Hindawi Publishing Corporation
Mathematical Problems in Engineering
Volume 2015, Article ID 435316, 11 pages
http://dx.doi.org/10.1155/2015/435316

Research Article

An Effective Methodology with Automated Product Configuration for Software Product Line Development

Scott Uk-Jin Lee

Department of Computer Science and Engineering, Hanyang University ERICA, Ansan 426-791, Republic of Korea

Correspondence should be addressed to Scott Uk-Jin Lee; scottlee@hanyang.ac.kr

Received 25 February 2015; Accepted 7 June 2015

Academic Editor: Sanghyuk Lee

The wide adaptation of product line engineering in software industry has enabled cost effective development of high quality software for diverse market segments. In software product line (SPL), a family of software is specified with a set of core assets representing reusable features with their variability, dependencies, and constraints. From such core assets, valid software products are configured after thoroughly analysing the represented features and their properties. However, current implementations of SPL lack effective means to configure a valid product as core assets specified in SPL, being high-dimensional data, are often too complex to analyse. This paper presents a time and cost effective methodology with associated tool supports to design a SPL model, analyse features, and configure a valid product. The proposed approach uses eXtensible Markup Language (XML) to model SPL, where an adequate schema is defined to precisely specify core assets. Furthermore, it enables automated product configuration by (i) extracting all the properties of required features from a given SPL model and calculating them with Alloy Analyzer; (ii) generating a decision model with appropriate eXtensible Stylesheet Language Transformation (XSLT) instructions embedded in each resolution effect; and (iii) processing XSLT instructions of all the selected resolution effects.

1. Introduction

The need for an efficient way to build high quality software-intensive systems, which can satisfy the diverse needs of different market segments, has led to the substantial adoption of software product line (SPL) [1] in software industry. SPL is a software development paradigm that produces a collection of similar software-intensive systems by assembling products in different configurations from a set of reusable software artifacts. Through the reuse of quality assured software artifacts, SPL enables the low cost production of high quality software systems that are customizable for particular market segments.

In SPL, a detailed product line design to correctly specify the targeted set of software systems and an effective product configuration that is valid against the specifications are essential for providing successful product line development and maintenance [2]. According to a thorough analysis of the application domain, a SPL should be designed as a structured collection of reusable software components where variability and commonality are identified and distinguished for each component to accommodate all possible product specifications of the considered market segments. Additionally, dependencies between components, especially the ones that are variable, and the constraints referring to business rules on the targeted software systems should be specified for SPL to represent the complete requirements of the application domain. Once a SPL is correctly designed, software systems can be configured by selecting appropriate variable components according to the specifications of the desired systems. During the configuration, the selection of components should reflect the variability, dependencies, and constraints specified in the design to meet the requirements of a software system.

As described above, the lifecycle of a SPL can be complicated by the factor of the number of variable components, dependencies, and constraints. As a consequence of such complication, the costs of SPL processes increase significantly in terms of time and effort. This is largely because software components with their associated properties and constraints are considered as high-dimensional data which is often too complex to model, analyse, and calculate. The main difficulties of implementing a SPL development are the following:

(i) *Variability expression in SPL design:* a detailed expression to specify variability must be provided to describe every configurable product in the design. In addition, constraints and dependencies associated with variability must also be considered regardless of whether they are specified as requirements or propagated from the way components are structured and designed.

(ii) *Ensuring consistency of SPL design:* the possible design errors, such as conflicts between constraints and dependencies associated with variability, must be analysed, detected, and prevented.

(iii) *Validation for full coverage of products:* the design of a SPL must ensure the full coverage of products where all possible software systems defined for a given product line must be configurable from the design.

(iv) *Validation of products:* the possibility of generating invalid software systems with respect to the specified requirements must be prevented through the validation of product configuration.

These challenges, if not dealt with properly, may jeopardize the benefits of SPL development by producing a large overhead in terms of time and effort. In order to address this issue, various approaches have been proposed to better manage the SPL development process. Initially, the Feature Oriented Domain Analysis (FODA) method [3] has been developed where the feature model [4] was first introduced to provide a high level specification of product line requirements that is easy to understand and simple to construct. After the introduction of FODA method, there were different proposals for SPL development methodologies [5–7] where feature models are utilized and extended to better manage variability of SPL. Besides the FODA-based approaches, Korba [8] and Pulse [9, 10] approaches proposed a decision model [11, 12] where the variability of SPL is managed by specifying possible decisions on variability and the associated dependencies between them. These approaches have pioneered the field of software product line and provided adequate ways to manage variability and configure a product. However, in these approaches, the large part of the product line development processes, from the design of product line model to the product derivation, must be conducted manually and still requires extensive amount of time and effort. In addition, these approaches also lack adequate means to ensure the correctness of designed SPL models and the validity of configured products.

In order to address these points, various approaches [13–16] have been proposed more recently to provide improved SPL modeling with better variability management. However, there is still a lack of methodology merged with techniques and tool supports to establish a time and cost effective product configuration process that is automated, scalable, and proven to be valid.

In this paper, we present an effective approach with automated product configuration to address the current shortcomings of the SPL development process. In the proposed methodology, eXtensible Markup Language (XML) is adopted to specify SPL design where variability with associated constraints and dependencies are expressed according to the developed schema. Additionally, with the formal reasoning tool called Alloy Analyzer [17] and the developed set of tool supports, an automated product configuration is enabled in a time and cost effective manner. During the product configuration process, a decision model is constructed with resolution effects for each decision specified as eXtensible Stylesheet Language Transformations (XSLT) [18] to automate the product derivation. Furthermore, the automated product configuration ensures the full coverage of all possible products and prevents the possibility of configuring an invalid product.

The remainder of this paper is organized as follows. Section 2 summarizes and discusses related research. Section 3 presents background information on Alloy Analyzer and decision model. In Section 4, the proposed methodology and tool supports for constructing a SPL model and configuring a product with Alloy Analyzer is explained in detail. In addition, the case study of a SPL development for a smart home system is used as an example throughout this section to better illustrate the techniques and tool supports established in the proposed approach. Finally, Section 5 concludes the paper and addresses the possible future works.

2. Related Work

As already mentioned in the Introduction, there are a number of related approaches that propose a better way to manage variability and configure a valid product from SPL. Despite considerable improvements on SPL processes, many approaches still require a large amount of time and effort to configure a valid product and are too domain specific to be applied to the general SPL development.

Cirilo et al. [19] proposed a model-based product derivation tool, GenArch, that automatically derives a software instantiation from a given SPL using tool generated feature, architecture, and configuration models. However, it still requires manual refinements of the tool generated models which may take significant amount of time and effort for a SPL with large amount of reusable components and complex constraints.

Weiss et al. [20] proposed an approach where variabilities are represented in multiple graphs and the graph walking algorithms are developed to automatically derive products. Similarly, White et al. [21] developed an approach to automatically calculate variability configuration in product derivation by defining a formal model of system family and using Constrain Satisfaction Problems (CSPs) solver. Although, these approaches provide automated product derivation, they must represent variability constraints by constructing multiple graphs of formal defined models which are complicated tasks and may require extensive amount of time and effort.

More recently, Fuentes and Gámez [22] applied SPL approach to automatically generate customized middleware instantiation for ambient intelligence (AmI) systems.

The approach consists of various tool supports to deal with complex and heterogeneous nature of the AmI system and enable automated generation of AmI middleware with minimal user inputs. However, this approach focuses specifically on the SPL for AmI middleware and is difficult to apply on a general SPL problems.

Most of the approaches and their associated tool supports proposed in the related research either produce significant overhead in configuring a product from a given SPL or focus on solving problems of a specific domain. The approach and the associated tool support proposed in this paper aim to provide a time and cost effective product configuration process that are automated, applicable to the general SPL development, and with minimal overhead. Moreover, our approach provides a means to ensure the validity of automatically configured products by detecting possible conflicts in variability constraints with the formal computation of Alloy Analyzer and validating the design of SPL with the comparison of decision model against the requirements extracted from the business domain.

3. Background

This section presents a brief description of the decision model specific to SPL, explaining how it provides interactive guideline for product configuration and enables the automation of product derivation. In addition, a short introduction to Alloy formal language and the associated reasoning tool, Alloy Analyzer, is presented in terms of its underlying logic, language constructs, and formal computation.

3.1. Decision Model. The decision model [11] is a model constructed to guide product configurations and to ensure the full coverage of products. It consists of decisions, possible resolutions for each decision, and set of effects related to each resolution. Using these constructs, a decision model represents all possible product configurations for a SPL and aid product derivation. The construction of a decision model involves the following steps:

(1) Define each variability constraint of a SPL model as a decision.

(2) Compute all possible choices for each decision that satisfy the corresponding variability constraints and define them as resolutions related to the decision.

(3) Derive set of effects for each resolution that describes how a variability specified in the SPL design can be fixed to reflect the choice made on the decision.

The decision model constructed as above takes various structural forms depending on the implementation choices. For example, decisions are represented as table of check lists in some approaches [8, 11] whereas they are represented as elements of an XML document in other approaches [12, 23]. In the proposed approach, a decision model can be constructed as a tree structure in an XML format where decisions, resolutions, and effects are organized using the hierarchical parent-child relationships. The constructed decision model describes all possible product configurations with each full path of the tree representing a well-formed product configuration. Note that the effects associated with each resolution in the decision model are specified using the template elements of the eXtensible Stylesheet Language Transformations (XSLT) to enable automated derivation of product model. Additionally, the proposed approach provides optional representation for the created decision model in the customized Unified Modeling Language (UML) activity diagram [24] where decisions, resolutions, and effects are specified as nodes and directed edges. This additional representation is incorporated in the product configuration tool where user can interactively traverse down the tree and select resolutions for each decision nodes to construct a full product configuration path that lead to a valid product. According to the user selected product configuration, the derivation of a product model can easily be automated with a simple script for collecting and processing the XSLT template elements specified in the effects associated with a given configuration path.

3.2. Alloy and Alloy Analyzer. Alloy [17] is a formal language developed by Software Design Group at Massachusetts Institute of Technology to model complex structural relationships and constraints. It is based on first order relational logic and utilizes the standard mathematical syntax. Alloy provides various constructs to increase its expressibility, but only the signature and the fact constructs are utilized in the context of product configuration. A signature denotes the set of entity objects to specify basic type and to introduce relations between entity objects. A fact denotes constraint on entity objects and relations that must be satisfied all the time. In addition, Alloy provides a set of predefined cardinality specifications of "`one`," "`lone`," "`some`," and "`set`." They are equivalent to the cardinality of one-to-one, zero-to-one, one-to-many, and zero-to-many, respectively. These are used to specify and calculate cardinality constraint between entity objects in signature or in relations.

Alloy Analyzer [17] is an automated reasoning tool coupled with Alloy language. A problem expressed in Alloy language can be automatically solved using Alloy Analyzer where instances of the specification, which satisfies the associated constraints, are generated through a satisfiability (SAT) solver [25]. The resulted instances can be viewed as a snapshot of the specification. Hence, Alloy Analyzer can effectively calculate the variability constraints of SPL and generate the solution in desired formats. Moreover, it can automatically detect conflicts between variability constraints during the calculation.

4. SPL Development Methodology with Automated Product Configuration

The proposed SPL development methodology with automated product configuration aims to design a SPL model and configure a valid software-intensive system in a time and cost effective manner. It is established by providing an adequate methodology with associated tool supports where XML related technologies and formal reasoning techniques are applied in the context of SPL development. The main focus

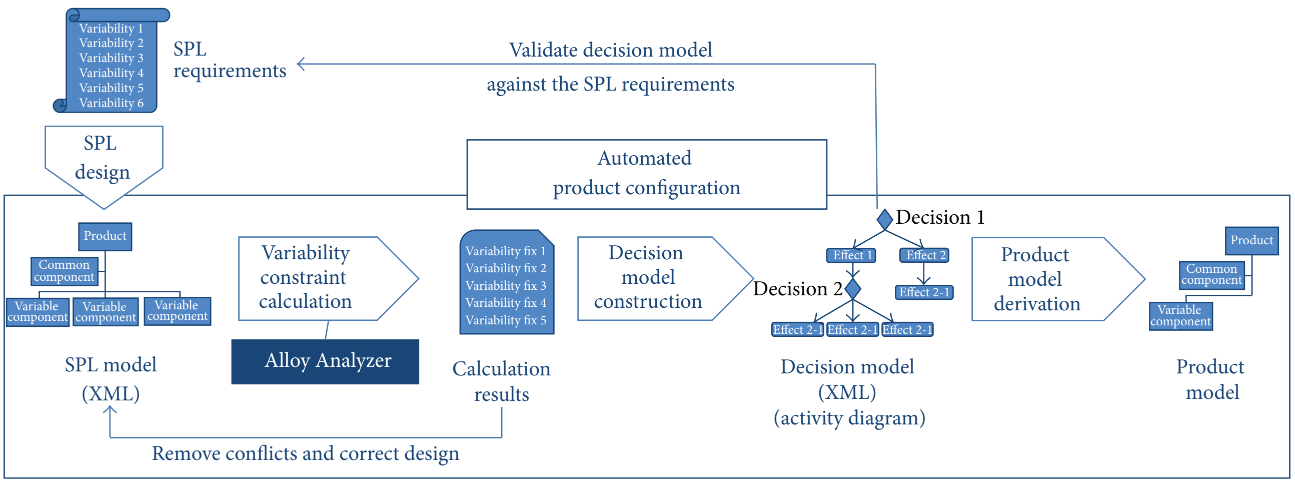


FIGURE 1: Overall software product line development process.

of the proposed approach is to provide (i) an appropriate representation of variability with associated constraints and dependencies that can be readily processed and (ii) tool supports that can automatically calculate variability constraints and produce a valid product configuration according to the user selection.

4.1. Overall Software Product Line Development Process. The established SPL development process consists of four different phases to (i) design SPL model; (ii) calculate variability constraints; (iii) construct decision model; and (iv) derive product model. In each phase, appropriate technologies are adopted and associated support tools are developed to automate the product configuration process. The overview of the proposed approach, describing different phases of the established SPL development process, is shown in Figure 1. It illustrates the entire process for SPL development established in the proposed approach where each phase will be described in detail in the following subsections. In addition, a part of the simplified SPL development case study for smart home software system will be used as an example throughout this section to better explain the proposed approach.

4.2. Software Product Line Design. The initial phase of the proposed development process is to design a SPL where a collection of similar software systems is specified in a single model. In this phase, a SPL is modeled in terms of reusable components according to the product requirements extracted from the application domain analysis. During the design, variability of the reusable components, its associated constraints, and dependencies between components distinguished from the domain analysis must also be expressed in the SPL model appropriately.

4.2.1. XML Schema for SPL Model Design. In order to design a SPL model in a precise and clear manner, the proposed approach adopts XML and its related technologies. An XML schema for representing a collection of reusable components in a tree structure and expressing variability with associated constraints is defined to guide the design process.

Figure 2 shows an XML schema that defines the metamodel to customize XML specifically for the design of SPL models. According to the developed schema, a SPL model can be designed with the following three main elements:

(i) The *ProductLine* element: it is the root element representing a SPL model to be designed where the name for the model must be specified using the *name* attribute. It can have arbitrary number of *component* and *variability* elements representing reusable software components and variability, respectively, to construct a SPL model.

(ii) The *component* element: it represents reusable software components in a SPL. Similar to the *ProductLine* element, the name of *component* element must be specified with the *name* attribute to distinguish which component the element refers to. Also, arbitrary number of other *component* and *variability* elements can be nested within this element to organize the reusable components into the tree structure of a SPL model. In addition, the *component* element can have *dependency* elements that are used to specify dependencies between components in the designed model.

(iii) The *variability* element: it represents variability in a SPL model. Among the entire reusable components of a SPL, each set of variable components that shares the same constraint is specified as children of a *variability* where associated variability constraints can be specified using the *type* attribute.

As Figure 2 illustrates, *ProductLine* element can be specified with a nesting of *component* and *variability* elements to construct a tree structured XML model of SPL. Besides the structural design of the SPL model, variability with associated constraints and dependencies between components can also be specified effectively with the developed XML schema.

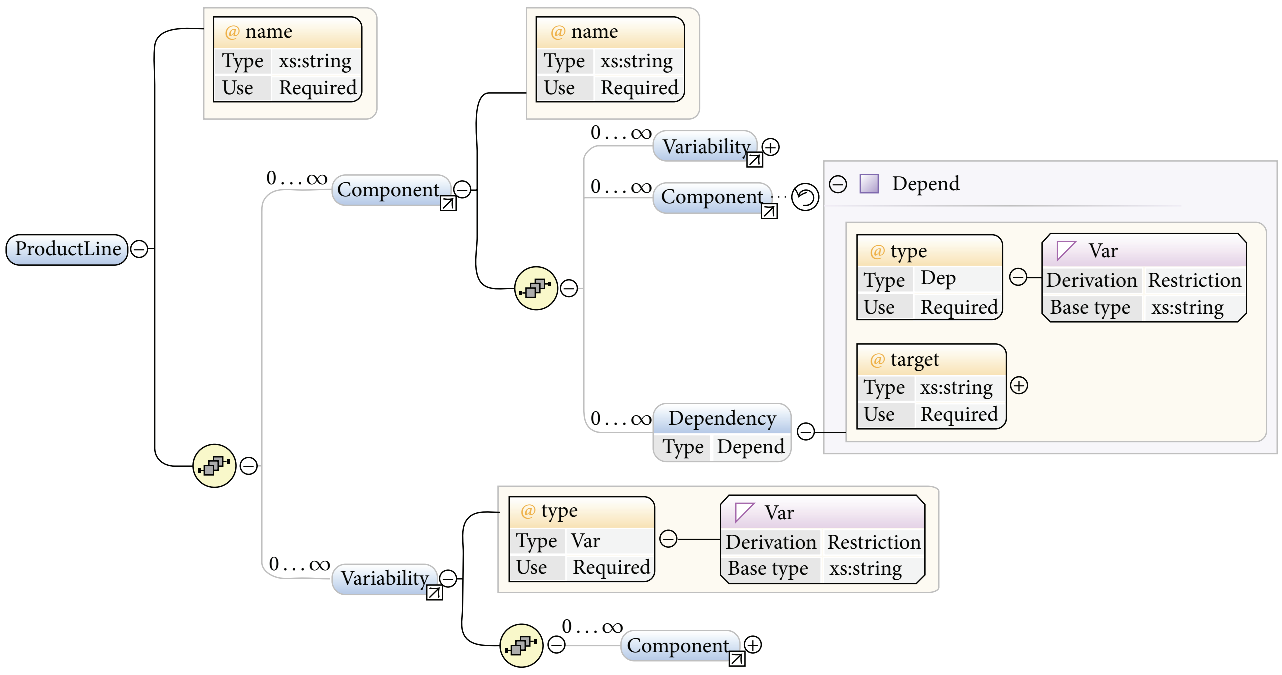


Figure 2: XML schema for software product line design.

Table 1: Predefined variability constraints and dependencies.

alternative	Only a single component from a set can exist
atLeastOne	At least one component from a set must exist
allPossibilities	Any combination of components from a set can exist
implication	Existence of the component implies that the targeted component also exists
equivalence	Existence of the component and the targeted component is dependent on each other where either all or none exist

The variability constraint to be applied on a set of variable components can be specified using the *type* attribute of a *variability* element. Although it is not shown in Figure 2, the value restriction on the *type* attribute is defined with the enumeration of predefined variability constraints as listed in Table 1.

According to the enumeration value selected for a *type* attribute, the corresponding constraint applies to the set of components that are children of the associated *variability* element. With the predefined constraints, dependencies between components can also be specified in the SPL model through the *target* and the *type* attributes of a *dependency* element. The value of the *target* attribute is specified with the name of the component that exists in the SPL model representing the targeted component for the dependency. The possible value for the *type* attribute is also restricted with the enumeration of predefined variability constraints. Note that the variability constraints are predefined to ease the design of SPL model without limiting its expressibility.

4.2.2. SPL Model for a Smart Home System in XML. In this section, a simplified version of the SPL development case study for a smart home software system is introduced to illustrate how a SPL model can be designed according to the developed schema. This case study is specific to a part of the system that manages the illumination of a house. The corresponding illumination manager software package can be configured with one of three different illumination systems described below:

(1) Automated system: the system automatically controls the artificial and natural lightings of a smart home. It consists of fully integrated lighting, occupancy sensor lighting, and automated shade control modules where at least one of the three modules have to be chosen to implement the system.

(2) Assisted system: the system is fitted with configurable lighting module. It still controls the artificial lightings in a smart home automatically but only after the lightings are configured by the user through the touchscreen smart controller module.

(3) Manual system: the user manually controls the lighting in a smart home.

In addition, the illumination manager has a lighting control system where either touchscreen smart controller module or wall switch module must be chosen to control the lighting in a smart home if needed. The results of requirement analysis for the illumination manager system described above are as follows:

(i) An illumination manager must have one illumination system either automated, assisted, or manual.

```
<ProductLine name="Smart Home"...>
...
<component name="Illumination Manager">
  <variability type="alternative">
    <component name="Automated Illumination System">
      <variability type="atLeastOne" >
        <component name="Fully integrated lighting"/>
        <component name="Occupancy sensor lighting"/>
        <component name="Automated shades control"/>
      </variability>
    </component>
    <component name="Assisted Illumination System">
      <component name="Configurable lighting">
        <dependency type="implication"
                    target="Touchscreen Smart Controller"/>
      </component>
    </component>
    <component name="Manual Illumination System"/>
  </variability>
  <component name="Lighting Control System">
     <variability type="alternative">
     <component name="Touchscreen Smart Controller"/>
     <component name="Wall Switch"/>
   </variability>
  </component>
</component>
...
</ProductLine>
```

Algorithm 1: Software product line model designed in XML.

(ii) An automated illumination system must have at least one of fully integrating lighting, occupancy censored lighting, and automated shade control modules.

(iii) An assisted illumination system has a configurable lighting module that requires smart controller module.

(iv) An illumination manager can have only one lighting control system among touchscreen smart controller and wall switch modules.

In order to configure an illumination manager for a smart home that satisfies all these extracted requirements, a SPL model for the system is designed in XML format according to the developed schema.

Algorithm 1 shows the SPL model constructed for the illumination manager package of a smart home system described above. It indicates that the illumination manager component consists of an alternative type variability element with three illumination systems and a light control system component. It expresses that the automated, assisted, and manual illumination systems are variable components as they are direct children of a variability element. On the other hand, the lighting control system without a variability element as its parent is a common component. The alternative type variability element with three illumination system components indicates that the illumination manager can have only one of automated, assisted, and manual systems. Similarly, the alternative type variability element under the lighting control system component expresses the same variability constraint for two different light controller modules. In the designed SPL model, the configurable lighting component is specified as a direct child of the assisted illumination system component indicating that configurable lighting module must exist for an assisted illumination system. In addition, the configurable lighting component has an implication type dependency to the touchscreen smart controller component expressing that the configurable lighting module must have the touchscreen smart controller module.

As illustrated in Algorithm 1, the XML representation of a SPL model is easy to read and understand. With the defined XML schema, the design process of a SPL model also requires less effort once the requirements of an application domain are analysed and extracted. In addition, the designed SPL model can be validated against the provided schema for ensuring the structural correctness of the XML representation.

4.3. Variability Constraint Calculation. In this phase, the entire variability constraints specified in a SPL model are calculated with Alloy Analyzer and all possible product configurations are computed accordingly. Prior to this process, a SPL model is analysed initially to extract all the variability constraints and the dependencies between components specified in the model. Additionally, during the analysis, the transitive variability that is propagated through the structure of

```
// signature definition for variable elements
lone sig component1, component2, component3, component4, component5... {}

fact {one component1 + component2}                  // alternative constraints
fact {some component3 + component4}                 // atLeastOne constraints
fact {set component5 + component6}                  // allPossibilities constraints
fact {one component7 implies one component8}        // implication constraints
fact {one component9 iff one component10}           // equivalence constraints

// variability constraints nested within a variable component
fact {one component3 iff (some component11 + component12)}
```

Algorithm 2: Alloy specification of variability constraints.

the model is detected and considered for calculation to ensure the computed product configurations to be valid against the product specification. The propagated variability constraints are detected by finding all the *component* elements that are not a direct child of a *variability* element but a descendent. Although not specified explicitly, this kind of components is considered variable as they can exist in the final product if and only if its parent component exists. The *equivalence* constraint is assigned to the propagated variability to correctly specify the constraints between the component involved and its parent component. In order to automate the entire product configuration process, an adequate tool is developed to automatically extract all the variability constraints and dependencies including the propagated ones.

In the SPL model constructed for a smart home, three variability constraints, a dependency, and a propagated variability constraint are detected and extracted by the automatic tool support. The extracted constraints and dependencies are (i) an *alternative* variability constraint for automated, assisted, and manual illumination system components; (ii) an *atLeastOne* variability constraint for fully integrated lighting, occupancy sensor lighting, and automated shades control modules; (iii) an *alternative* variability constraint for touchscreen smart controller and wall switch modules; (iv) an *implication* dependency from configurable lighting module to touchscreen smart controller module; and (v) an *equivalence* variability constraint for assisted illumination system and configurable lighting module. Note that the *equivalence* constraint represents the propagated variability as the configurable lighting module can exist in the product if and only if its parent, the assisted illumination system, exists.

Once the entire variability constraints and dependencies are extracted, they are specified in Alloy language and calculated with Alloy Analyzer. In order to provide an adequate calculation, a tool that automatically creates an Alloy specification from extracted variability constraints and calculates all possible product configurations is developed. The developed tool supports incorporate the Java implementation of Alloy Analyzer as a solver to provide valid computation results. Hence, the tool support also creates the Alloy specification that is in the format required by the Java implementation of Alloy Analyzer.

Algorithm 2 illustrates how the extracted variability constraints can be parsed into an Alloy specification. Initially, all the variable components are declared as `signatures` where the cardinality is specified as `lone` indicating that the declared components are variable. Then the variability constraint is represented as a `fact` of cardinality specification on the set of associated components since the constraints represent which combination of elements can appear in the final product. The *alternative*, *atLeastOne*, and *allPossibilities* constraints can be specified with `one`, `some`, and `set` cardinalities, respectively, whereas the Alloy already has `implies` and `iff` operators to specify *implication* and *equivalence* constraints, respectively. Note that the cardinality `one` is specified for the components involved in *implication* and *equivalence* constraints to express the existence of the components in the configured product. For example, `fact{one component7 implies one component8}` expresses that the existence of `component7` in the configured product implies the existence of `component8`. The Alloy specification described so far applies to all the variability constraints and the dependencies except for the variability constraints nested within a variable component. When a variability is nested within a variable component, the existence of the components associated with the variability solely depends on the existence of the variable component. Hence, the *equivalence* constraint between the variable components and the nested variability constraint is specified to provide a correct Alloy specification. The last line in Algorithm 2 indicates how the *atLeastOne* variability constraint nested within a variable component is specified. Other kinds of nested variability constraint can be specified similarly.

Algorithm 3 shows the Alloy specification of variability constraints extracted from the smart home SPL model. The `fact` definitions, from the top to bottom, indicates (i) *alternative* for the three illumination system; (ii) *atLeastOne* variability constraints for the three lighting modules of an automated illumination system; (iii) propagated variability constraints between assisted illumination system and configurable lighting module; (iv) *implication* dependency from configurable lighting module to touchscreen smart controller module; and (v) *alternative* variability constraints for the two light controller modules. Note that the above specification in Alloy syntax is provided to better convey how the proposed technique specifies extracted variability constraints. The developed tool supports will automatically generate equivalent Java representation of this Alloy specification as

```
// IS: Illumination System                CL: Configurable Lighting
// FIL: Fully Integrated Lighting          TSC: Touchscreen Smart Controller
// OSL: Occupancy Sensor Lighting         WS: Wall Switch
// ASC: Automated Shades Control
lone sig Automated_IS, Assisted_IS, Manual_IS, FIL, OSL, ASC, CL, TSC, WS {}

fact {one Automated_IS + Assisted_IS + Manual_IS}
fact {one Automated_IS iff (some FIL + OSL + ASC)}
fact {one Assisted_IS iff one CL}
fact {one CL implies one TSC}
fact {one TSC + WS}
```

Algorithm 3: Alloy specification of variability constraints from smart home SPL.

it utilizes the Java implementation of Alloy Analyzer for variability computation.

Once the Alloy specification of the variability constraints is generated, it is calculated using Alloy Analyzer to produce all feasible combinations of the variable elements. The result of the formal computation is produced in a textual two-dimensional array and used to construct a decision model. In the calculation result, the outer array describes different product configurations possible for a given SPL whereas the inner array represents the combination of variable components chosen for a particular product configuration.

In the case study of the smart home SPL, the calculation of variability constraints using Alloy Analyzer will produce the following result:

```
[ [Automated_IS, FIL, TSC],
  [Automated_IS, FIL, WS],
  [Automated_IS, OSL, TSC],
  [Automated_IS, OSL, WS],
  [Automated_IS, ASC, TSC],
  [Automated_IS, ASC, WS],
  [Automated_IS, FIL, OSL, TSC],
  [Automated_IS, FIL, OSL, WS],
  [Automated_IS, FIL, ASC, TSC],
  [Automated_IS, FIL, ASC, WS],
  [Automated_IS, OSL, ASC, TSC],
  [Automated_IS, OSL, ASC, WS],
  [Automated_IS, FIL, OSL, ASC, TSC],
  [Automated_IS, FIL, OSL, ASC, WS],
  [Assisted_IS, CL, TSC],
  [Manual_IS, TSC],
  [Manual_IS, WS] ].
```

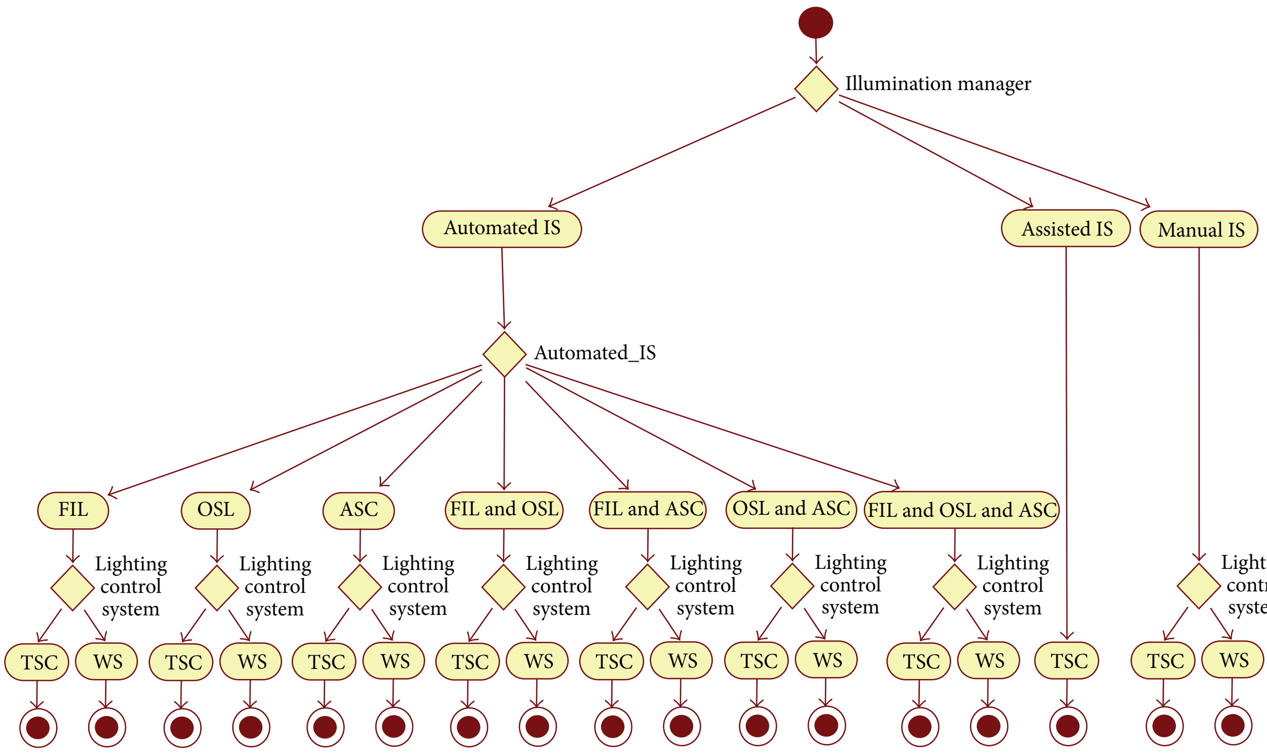
The above calculation result consists of seventeen inner arrays indicating that there are seventeen different product configurations feasible for the smart home SPL. Among these configurations, the first inner array represents the configuration of an illumination manager system with automatic illumination system, fully integrated lighting module, and the touchscreen smart controller module whereas the last inner array represents the configuration with manual illumination system and wall switch module. The rest of the inner array in the calculation result represents the possible product configuration in similar way.

Note that the formal computation with Alloy Analyzer can also detect possible conflicts between variability constraints and dependencies caused by human errors during the SPL design phase by generating an empty array as a calculation result. When the conflict is detected the designed SPL model needs to be revised to prevent possible derivation of an invalid product configuration.

4.4. Decision Model Construction. In this phase, a decision model representing all possible product configurations is constructed according to the results produced from the calculation of variability constraints. In the constructed model, each variability constraint is represented as a decision where all decisions consist of a set of resolutions and associated effects representing all possible variability fixes that satisfies the corresponding variability constraints. In the proposed approach, such a decision model is constructed with the following steps:

(1) Create a decision for a variability constraint.
(2) Derive all feasible combinations of variable components involved in the variability constraint by examining each product configuration that resulted from Alloy calculation.
(3) Create a resolution with associated effects under the decision for each combination of variable components.
(4) Repeat steps (1)–(3) for each variability constraints extracted from the designed SPL model and construct decision tree incrementally.
(5) Remove decision with a single resolution but keep the resolution and associated effects.

This process of constructing a decision model is fully automated with the developed tool support. The constructed decision model is in XML format representing all possible product configurations of the SPL. It also enables an automated derivation of a particular product configuration



IS: illumination manager
FIL: fully integrated lighting
OSL: occupancy sensor lighting
ASC: automated shades control
TSC: touchscreen smart controller
WS: wall switch

FIGURE 3: Activity diagram representation of the generated decision model.

as the resolution effects for each decision provides corresponding product configuration instructions using XSLT. In addition, the constructed decision model can be used to ensure the correctness of the designed SPL model through the validation against the set of requirements extracted from the business domain.

Figure 3 presents an automatically constructed decision model for the case study of the smart home SPL. It is an activity diagram representation of the XML formatted model generated by the user interactive tool support developed for the product derivation process. The decision model consists of seventeen full paths indicating that there are seventeen different possible product configurations for the system. The *Illumination Manager* decision corresponds to the *alternative* variability constraints specified in the SPL model for configuring the type of illumination system in the final product. It consists of three different resolutions indicated by edges directed towards the *Automated_IS*, the *Assisted_IS*, and the *Manual_IS* effects where each effect indicates the possible combination of variable components to be selected for the corresponding resolution. For example, the *Assisted_IS* effect indicates that the corresponding resolution will select assisted illumination system for the illumination manager in the final product. The effects also consist of automatically generated XSLT templates for transforming the parts of SPL model to select corresponding variable components during the product derivation phase. As an example, the XSLT template automatically generated for the *Assisted_IS* effect is shown in Algorithm 4.

This template, when applied, transforms the *variability* element that has the *Assisted Illumination System* element as a child. From all the children of the *variability* element, it keeps the *Assisted Illumination System* element by copying the element and all its decedents from the designed SPL model but removes all the other children. Additionally, the *variability* element itself is also removed. For such transformation, the above template calls the *copyComponents* template that are predefined for the product derivation tool to copy the current elements and all its decedents recursively. With the XSLT templates for *Assisted_IS* effect, selection of *Assisted Illumination System* for the *Illumination Manager* can automatically be conducted during the product derivation phase. Note that the XSLT templates are hidden in the decision model diagram to provide simpler and more understandable representation of effects to users.

4.5. Product Model Derivation. The final phase of the automated product configuration is product derivation where a particular product configuration is generated by fixing all variability specified in the SPL model according to the user selections. In this phase, an automated product derivation tool support is developed with an interactive user interface

```
<xsl:template match="//variability[component[@name='Assisted Illumination System']]">
    <xsl:for-each select="./component[@name='Assisted Illumination System']">
        <xsl:call-template name="copyComponents"/>
    </xsl:for-each>
</xsl:template>
```

Algorithm 4: Automatically generated XSLT template for the *Assisted_IS* effect.

to select a desired resolution for each decision from the decision model represented in activity diagram. In addition, a simple script is developed to automatically collect XSLT templates from each resolution effect that corresponds to the user choices made in the decision model. The collected templates are then processed into a single XSLT and executed automatically to generate the corresponding product configuration from the designed SPL model. The derived product configuration can then be passed on to the production to build final products. In addition, the validity of a derived configuration that represents a well-formed product is ensured by validating the decision model against the product line requirements prior to the derivation.

5. Conclusions

The adoption of product line engineering in software development enabled cost effective software production for diverse market segments. However, the current implementations of SPL lack an effective way to design product line model and an efficient approach to configure a valid product. This may lead to the increase in the time and cost overhead of product configuration and can even jeopardise the benefits of SPL. In order to resolve this problem, an effective methodology with tool supports is developed to automate product configuration process and ensure product validity. The proposed approach consists of the following:

(i) XML-based SPL model design method and the predefined schema that enables precise and clear specification of variability with associated constraints and dependencies.

(ii) Automated variability analysis technique to detect variability propagation, extract associated constraints, and construct a set of variability constraints from a SPL design model.

(iii) Tool support using Alloy Analyzer for automated calculation of variability constraints where the consistency of SPL design and full coverage of products are ensured.

(iv) Automated tool support for constructing a decision model and deriving produce configurations with enabled validation of SPL model against requirements extracted from the domain analysis.

These techniques and tool supports illustrated in the proposed approach enable precise and clear design of SPL model and provide a cost and time effective product configuration with ensured validity.

A possible future work of this research is to incorporate the nonfunctional properties into the proposed approach of SPL development. Once the nonfunctional property is successfully integrated into the design of SPL and the automated product configuration, a decision model with priority or ordering based on specified values of nonfunctional properties can also be implemented. For example, user can specify the decision nodes that consist of resolution effects with the highest implementation cost to be on the top of the decision tree. This will surely increase the efficiency and effectiveness of the automated product configuration established in the proposed approach.

Conflict of Interests

The author declares that there is no conflict of interests regarding the publication of this paper.

Acknowledgment

This work was supported by the research fund of Hanyang University (HY-2011-N).

References

[1] K. Pohl, G. Böckle, and F. J. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, Secaucus, NJ, USA, 2005.

[2] F. J. van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: The Best Industrial Practice in Product Line Engineering*, Springer, Secaucus, NJ, USA, 2007.

[3] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Tech. Rep. CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990, http://www.sei.cmu.edu/library/abstracts/reports/90tr021.cfm.

[4] G. Chastek and P. Donohoe, "Product line analysis for practitioners," Tech. Rep. CMU/SEI-2003-TR-008, Software Engineering Institute, Carnegie Mellon University, 2003, http://www.sei.cmu.edu/library/abstracts/reports/03tr008.cfm.

[5] M. L. Griss, "Implementing product-line features with component reuse," in *Software Reuse: Advances in Software Reusability: Proceedings of the 6th International Conference, ICSR-6, Vienna, Austria, June 27–29, 2000*, vol. 1844 of *Lecture Notes in Computer Science*, pp. 137–152, Springer, London, UK, 2000.

[6] J. van Gurp, J. Bosch, and M. Svahnberg, "On the notion of variability in software product lines," in *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pp. 45–54, IEEE, Washington, DC, USA, 2001.

[7] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, “Extending feature diagrams with UML multiplicities,” in *Proceedings of the 6th World Conference on Integrated Design & Process Technology*, Pasadena, Calif, USA, 2002.

[8] C. Atkinson, J. Bayer, and D. Muthig, “Componentbased product line development: the kobra approach,” in *Proceedings of the 1st Conference on Software Product Lines*, pp. 289–309, Kluwer Academic Publishers, Norwell, Mss, USA, 2000.

[9] J. Bayer, O. Flege, P. Knauber et al., “PuLSE: a methodology to develop software product lines,” in *Proceedings of the Symposium on Software Reusability (SSR ’99)*, pp. 122–131, ACM, May 1999.

[10] M. Becker, “Towards a general model of variability in product families,” in *Proceedings of the 1st Workshop on Software Variability Management*, Groningen, The Netherlands, 2003.

[11] C. Atkinson, J. Bayer, C. Bunse et al., *Component-Based Product Line Engineering with UML*, Addison-Wesley, Boston, Mass, USA, 2002.

[12] J. Bayer, “Towards engineering product lines using concerns,” in *Proceedings of the Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE ’00)*, Limerick, Ireland, 2000.

[13] M. Mendonca, A. Wasowski, K. Czarnecki, and D. Cowan, “Efficient compilation techniques for large scale feature models,” in *Proceedings of the 7th International Conference on Generative Programming and Component Engineering (GPCE’ 08)*, pp. 13–21, ACM, October 2008.

[14] R. Gheyi, T. Massoni, and P. Borba, “Automatically checking feature model refactorings,” *Journal of Universal Computer Science*, vol. 17, no. 5, pp. 684–711, 2011.

[15] P. Fernandes, C. Werner, and E. Teixeira, “An approach for feature modeling of context-aware software product line,” *Journal of Universal Computer Science*, vol. 17, no. 5, pp. 807–829, 2011.

[16] E. A. Oliveira Junior, I. M. S. Gimenes, J. C. Maldonado, P. C. Masiero, and L. Barroca, “Systematic evaluation of software product line architectures,” *Journal of Universal Computer Science*, vol. 19, no. 1, pp. 25–52, 2013.

[17] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*, The MIT Press, 2012.

[18] D. Tidwell, *XSLT*, O’Reilly Media, Sebastopol, Calif, USA, 2009.

[19] E. Cirilo, U. Kulesza, and C. J. P. de Lucena, “A product derivation tool based on model-driven techniques and annotations,” *Journal of Universal Computer Science*, vol. 14, no. 8, pp. 1344–1367, 2008.

[20] D. M. Weiss, J. J. Li, H. Slye, T. Dinh-Trong, and H. Sun, “Decision-model-based code generation for SPLE,” in *Proceedings of the 12th International Software Product Line Conference (SPLC’ 08)*, pp. 129–138, IEEE Computer Society, Limerick, Ireland, September 2008.

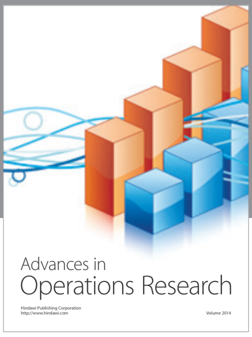
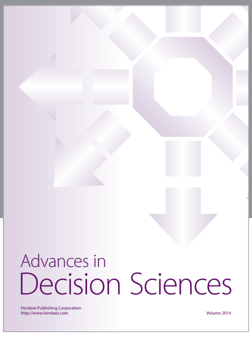
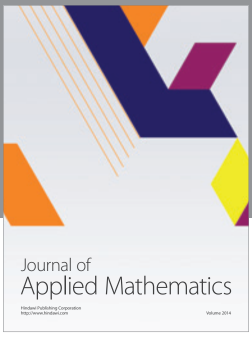
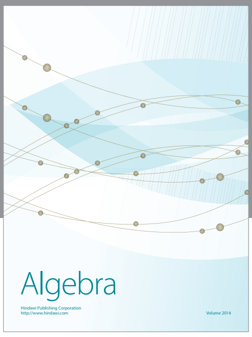
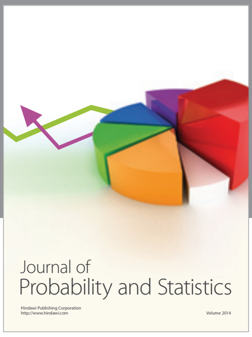
[21] J. White, D. Benavides, B. Dougherty, and D. Schmidt, “Automated reasoning for multi-step software product line configuration problems,” in *Proceedings of the 13th International Software Product Line Conference*, pp. 1–10, San Francisco, Calif, USA, August 2009.

[22] L. Fuentes and N. Gámez, “Configuration process of a software product line for AmI middleware,” *Journal of Universal Computer Science*, vol. 16, no. 12, pp. 1592–1611, 2010.

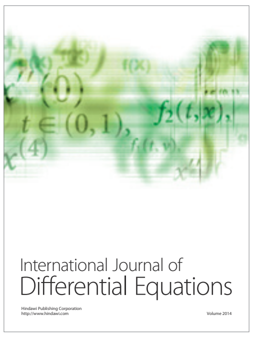
[23] J. Mansell and D. Sellier, “Decision model and flexible component definition based on XML technology,” in *Software Product-Family Engineering*, vol. 3014 of *Lecture Notes in Computer Science*, pp. 466–472, Springer, Berlin, Germany, 2004.

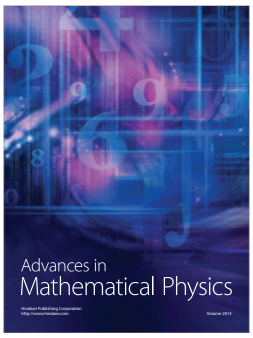
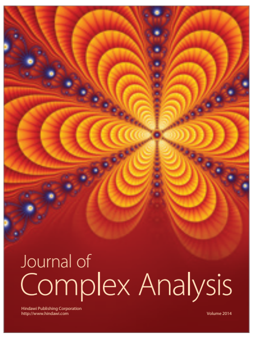
[24] J. Arlow and I. Neustadt, *UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design*, Pearson Education, 2005.

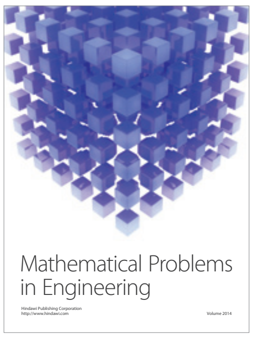
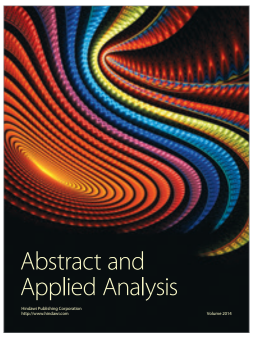
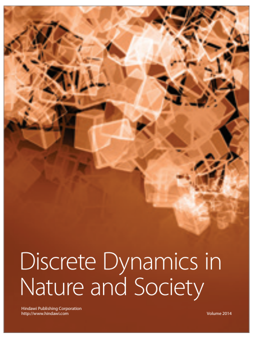
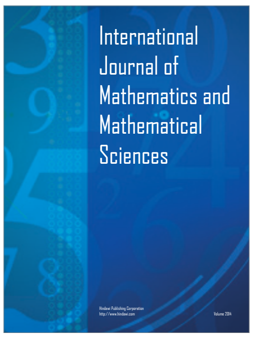
[25] D. L. Berre and A. Parrain, “The Sat4j library, release 2.2,” *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 7, no. 2-3, pp. 59–64, 2010.


Advances in
Operations Research

Advances in
Decision Sciences

Journal of
Applied Mathematics

Algebra

Journal of
Probability and Statistics

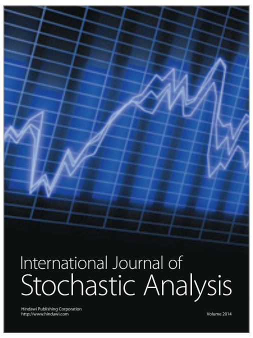
The Scientific
World Journal


International Journal of
Differential Equations

International Journal of
Combinatorics

Advances in
Mathematical Physics

Journal of
Complex Analysis

Journal of
Mathematics

Mathematical Problems
in Engineering

Abstract and
Applied Analysis

Discrete Dynamics in
Nature and Society

International
Journal of
Mathematics and
Mathematical
Sciences

Journal of
Discrete Mathematics

Journal of
Function Spaces

International Journal of
Stochastic Analysis

Journal of
Optimization