration. It may crash or show incorrect behaviors, if 1) a reconfiguration is triggered at an inappropriate situation (when to reconfigure), 2) an inappropriate reconfiguration strategy is selected (how to reconfigure), or 3) a reconfiguration transaction changes some parts of the system that should not be changed (what to do to reconfigure). Therefore, these three concerns of dynamic reconfiguration (i.e., when, how, and what) should be analyzed thoroughly and specified precisely.

In this section, ways to analyze and specify context information, reconfiguration strategies, and reconfiguration actions are introduced. Also, a conceptual design of a dynamic reconfigurator is explained.

3.1 Context analysis

In our approach, decisions of when to start a reconfiguration are analyzed through an operational context analysis. The context analysis consists of three sub-activities: contextual parameter identification, situation definition, and mapping of each situation to a reconfiguration request. Each of these activities is explained with examples in the following.

The context analysis starts with identification of contextual parameters of a product line. A contextual parameter is defined as an environmental element that has a piece of information about the system’s context (e.g., brightness of a current location, battery remaining time). Once contextual parameters are identified, we refine them by defining attributes of each parameter. An attribute may be data type, sampling rates, and validity conditions. (See Table 2 for a sample of dynamic parameter definition for HSR.) In the Type column, the types of contextual parameter values are defined with their units, if applicable. The Sampling Rate defines how often the contextual parameters should be checked. A contextual parameter may be valid only if its value is within a range or one of a pre-defined set of values: such conditions are defined in the Validity column. The validity conditions of each contextual parameter should be satisfied before a contextual parameter is used to detect contextual changes.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>Type (unit of value)</th>
<th>Sampling Rate</th>
<th>Validity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brightness (B)</td>
<td>Integer (lux)</td>
<td>10 seconds</td>
<td>0 &lt;= B &lt; 100</td>
</tr>
<tr>
<td>Battery Remaining Time (BRT)</td>
<td>Integer (minute)</td>
<td>10 seconds</td>
<td>0 &lt; BRT &lt; 300</td>
</tr>
</tbody>
</table>

Table 2. Contextual parameter definition

Then, situations are specified as a logical expression of dynamic parameters. A situation is an event that triggers dynamic reconfiguration. For instance, a Robot in the Dark situation is true when Brightness (B) is less than 5, Battery Remaining Time (BRT) is greater than 30, and the availability of SM is true. On this situation, a Bind SM and Start Surveillance reconfiguration request must be triggered, and the dynamic reconfiguration strategy for the reconfiguration request must be analyzed and specified.

3.2 Dynamic reconfiguration strategy and action specifications

A dynamic reconfiguration strategy is about "how" to perform dynamic reconfiguration. The strategy is specified with consideration of binding dependencies (i.e., require and exclude), change impacts to other binding units, and required resources (e.g., components). Each reconfiguration strategy is specified for six reconfiguration phases. The six phases are: 1) check pre-conditions, 2) send a Suspend event to currently active binding units that are involved in reconfiguration, 3) remove or parameterize binding units that have to be deleted or changed, 4) instantiate and bind binding units that are newly added, 5) check post-conditions, and 6) resume suspended binding units and start newly added binding units.

The first and fifth phases that check pre/post conditions of a reconfiguration are performed by checking "required" and "excluded" binding units. For example, a current configuration must have "required" binding units (e.g., USER NOTIFY requires SM in a product) and must not have "excluded" binding units (e.g., SM cannot be bound, if TP is currently active.). Also, the reconfigured product is checked whether it includes all feature binding units of the target configuration.
At the second phase, we determine whether an involved feature binding unit should be suspended or should provide services continuously during reconfiguration. At the third phase, binding units that are no longer needed in the target product configuration are removed. If a binding unit's behavior can be changed through parameterization, appropriate parameters for the new configuration are sent to the binding unit. At the fourth phase, binding units to be added newly are instantiated and bound to the product. Finally, binding units are activated, after checking the post-conditions of the reconfiguration.

A reconfiguration action specification refines abstract events (e.g., Suspend, Reconfigure, and Resume) of reconfiguration strategy into concrete commands, which will be sent to components to control their behaviors and configurations. For example, newly instantiated architectural components are bound using the "weld" command, which binds an architectural component to connectors in C2 style. (Figure 8 shows a part of reconfiguration action specification for the Bind SM and Start Surveillance request.) In the next section, a conceptual model of a reconfigurator is introduced.

```c
dynamicReconfiguration_BindSMandStartSurveillance {
    ... invoke_methods checkCurrentArchitectureConfiguration(SM); ... invoke_methods setSuspendedCC; invoke_methods setSuspendedUF; ...
    add 
        condition IsInstantiated(SecurityMonitoring) == YES;
        condition IsInstantiated(FrontCamera) == YES;
    ... weld (topConnector, SecurityMonitoring)
    ... weld (SecurityMonitoring, bottomConnector)
    ... }
```

Figure 8. Reconfiguration action specification for Bind SM and Start Surveillance

### 3.3 Conceptual model for a reconfigurator

For the design of a reconfigurator, the conceptual model shown in Figure 9 is proposed. The conceptual model consists of two parts: Master Configurator and Local Configurator(s). Master Configurator is responsible for monitoring the context and product status, and processing reconfiguration requests. Local Configurator(s) provides Master Configurator with product state information by analyzing messages at each connector and executes reconfiguration commands received from the Master Configurator.

The **Master Configurator** consists of four conceptual components, and specifications of the three concerns for dynamic reconfiguration (i.e., situation, reconfiguration strategy, and reconfiguration action specifications) are allocated to the conceptual components. These conceptual components collaborate with each other to reconfigure a product at run time. (The right portion of Figure 9 depicts this concept.) The role of each component is described in the following:

- **Context Analyzer** collects external data from context probes and internal data from Local Configurator(s), and determines when to trigger reconfiguration by analyzing collected data.

- **Reconfiguration Strategy Analyzer** maintains current product configuration and determines a reconfiguration strategy after validating a reconfiguration request under current configuration and available resources.

- **Resource Manager** maintains resource information such as available software components and hardware devices.

- **Reconfiguration Handler** has reconfiguration action specification and performs reconfiguration as it communicates with local configurators.

In this section, activities to analyze and specify three concerns of reconfiguration and a conceptual model for reconfigurator design are explained. The next section discusses and evaluates our approach.
4. Discussion

In this section, we discuss some benefits and shortcomings of our approach. Our approach takes feature binding analysis results as a key driver for developing dynamically reconfigurable core assets. This feature binding based view can provide product line engineers with the following guidelines:

- **Identification of units of product configuration with right granularity**: A binding unit contains a set of features that need to be bound together into a product to provide services correctly. Therefore, all components that implement features in a feature binding unit should be bound together when the binding unit is bound at run time. This grouping reduces complexity in managing dynamic changes of product configuration, thus helping engineers to analyze change impacts of a dynamic reconfiguration request.

- **Explicit expression of feature binding time**: Though the importance of feature binding time is now being recognized in the literature [4,21,22], difficulties for analyzing and specifying feature binding time have not been addressed explicitly. In our approach, feature binding time of a feature binding unit is determined using two viewpoints: product line lifecycle view and feature binding state view. We found that binding time decisions with these two views could identify clearly what features are required to be bound dynamically and what features are not.

- **Improved management of variation point dependency**: With the feature binding based view, binding dependency between variation points at run time can be identified and managed efficiently with mappings to feature binding units. For instance, it became clear that *USER NOTIFY* could be bound at run time only after its parent binding unit, *SM*, was bound. We could also explore appropriate dynamic binding techniques based on this information.

Our approach, however, needs to be extended to address other issues for dynamic reconfiguration such as exception handling strategies during dynamic reconfiguration, a formal base for analyzing consistency between various specifications (e.g., behavior, DFD, and reconfiguration specifications), and a support for evolutionary changes [23]. The next section summarizes some related works in the literature.

5. Related works

Most efforts in product line engineering have focused on core asset development with variation points for static configuration of products: identification and specification of variation points, consistency management among them, and techniques for product code generation [1,2,3,4]. Recently, Reconfigurable Product Line UML Based SE Environment (RPLUSEE) [5] is proposed and its specialty is the provision of software dynamic reconfiguration patterns. Depending on the location of dynamic reconfiguration information, these patterns are classified into master-slave, centralized, client-server, and decentralized. This method also provides reconfiguration Statechart and reconfiguration transaction models for the dynamic reconfiguration. This approach focuses on high-level specifications of dynamic reconfigurable units; however, it does not describe techniques and guidelines for reconfigurable component identification, design, and implementation in detail.

On the other hand, most efforts in dynamic product reconfiguration (e.g., context-aware and self-healing systems) have focused on specific problems in each application area: behavior models to adapt dynamic changes of operational context [6], recognition of current situations by analyzing data from software or hardware environments [10,11,12], and autonomous management of product configurations and software component versions [7,8,9,13,14]. These techniques are essential to support dynamic product reconfiguration; however, how these techniques can be organized to support product line variations has not been addressed. That is, a support for static and dynamic variations of product configurations, which may differ from one product to other, has not been considered in the fields of dynamic reconfigurations.

6. Conclusions

As the number of product features increases, management of features and their variations became a big burden to product line asset developers. This is even more difficult for the development of core assets for dynamically reconfigurable products. We have alleviated this difficulty through grouping of features into feature binding units that has the same binding time and by taking feature binding units as a key design driver. In our approach, we have attempted to improve manageability of product variations and visibility of change impacts of a reconfiguration, which are critical factors in dynamic product reconfiguration.

For the development of a reconfigurator, we proposed three concerns of dynamic reconfiguration and provided ways to specify each of these concerns. With the step-by-step engineering guidelines, we have addressed difficulties in analyzing and defining complex contexts of a product line, and in connecting the contextual information to reconfiguration transactions.

Currently, a prototype demonstrating the feasibility of the method has been developed [24]. A formal
framework for analyzing consistency between binding unit behaviors and dynamic reconfiguration specifications is in progress. Our future work includes researches on exception handling strategy to improve dependability of dynamic reconfiguration.

7. References


