Supporting Feature Model Configuration using a Demonstration-based Approach

Yu Sun
Department of Computer and Information Sciences
University of Alabama at Birmingham
Birmingham, AL 35294
yusun@cis.uab.edu

Hyun Cho, Jeff Gray
Department of Computer Science
University of Alabama
Tuscaloosa, AL 35487
hcho7@crimson.ua.edu
gray@cs.ua.edu

Jules White
Department of Electrical and Computer Engineering
Virginia Tech
Blacksburg, VA 24060
julesw@vt.edu

ABSTRACT
Configuration of feature models in software product-lines typically involves manipulating a model to modify the feature selections and analyzing the model to ensure that no configuration constraints are violated. In order to capture and reuse configuration knowledge from different users, model transformation and constraint languages can be used to specify and automate the constraint checking and model manipulation processes. However, this approach presents challenges to general end-users (e.g., domain experts who may not be programmers) who do not have experience using these languages. This paper presents a demonstration-based technique to support the capture and reuse of feature model configurations.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms
Design, Languages

Keywords
Feature Model, Model Transformation By Demonstration

1. INTRODUCTION
Feature models have been widely used to model software product-line (SPL) variability [1]. A correct variant is configured by selecting the desired features that satisfy the product requirements without violating any feature model constraints. Instead of configuring the whole variant by a single expert, an SPL may need to be configured collaboratively by different individuals through a series of feature manipulation and analysis actions. For instance, an SPL configuration may span multiple engineering domains, such as hardware and software features, so defining a complete configuration requires participation of multiple engineers to make feature selections and perform correctness checking (e.g., check the cross-tree constraints in a feature model). In addition, the same part of a configuration may be refined by different individuals (e.g., senior engineers may address errors or undesired configurations specified by entry-level engineers).

PLEASE'11, May 22-23, 2011, Waikiki, Honolulu, HI, USA.

Based on this collaborative context, a key challenge is that not all individuals may have the required domain knowledge to completely configure a variant in a feature model. Moreover, as individuals join and leave an organization, critical configuration knowledge may be lost. Therefore, capturing configuration knowledge from different individuals and supporting feature model configuration through knowledge reuse is an essential task.

If feature models are created using domain-specific modeling tools (e.g., GEMS [7]), a simple approach to capture configuration knowledge is to save representative configuration examples in individual files or models, which can be reused in other contexts. However, reusing this type of knowledge is not flexible due to a lack of automation. Users have to first understand the reference configuration and then manually perform the necessary manipulation or analysis on their feature models to duplicate it. An alternative to automating the capture and reuse of feature model configuration knowledge is to specify the configuration and constraint rules using a Model Transformation Languages (MTL) [3], or constraint language (e.g., OCL). With this approach, the model editor can check constraints and perform necessary configuration actions automatically at modeling time when users are building the configurations. However, the usage of MTLs and constraint languages requires developers to learn these languages, which are not typically focused on product-lines, as well as the related deeper modeling concepts (e.g., understanding technical details of the metamodel). Such challenges may prevent some general end-users from capturing and specifying feature model manipulation and analysis tasks for which they have extensive domain experience. Similarly, although automated configuration approaches (e.g., formulations of feature models as a Constraint Satisfaction Problem - CSP) can provide intelligent support in feature configuration [6], optimization and error correction, there are often configuration rules that domain experts do not know how to formalize using CSP or SAT languages

To overcome these challenges and provide an end-user approach for capturing feature model configuration rules without learning a transformation or constraint language, we have investigated the idea of using Model Transformation By Demonstration (MTBD) [2] to automatically capture configuration knowledge and automate the reuse of the captured knowledge.

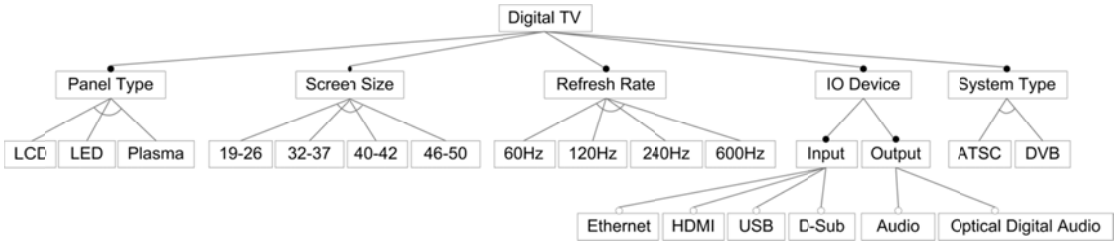


Figure 1.A simplified TV feature model

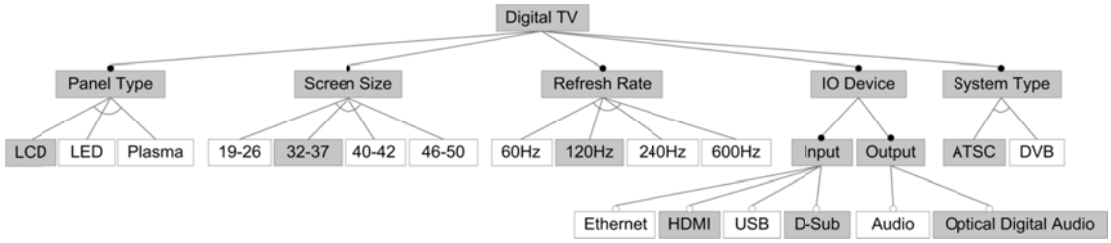


Figure 2. A configuration for LCD32_37

The goal is to provide an innovative approach in a feature modeling environment that 1) provides a simple way to specify how a feature model should be configured in a given context, 2) can automatically apply captured configuration rules to a partially configured feature model, and 3) does not require learning any additional languages, such as a model transformation language or metamodeling language.

The rest of the paper is organized as follows: Section 2 presents a motivating example to demonstrate the commonly used manipulation and analysis in feature model configuration; Section 3 explains the solution by introducing MTBD with several examples; Section 4 discusses the advantages of the approach, as well as the current limitations. Section 5 compares the related work, and Section 6 offers concluding remarks.

2. MOTIVATING EXAMPLE

As the context for a motivating example, Figure 1 shows a feature model for the configuration of televisions. This is a simplified version specifying the main features such as *Panel Type*, *Screen Size*, etc. The feature model is used to make selections on the features and define the desired configuration for various types of digital TVs. Figure 2 presents the configuration for *LCD32_37*. During the configuration process, some manipulation and analysis might be performed regularly.

Scenario 1 – Configuration Saving/Loading. A specific digital TV configuration consists of a unique set of feature selections. Making a configuration for *LED19_26*, for example, requires selecting different features for *Panel Type*, *Screen Size*, and *Refresh Rate* compared with *LCD32_37*. Saving a specific configuration and loading it in the base feature model is commonly needed in the model editor. The saved configuration can also represent partial configuration patterns, such as selecting all required features, or selecting a popular combination of *Screen Size* (e.g., *40-42*) and *Refresh Rate* (e.g., *120Hz*).

Scenario 2 – Automatic Constraint Checking. Several constraints in the digital TV product-line exist. For example, the *Panel Type-Plasma* can only apply a *Refresh Rate* at *600Hz*; the *LCD* with *Screen Size 32-37* cannot use *Refresh Rate* over *120Hz*. During the modeling process, users expect to conform to all positive and negative constraints. An automatic checking mechanism can be helpful in assessing configuration correctness.

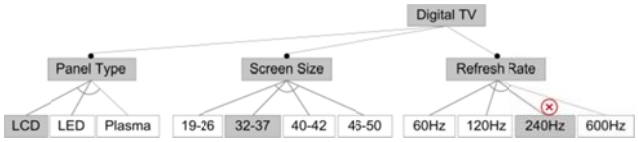


Figure 3. An incorrect configuration

Scenario 3 – Automatic Error Correction. After a constraint violation is found, corresponding actions should be performed to fix the error. For instance, in Figure 3, the *LCD* with a *Screen Size 32-37* is configured with a *Refresh Rate* at *240Hz*, which is over the normal range (i.e., *60Hz-120Hz*). The most direct approach to fix this error is to deselect the *240Hz* feature to remove the violation. An automatic error detection and correction procedure can improve the quality of the configurations.

3. MTBD AND FEATURE MODELS

Our end-user solution to automate feature model manipulation and analysis is to use a demonstration-based technique, Model Transformation By Demonstration (MTBD) [2], which is a new approach to implement model transformations. The goal of MTBD is to enable general users (e.g., domain experts) to specify reusable configuration tasks without knowing model transformation languages or metamodel definitions. Thus, instead of manually writing constraint rules or model transformation rules to specify an automated configuration process, users can specify these rules by demonstrating the specific feature configuration process (e.g., select features) on the feature model and offering feedback on the model to reflect the desired constraints. A recording and inference engine captures all feature selections and deselections, generalizes a user's intention in a configuration task, and generates a reusable transformation pattern automatically.

Each configuration task includes a precondition to ensure that the task is not applied in an incorrect context, and a transformation that will automatically select/deselect a subset of the feature in the model. After configuration knowledge is captured as a reusable task, it can be applied by different domain experts. For example, a new engineer can use a configuration task, captured by a senior engineer, to automatically configure a specific subset of features that he/she is unfamiliar with. Moreover, an engineer may use a configuration task captured by an expert from a different domain to automatically configure a portion of the feature model that is outside of his/her expertise. We present the main components and steps in MTBD in Section 3.1, followed by an illustration of the idea with several examples in Section 3.2.

3.1 Introduction to MTBD

Figure 4 is an overview of the MTBD approach, which was originally presented in [2]. To specify a model transformation, users first give a demonstration by directly editing a model instance (e.g., add a feature to a configuration) to simulate a transformation task. During the demonstration, an event listener monitors all the operations occurring in the feature model editor and collects the information for each operation in sequence. After the demonstration, the engine optimizes the recorded operations to eliminate any meaningless or useless actions.

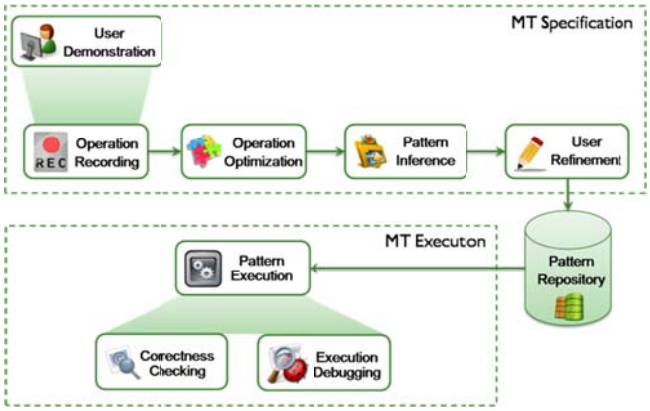


Figure 4. Overview of MTBD

With an optimized list of recorded operations, the transformation can be generalized and inferred. Because MTBD does not rely on a model transformation language, no specific transformation rules are generated. Instead, we generate a transformation pattern<*P*, *T*>, where *P* summarizes the precondition of a transformation (i.e., where a transformation applies), and *T* specifies the actions needed in a transformation (i.e., how a transformation is performed). For example, a precondition can specify a set of features that must be selected or quality attribute values that must be present for a specific automated configuration to be applied.

The initial pattern inferred is specific to the demonstration and may not be completely accurate since it is inferred from a demonstration that may depend on domain information not related to the features directly manipulated in the demonstration. Users are allowed to refine the inferred transformation by providing more feedback for the precondition of the desired transformation scenario from two perspectives – structure and attributes. For instance, users can restrict the precondition by selecting and confirming extra features or relationships between features in the feature model editor that must be included in the pattern (e.g., the *240Hz* feature must be a child of the *Refresh Rate* feature). A new type of operation (*Confirm Containment*) is implemented in the editor for this purpose. The refinement on the attributes can be realized by choosing the element in the demonstration and typing the specific conditions (e.g., only select a feature if the refresh rate is "*600Hz*"). The refined transformation pattern <*P'*, *T*> will be finalized and stored in the pattern repository for future use.

The final generated patterns can be executed on any feature model instances that satisfy the required preconditions. The execution starts with matching the precondition *P'* in a model instance and then carrying out the transformation actions (*T*) on the matched locations of the model. When the syntax or semantics are violated, the execution of each transformation action will be logged and the model instance correctness checking is performed after every execution. If a certain action violates the metamodel definition, all executed actions are undone and the whole transformation is cancelled. We are working on a debugging tool as part of MTBD to aid in identification of errors in the demonstration process.

3.2 Assisting Feature Configuration

To use MTBD to handle common feature model manipulation and analysis scenarios, we can follow the MTBD steps to demonstrate the process, refine the pattern, and then execute the pattern whenever needed.

Supporting Configuration Saving/Loading. As an example, after understanding how to configure the *LCD32_37*, we simply demonstrate the selection process on a base model, by setting the 15 features to be selected, and change the model shown in Figure 1 to the new configuration in Figure 2. The recording engine captures all 15 operations performed, and generalizes an initial transformation pattern, which contains a minimum precondition and a set of transformation actions. Figure 5 is an abstract representation of the generalized pattern. The precondition is minimum because it only specifies the minimum number of model elements needed to correctly execute all the transformation actions with enough operands. In this case, 15 different features are sufficient to guarantee the recorded actions.

The initial generalized pattern is not accurate because it lacks a specification of the relationships between all features, as well as the constraints on the name of each feature. In the user refinement step, users can provide more feedback to restrict the generalized precondition. The refinement on the structural precondition is given by using the *Confirm Containment* operation extended in the editor. Users simply choose the desired model element (e.g., the feature relationship between the two selected features) in the editor, and choose *Confirm Containment*, after which the chosen elements will be included in the precondition (see Figure 6).

The refinement on the attribute precondition is done through an attribute precondition dialog, which enables users to select any model elements in the current precondition, and specify a textual constraint on its attribute. For example, the root of the feature model must be "*Digital TV*," so we click on the name of the feature *f1*, and *type == "Digital TV."* Similarly, feature *f2* (i.e., *Panel Type*) uses the *XOR* for composition of its children, so it is also specified in the dialog. Figure 7 shows the final transformation pattern after refinement.

Precondition	Actions
	f1.selected=true f2.selected=true f3.selected=true … … f15.selected=true
f1 – f15: Feature	

Figure 5. The initial generalized transformation pattern

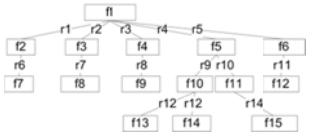
Precondition		Actions
		f1.selected=true f2.selected=true f3.selected=true … … f15.selected=true
f1-f15: Feature	r1-r15: Feature Relationship	

Figure 6. The transformation pattern with structural precondition

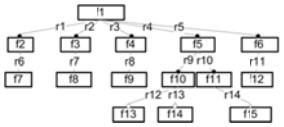
Precondition		Actions
		f1.selected=true f2.selected=true f3.selected=true … … f15.selected=true
f1-f15: Feature	r1-r15: Feature Relationship	
f1.name == "Digital TV" f1.ChildComposition == Required f2.name == "Panel Type" f2.ChildComposition == XOR … …		

Figure 7. The transformation pattern with attribute precondition

With the finalized pattern, the *LCD32_37* configuration can be loaded anytime by executing the pattern. The execution engine regards the precondition of the pattern as a graph with constraints, and traverses the model to carry out a graph matching. The actions will be executed on the match location, setting all the features as selected. Following the same approach, users can also demonstrate configuration on part of the feature model, or some selection patterns for future reuse.

Supporting Constraint Checking. Another challenge that the MTBD approach helps alleviate is ensuring that configurations do not violate domain best practices or rules that are not directly captured in the feature model. For example, although it may be possible to select a *Plasma* panel with a refresh rate lower than *600Hz*, it is not a typical configuration. In this case, the captured configuration tasks can aid engineers in checking these constraints external to the feature model.

From these examples, it can be seen that the precondition of the pattern serves as the matching criteria for the execution engine. In other words, a successful pattern match implies the satisfaction of the constraints. Therefore, MTBD can be applied to demonstrate the specification of the desired constraints and let the engine automatically check them. In the second example, we demonstrate the constraint corresponding to the requirement that a *Panel Type-Plasma* can only apply a *Refresh Rate* at *600Hz*.

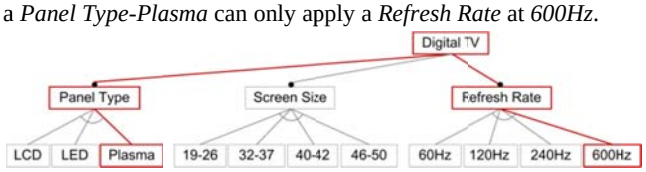


Figure 8. Confirming the involved features in the constraint

Specifying a constraint does not require regular editing operations. Users can perform *Confirm Containment* operations to reflect the desired structure of the rule (e.g., the *Plasma* feature and *600Hz* feature, as well as their parents and relationships, as illustrated in Figure 8). To further restrict the pattern and give more specific constraints, we provide attribute refinement on the precondition – both "*Plasma*" and "*600Hz*" are selected (i.e., both *selected* attributes are *true*), as shown in Figure 9.

Later, when the pattern is executed, a successful pattern match means that the specific constraint is satisfied in the current model. Similarly, negative constraints can be demonstrated (e.g., demonstrate the selected *LCD* with the *Screen Size 32-37* is configured with a selected *Refresh Rate240Hz*), and a successful pattern matching means the constraint is violated.

Supporting Error Correction. Based on constraint checking, error correction can be realized by demonstrating the correction operations. For example, if the situation in Figure 3 is present, users can demonstrate deselecting the 240Hz feature, and then provide the structural precondition and attribute precondition. Executing the finalized pattern (Figure 10) will match the location that violates the constraints and execute the actions to fix the violation.

Precondition		Actions
	f1-f5: Feature r1-r4: FeatureRelationship	N/A
	f1.name == "Digital TV" … … f5.name == "Plasma" f5.selected == true f6.name == "600Hz" f6.selected == true	

Figure 9. Finalized transformation pattern for constraint checking

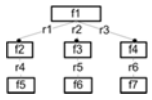
Precondition		Actions
	f1-f5: Feature r1-r4: FeatureRelationship	f7.selected = false
… … f7.name == "LCD"f7.selected == true f8.name == "32-37" f8.selected == true f8.name == "240Hz" f8.selected == true		

Figure 10. Finalized transformation pattern for error correction

4. DISCUSSION

MTBD has been implemented in GEMS (Generic Eclipse Modeling System) [7]. With a feature modeling language defined in GEMS, we are able to use MTBD to support the main feature model manipulation and analysis highlighted in this paper. Using MTBD, users are only involved in editing feature model instances to demonstrate the configurations and giving feedback on the constraints after the demonstration. All of the other tasks (i.e., optimization, inference, generation, execution, and correctness checking) are fully automated. Additionally, in the steps where users are involved, all the information exposed to users is at the concrete feature modeling level at the specific working domain, rather than at the generic metamodel level. The containment confirmation is simply realized by a one-click operation on the desired feature or relationship, and the extra precondition is given using the dialog where users can access all the elements in the precondition and type the constraints

directly. The generated patterns are invisible to users (Figure 5, 6, 7, 9, 10 are presented for the sake of explanation, which are not visible to users when using MTBD). Therefore, users are fully isolated from metamodel definitions and implementation details. Moreover, because no MTLs and tools are used in the implementation of MTBD, users do not need to know any constraint or model transformation languages.

In the current implementation, some limitations are still present that require further improvement. It is not convenient to express more generic constraints by demonstration and refinement. For example, if *LCD32_37* can only apply the maximum *Refresh Rate*, the maximum value cannot be reflected easily by demonstration or refinement. Furthermore, it would be more helpful to enable live transformation pattern matching and execution, rather than allowing users to execute the patterns manually. Users can be prompted with notifications about the potential constraint satisfaction/violation, or error correction. In addition, instead of storing the transformation patterns locally, using a remote repository to share their configuration and analysis patterns may improve knowledge reuse and communication. Finally, how to check the correctness of the demonstration, when applying multiple patterns to a feature model, and how to detect and resolve the conflicts among them are other essential issues.

5. RELATED WORK

MTLs are powerful tools to support feature model configurations. Both textual and graphical MTLs are applicable to these tasks, but they all share the challenges of a steep learning curve that requires knowledge of metamodel definitions. Similar to MTBD, Model Transformation By Example (MTBE) [4] is another approach to simplify the implementation of model transformations by inferring transformation rules from the given mappings. However, this approach focuses on transformation between different domains, so it is not appropriate for the feature model manipulation and analysis tasks that occur in the same domain.

Some intelligent approaches to automatically detect and fix configuration have been investigated by White et al. [5]. They focus on specific constraint checking and correction techniques, which calculate the minimum error fixing operations based on the input model and a set of constraints. Instead, we concentrate on more generic figure model configuration tasks and enabling end-users to easily specify their own constraints during the feature model editing process.

6. CONCLUSIONS

A number of manipulation and analyses are frequently performed when configuring feature models. This paper described an application of Model Transformation by Demonstration to simplify the specification of such manipulation and analysis in feature models, so that general end-users can convert their knowledge on configuration and constraints into transformation patterns, which can be reused in any feature model to automate similar processes.

7. ACKNOWLEDGMENTS

This work is supported by NSF CAREER award CCF-1052616.

8. REFERENCES

[1] Metzger, A., Pohl, K., Heymans, P., Schobbens, P., & Saval, G. Disambiguating the Documentation of Variability in Software Product Lines: A separation of concerns, formalization and automated analysis. *Requirements Engineering Conference*, New Delhi, India, Oct. 2007, 243–253.

[2] Sun, Y., White, J., & Gray, J. Model Transformation by Demonstration. *Model Driven Engineering Languages and Systems*, Denver, CO, Oct. 2009, 712-726.

[3] Sendall, S. & Kozaczynski, W. Model Transformation - The Heart and Soul of Model-Driven Software Development. *IEEE Software*, vol. 20, no. 5, 2003, 42–45.

[4] Balogh, Z., & Varró, D. Model Transformation by Example Using Inductive Logic Programming. *Software and Systems Modeling*. vol. 8, no. 3, 2009, 347-364.

[5] White, J., Schmidt, D., Benavides, D., Trinidad, P., & Ruiz-Cortes, A. Automated Diagnosis of Product Line Configuration Errors in Feature Models. *International Software Product Line Conference*, Limerick, Ireland, Sep. 2008, 225–234.

[6] Batory, D., Benavides, D., & Ruiz-Cortes, A. Automated Analysis of Feature Models: Challenges Ahead. *Comm. of the ACM*, vol. 49, no. 12, 2006, 45-47.

[7] GEMS, http://www.eclipse.org/gmt/gems/