Supporting Model Maintenance in Component-based Product Lines

Markus Jahn Rick Rabiser Paul Grünbacher Markus Löberbauer Reinhard Wolfinger Hanspeter Mössenböck
Christian Doppler Laboratory for Automated Software Engineering
Johannes Kepler University Linz, Austria
markus.jahn@jku.at

***Abstract*—Software product line engineering aims at increasing software quality and development productivity by mastering the variability of large software systems. Models are frequently used to define the reusable assets and the restrictions regarding asset composition in different products. However, product line engineering is challenged by evolution. Reusable assets such as software components need to be adapted to meet new customer or market requirements as well as technological needs. In this paper we present an approach that supports the maintenance of product line models by checking their consistency with the available components. We describe algorithms and heuristics for creating and updating the models defining a product line's features and components. We evaluate our approach using realistic change scenarios from a product line of time recorder applications.**

***Index Terms*—Components, product lines, model-based development, evolution.**

I. Introduction

Software product line engineering (SPLE) leverages systematic reuse to improve quality and development productivity and to decrease time-to-market and costs in software development [1, 2]. In domain engineering, reusable assets such as requirements, components, or test cases are developed. The commonalities and variability of these assets are typically defined in variability models [3]. In application engineering, products for a particular customer or market are then derived by exploiting the variability.

SPLE is a process involving diverse roles such as sales people, customers, product managers, architects, developers, or testers. The involvement of these different roles means that two views on the variability of software systems are commonly distinguished [4]: the external view addresses variability as visible to customers and the market, i.e., the features available in different products of the product line; the internal view covers the solution components, their dependencies, and the actual implementation of the variability. Numerous approaches are available in product line engineering and component-based software development to realize these two views. Examples include variability modeling [3], architecture description languages [5], or component technologies [6].

Most product lines are developed and used for many years and managing their evolution is thus critical for success [7]. For instance, developers modify the solution space by adding, changing, or deleting reusable assets to realize new features required by customers. Similarly, product managers adapt the problem space by defining new features and their variability. Product line models thus have to be maintained frequently to reflect these changes and to ensure their consistency with the underlying assets. This is a challenging, costly, and error-prone process [8, 9]. Despite some progress it remains particularly challenging to understand the impacts of technical changes to product line models.

We present an approach that aims at reducing the maintenance effort of product line models. Our approach distinguishes three different views in product line models representing the perspectives of developers, product managers, and customers (cf. Figure 1). The development view describes the reusable elements and their dependencies (e.g., restrictions defining component composition). The product management view defines the features of the product line. The customer view defines the configuration options made available to customers selecting between different product line members.

We claim two contributions supporting the evolution of component-based product lines: i) We present an automated approach to generate and update the development view and to check the consistency between the development view and the product management and customer views. ii) We propose a set of algorithms and heuristics to support the maintenance of the product management and customer views on the product line model.

We use the DOPLER product line approach [10] and the Plux component infrastructure [11] to implement our approach. In in earlier paper [12] we described how the DOPLER tool suite can be integrated with the Plux infrastructure. However, we did not address evolution. In this paper we present an approach to facilitate model maintenance during the evolution of the product line.

The remainder of the paper is structured as follows: Section II presents our approach and summarizes DOPLER [10] and Plux [11]. Using an illustrative example we explain the structure of the product line models in Section III and discuss the implementation of algorithms and heuristics for model analysis in Section IV. Section V uses a realistic development scenario to show how the maintainenance of product line models can be supported with our approach. In Section VI we compare our approach to existing literature. We round out the paper with a summary and an outlook on future work.

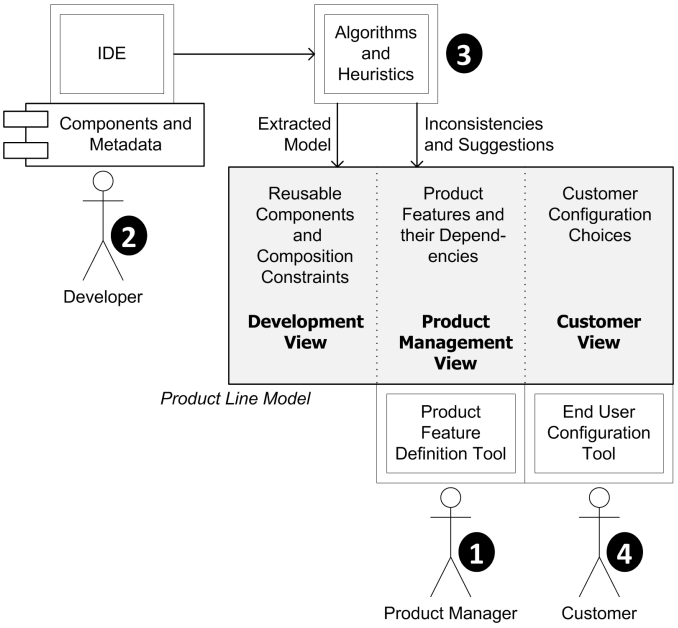

Figure 1. Supporting the maintenance of a product line model comprising a development, product management, and customer view.

II. Approach

Figure 1 gives an overview of our approach. The product line model provides three different views taking into account the perspectives of the developer, the product manager, and the customer.

The *developer* uses an IDE to develop and maintain the repository of software components with metadata describing interfaces and contracts. Our approach uses model extraction to reflect the changes to the component repository in the development view of the model defining the reusable elements and their composition constraints.

The *product manager* uses a modeling tool to maintain the product management view, i.e., the available features, their variability, and dependencies. Our approach supports the product manager with algorithms and heuristics suggesting changes to the view depending on the changes made by the developers.

The *customer* uses an end-user configuration tool to derive a product from the product line by selecting the features matching his requirements. The configuration choices presented to the customer depend on the available features and their variability.

The typical change scenario shown in Figure 1 is as follows: (1) The product manager plans and defines a new product feature. (2) The developer implements the feature using the IDE. Typical activities involve adding new components, splitting and refactoring components, or refactoring component interfaces. (3) The tool automatically updates the product line model to minimize manual model maintenance. It identifies potential inconsistencies and suggests updates to the modeler. (4) The end user derives a product based on the new product line model by making configuration decisions to select features. Selecting features leads to the automatic selection of the components realizing these features for the product.

A. DOPLER Product Line Model

The product line model is developed and maintained using the tool-supported approach DOPLER (**D**ecision-**O**riented **P**roduct **L**ine **E**ngineering for effective **R**euse) [10]. DOPLER is flexible and extensible and can be tailored to different organizations' needs [10].

The *customer configuration choices* are based on decision models [3] defining the available configuration options. Decisions distinguish the different members of a product line and allow presenting variability during product derivation. Important attributes of decisions in DOPLER are a unique id, a question that is asked to a user during product derivation, and a decision type to define the possible answers (boolean, enumeration, string, or number). Another important attribute of decisions is a default value suggesting a predefined answer to the user. Decisions can depend on other decisions hierarchically (if they need to be made before other decisions) or logically (if they affect other decisions).

The *reusable components and composition constraints* are defined in DOPLER using assets models. Assets represent the software components in the solution space and can depend on each other functionally or structurally. *Inclusion conditions* link assets to decisions and define which assets are present in a derived product depending on the values of decisions. Asset attributes can also depend on answers to decisions to enable component parameterization.

The *product features and their dependencies* are also defined using assets. A feature can be seen as a unit of functionality that is based on several components. Relationships from feature assets to component assets and from features to decisions allow selecting and parameterizing components based on the users' configuration choices.

B. Plux Developer IDE and Component Repository

The developer uses the Plux composition infrastructure [11, 13, 14]. Applications developed with Plux comprise fine-grained components which are connected by the composition infrastructure using a plug-and-play approach. Users can adapt the application dynamically to align it with the working situation at hand by swapping sets of components at runtime. With plug-and-play composition, this can be done without configuration or programming effort [11]. Plux differs from other plug-in systems [15] such as OSGi (http://www.osgi.org) or Eclipse (http://www.eclipse.org) by providing a built-in composer which maintains a global composition state, using an event-based programming model, and an exchangeable plug-in discovery mechanism. The composer replaces programmatic composition with automatic composition. Programmatic composition, as for example in Eclipse, means that the host component has to query a component registry and must create and integrate its contributors itself. Automatic composition, as in Plux, means that the components just declare their requirements and provisions using metadata. The composer then uses these metadata to match requirements and provisions and connects matching components. At any time, Plux maintains a global composition state, i.e., it stores which host components use

which contributor components. Host components retrieve their contributors from the composition state. Optionally, components can react to events sent by the composer, e.g., if configuration changes should be shown in the user interface immediately. Component discovery is the process of detecting components and extracting their metadata. The discovery mechanism is a plug-in itself, which makes it replaceable.

Our model extraction approach automatically creates the development view (i.e., a DOPLER asset model) based on Plux metadata as we will describe in Section III.

C. DOPLER Product Feature Definition Tool

The product manager uses DOPLER's variability model editor [10] to define and maintain the end-user features and their dependencies to the Plux components realizing them. The variability model editor allows creating and defining product line assets, their attributes, and relationships for all asset types defined in a domain-specific meta-model (cf. Figure 3). The variability of the features is defined using decisions including questions explaining the configuration choices to end users.

Our approach uses heuristics to suggest new features by analyzing the changes made by developers to the Plux component repository as we will show in Section IV.

D. DOPLER End-User Configuration Tool

The customer derives a product using an end-user configuration tool: DOPLER's ConfigurationWizard [16, 17] provides capabilities for product customization based on DOPLER variability models. The output of ConfigurationWizard is a product derived from the product line. End users can make configuration choices by answering questions. The tool then triggers generators to compose the desired product automatically based on the information in the underlying model.

In this paper we focus on the algorithms and heuristics to support model maintenance and not on the end-user view on variability as provided by the ConfigurationWizard. We refer the reader to [16, 17] for details on product derivation and configuration with DOPLER.

III. Product Line Model

We describe the key concepts of Plux and present the meta-model we defined in DOPLER to support the three views of developers, product managers, and customers. We show how our approach supports extracting a product line model from the Plux repository. As a running example, we use a time recorder system for recording working hours. Details about this system and its maintenance are discussed in Section V.

A. Model Elements

Plux uses the metaphor of extensions, slots and plugs (cf. Figure 2). An *extension* is a component that provides services to other extensions and uses services provided by other extensions. If an extension wants to use a service of some other extension it declares a *slot*. Such an extension is called a *host*. If an extension wants to provide a service to other extensions it declares a *plug*. Such an extension is called a *contributor*. Several related extensions can be packaged as a *plug-in*, which is a dynamic link library (dll) file that can be deployed and loaded separately. Plug-ins can have dependencies to other dll files without extensions. Such dll files are called *libraries*.

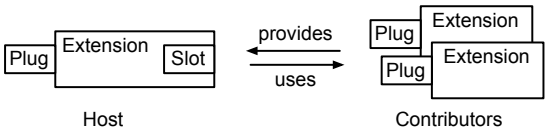

Figure 2. Plux metadata for extensions with slots and plugs.

Slots and plugs are identified by their names. A plug matches a slot if their names match. If so, the plug can be connected to the slot. A slot represents an interface, which has to be implemented by a matching plug. The interface is specified in a *slot definition* with a unique name as well as optional parameters that are provided by the contributors and retrieved by the hosts. The names of slots and plugs refer to their respective slot definitions. Several related slot definitions can be packaged as a *contract*, which is a dll file.

Based on the DOPLER meta-meta-model [10] we have developed a domain-specific meta-model for Plux-based systems (cf. Figure 3). The meta-model defines different asset types needed for Plux-based systems together with attributes and possible dependencies. We explain these asset types and dependencies using the time recorder example for illustration.

Feature. A Plux feature defines a set of functionalities visible to users. It requires one or more plug-ins to implement these functionalities (cf. dependency *Feature requires Plugin*). For example, there might be a mandatory *Base* feature referring to all plug-ins that constitute the basic product that can be derived from the product line. The *MobileSync* feature on the other hand is optional and refers to a plug-in with mobile synchronization capabilities.

Plug-in. A Plux plug-in packages several related extensions. It is developed based on existing libraries. For instance, basic presentation capabilities are provided by the *Plux.Presentation* library (cf. dependency *Plugin requires Library*). A plug-in also adheres to a certain contract. For example, a basic definition of how presentation in a GUI can be implemented is defined by the *Plux.Presentation* contract (cf. dependency *Plugin requires Contract*). Plug-ins consist of extensions meaning that the basic functionality of the system is extended in pre-defined ways (cf. dependency *Plugin has Extension*). For example, the *TimeRecorder* plug-in has a *Statistics* extension for adding arbitrary statistics capabilities.

Library. Libraries in Plux capture basic capabilities that can be used by multiple plug-ins. An example is a library for developing GUIs. Libraries can require other libraries (cf. dependency *Library requires Library*) and can require contracts (cf. dependency *Library requires Contract*) defining how basic capabilities can be implemented.

Contract. The names of slots and plugs refer to their respective slot definitions. Several related slot definitions are packaged as a contract (cf. dependency *Contract defines Slot*).

Extension. An extension is a component (i.e., a class) that

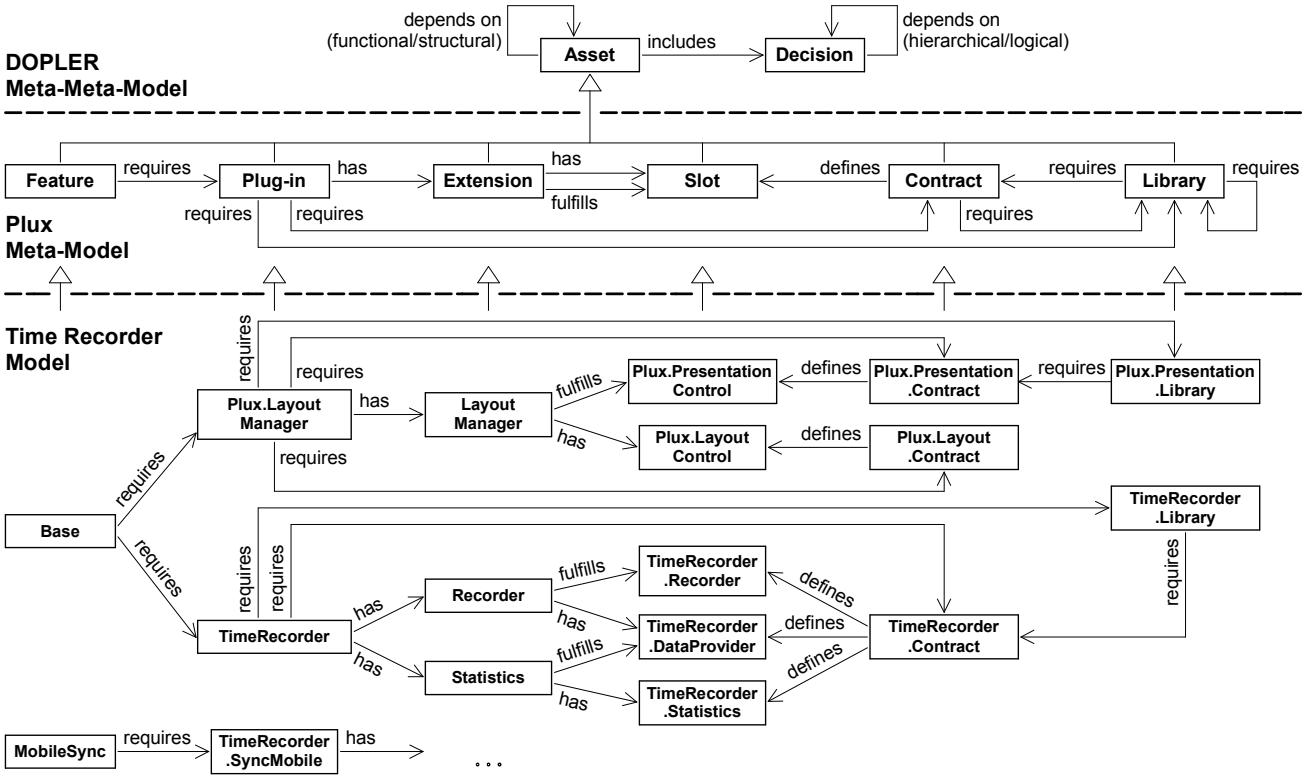

Figure 3. DOPLER meta-meta-model, Plux meta-model, and (partial) time recorder asset model.

provides services to other extensions and uses services provided by other extensions. If an extension wants to use a service of some other extension it declares a slot. An extension can thus fulfill (cf. dependency *Extension fulfills Slot*) and have a slot (cf. dependency *Extension has Slot*). These two relationships represent the metaphor of extensions, slots and plugs: if an extension fulfills a slot, it has a plug fitting into this slot; if an extension has a slot, extensions with a fitting plug can be connected.

Slot. A slot represents an interface, which has to be implemented by every extension that has a matching plug.

Modeling a Plux-based system based on this meta-model thus allows linking the technical views of developers with the views of product managers and customers. Implementation details of a system implemented using Plux are not needed in the DOPLER product line model. If customers select a feature the model allows determining the plug-ins, libraries, contracts, extensions, and slots required for a derived product without further intervention. This is sufficent for the Plux composition infrastructure to deploy a product. For instance, two decisions might exist for the time recorder example shown in Figure 3: "Do you want to buy a time recorder?" and "Do you need to sync with mobile devices?". Answering the first decision would select the Base feature while answering the second decision would select the MobileSync feature. The model also allows automated checks supporting Plux developers as we will describe next.

B. Model Extraction and Tool Integration

In Plux, system capabilities are implemented by extensions. Extensions, as well as their slots and plugs, are described by their metadata. If a program evolves, the metadata of the extensions likely change. Integrating Plux and DOPLER thus relies on exporting these metadata changes from Plux and importing them into DOPLER.

We thus developed a tool that exports the content of the Plux metadata into an XML file containing the information required by DOPLER, i.e., the plug-ins, libraries, contracts, extensions, slots and their dependencies. When the Plux repository changes, the variability model is updated by re-importing the new version of the XML file. New elements and dependencies are automatically included; renamed and deleted elements are marked for manual resolution.

The way for providing metadata in Plux is customizable. The default mechanism extracts metadata from .Net attributes in Plux plug-in and contract files [18]. Plux has the following attributes: The *SlotDefinition* attribute to tag an interface as a slot definition, the *Extension* attribute to tag a class that implements an extension, the *Slot* attribute to declare requirements in hosts, the *Plug* attribute to declare provisions in contributors, the *ParamDefinition* attribute to declare required parameters in slot definitions, and the *Param* attribute to specify provided parameter values in contributors.

For instance, the frontend of the time recorder works with time stamps that hold begin and end times as well as task descriptions. The data source for time stamps is implemented

as a contributor that plugs into the frontend. Listing 1 shows the definition of the *DataSource* slot, the contributor *Store* (which acts as a data source and therefore has a *DataSource* plug), and the *Frontend* host (which has a *DataSource* slot and also an *Application* plug that fits into the *Application* slot of the Plux core).

Listing 1. Plux metadata of the simplified time recorder.

```
[SlotDefinition("DataSource")]
interface IDataSource { ... }

[Extension]
[Plug("DataSource")]
class Store : IDataSource { ... }

[Extension]
[Plug("Application")]
[Slot("DataSource")]
class Frontend : IApplication { ... }
```

To complete the example, we compile the slot definition to a contract dll file, and the classes *Frontend* and *Store* to plug-in dll files. From these files, Plux composes the application as shown in Figure 4. In doing so, Plux discovers extensions from the plug-ins and composes the program from them by connecting matching slots and plugs. The plug-ins and contracts are typically stored in a directory of the file system.

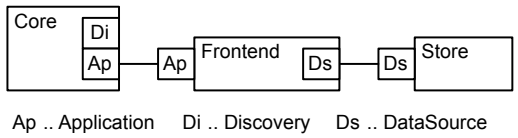

Figure 4. Schematic composition state of the time recorder.

Figure 5 explains the subsystems of Plux and how they interact. The *discoverer* ensures that at any time the type store contains the metadata of extensions and slot definitions from the plug-in directory. When the discoverer detects an addition, it extracts the metadata from the dll file and adds them to the *type store*. The type store maintains the type metadata of slot definitions and extensions which are available for composition and notifies the *composer* about changes. The composer assembles a program by matching requirements and provisions as declared in the metadata and stores the composition state in the *instance store*.

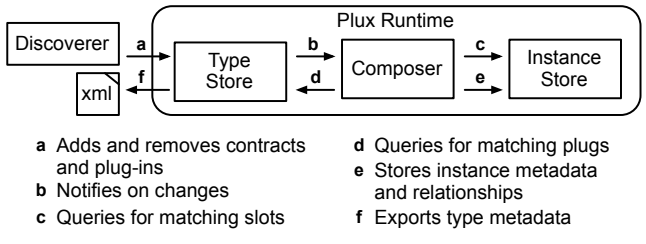

Figure 5. Architecture of the Plux composition infrastructure with type metadata export support.

IV. Model Analysis: Detecting Inconsistencies and Creating Suggestions

We developed several algorithms and heuristics to support model maintenance by detecting inconsistencies and suggesting model updates after changes to the Plux repository or the variability model.

A. Inconsistencies

Five different types of problems can be found by analyzing product line models after changes:

A *dead asset (DA)* will never be included in a derived product. This can happen if it is neither related to a decision nor to any other non-dead asset which would require its inclusion. The algorithm thus checks for each feature, plug-in, slot, extension, contract, and library whether it is related to a decision or whether at least one related non-dead asset exists. For each identified dead asset a warning is shown.

An *unused extension (UE)* does neither provide nor fulfill a slot and is hence not connected to the overall system. The algorithm checks for each asset of type extension whether it has at least one 'has' or 'fulfills' relationship. For each identified unused extension the algorithm produces a warning for the modeler to consider removing it.

An *unused slot (US)* is not used by any extension and is hence not used by the system. It might be existing on purpose for future extensions, but might as well be useless. As shown in Listing 2, for each asset of type slot, the algorithm visits all assets of type extension that have a relation to this slot. If none are found, the slot is potentially useless and the algorithm produces a warning to consider removing the slot.

Listing 2. Algorithm for detecting unused slots in the product line.

```
// (US) unused slot algorithm
// Detects slots that are not used by any extension
if (asset.isOfType(Slot))
  for (Asset a : asset.requiringAssets)
    if (a.isOfType(Extension)) return
print("Slot " + asset + " is not used by any"
   + " Extension; consider removing")
```

Feature updates (FU) are necessary to preserve consistency between the problem space and the solution space. When plug-ins are renamed in the system and the variability model is updated, the original plug-ins are kept in the model and marked as candidate removes. This is done because system developers might rename an existing plug-in and then create a new plug-in with the original name of the just renamed plug-in. Manual intervention is required in such cases. When a feature is related to a plug-in that is a candidate remove, the modeler has to relate the feature to another existing plug-in.

Features without plug-in (FWOP) are the result of removing plug-ins from the system without considering the features they realize. The FWOP algorithm checks for each feature whether it requires at least one plug-in. If not, the algorithm produces a warning.

B. Suggestions

We provide a heuristic to suggest features to the product manager based on the similarity of plug-ins (cf. Listing 3) to

further support model maintenance. If two plug-ins have similar characteristics and one of them is required by a feature and the other is not, the algorithm suggests to either add the similar plug-in to the existing feature or to create a new feature for it. The algorithm compares each plug-in $p1$ with all other plug-ins $p2$. $p2$ is considered as potentially similar with $p1$ if: (i) it has at least the same number of extensions as $p1$ and (ii) the number of slots fulfilled by the extensions of $p2$ is equal or greater than the number of slots fulfilled by the extensions of $p1$. Depending on $THRESHOLD$ $p1$ and $p2$ are regarded as similar.

Listing 3. Heuristic for computing the similarity of plug-ins and for suggesting feature candidates.

```
void printFeatureCandidates(List<Plugin> allPlugins)
  Map<Plugin, List<Plugin>> candidates =
     getSimilarPlugins(allPlugins)
  for (Plugin p1 : candidates.keySet())
    for (Plugin p2 : candidates.get(p1))
      if (!p2.isRequiredByAFeature())
        if (computeSimilarity(p1, p2) > THRESHOLD)
          print("Plugin " + p2 + " is similar to plugin "
             + p1 + " but not required by any feature."
             + " Consider adding it to a feature (e.g., "
             + getFeatureThatRequiresPlugin(p1)
             + ") or creating a new feature.")

Map<Plugin, List<Plugin>> getSimilarPlugins(
    List<Plugin> allPlugins)
  Map<Plugin, List<Plugin>> candidates = new ...
  for (Plugin p1 : allPlugins)
    for (Plugin p2 : allPlugins)
      if (p1 != p2
          && p2.extensions.size >= p1.extensions.size
          && countSlots(p2) >= countSlots(p1))
        candidates.get(p1).add(p2)
  return candidates

int countSlots(Plugin p)
  int count = 0;
  for (Extension e : p.extensions)
    count += e.slots.size
  return count

float computeSimilarity(Plugin p1, Plugin p2)
  int equalSlots = 0, totalSlots = 0
  for (Extension e1 : p1.extensions)
    for (Extension e2 : p2.extensions)
      for (Slot s1 : e1.slots)
        for (Slot s2 : e2.slots)
          if (s1.name == s2.name) equalSlots++;
          totalSlots++;
  return equalSlots / totalSlots
```

V. Evaluation

We use the time recorder system as the example for our evaluation. The time recorder provides features for recording and evaluating working hours. Users can log the start time, the end time, the related project, and a work description. The system provides a desktop frontend, a mobile application, or a dedicated hardware recorder. Furthermore, users can extend the time recorder with statistics extensions to analyze the recorded data. The system has been implemented using Plux as a set of extensions. The user interface is separated from the business logic, and the data are stored in a common generic data model.

To evaluate our approach, we performed the evolution of the time recorder system in 12 evolution scenarios. Our main research goal was to find out how well our approach supports developers in these basic evolution scenarios and how it helps to reduce model maintenance.

We started with the initial development of the time recorder system and in each scenario we added, removed, and refactored plug-ins, extension, slots, libraries, and contracts. The evolution scenarios reflect the evolution of the time recorder system as it has really been performed over time. However, evolution was simulated for the purpose of this evaluation.

A. Evaluation Steps

In our evaluation we performed the following research steps:

Developing and maintaining the system. We developed and evolved the time recorder system according to the 12 evolution scenarios.

Generating DOPLER model from Plux type store. We used our tool to automatically create a DOPLER model from the Plux type store via the XML export/import. Initially a new model was created. In all subsequent evolution scenarios the DOPLER model was updated after the changes.

Resolving warnings and implementing suggestions. After each evolution scenario, for the created DOPLER model, our tool detected inconsistencies and/or suggested creating features. We analyzed these suggestions and, where relevant, manually made model changes and created features accordingly.

Defining feature variability. Features are included in DOPLER based on making decisions. Whenever a new feature was added to a DOPLER model we added decisions to allow selecting the feature in product derivation.

Determining the success of the approach. Based on the results of every evolution scenario, we analyzed how well our approach supported model maintenance.

B. Evolution Scenarios

For each evolution scenario we describe the changes in the Plux component repository as well as the warnings and suggestions provided by our tool (for a summary of the results see Table I). For the similar evolution scenarios 1-6, we aggregated the description and results. Figure 6 shows our tool support reporting warnings and heuristics suggestions for scenario 7.

Change Scenario 0 – Initial development. The initial development led to five plug-ins, six libraries, three contracts, ten extensions, and eight slots, which were the input for creating the initial DOPLER model. The tool provided several dead asset warnings as expected, because the new assets were not yet included by any decisions, which still had to be defined.

Change Scenario 1-6 – Growing phase: New plug-ins and extensions, reuse of existing slots. Over these initial six evolution scenarios, six new plug-ins were developed including one extension each to provide a graphical viewer, a project recorder, a stamp note editor, a backup tool, a hardware time recorder, and support for synchronization with mobile phones. During this phase again several dead asset warnings were created for the new plug-ins. In addition, our heuristics provided several suggestions. For instance, the extension of the project recorder plug-in has and fulfills partly the same slots as the graphical viewer. Thus our heuristic suggested to either add the project recorder to the same feature as the graphical viewer or to create a new feature with the project recorder.

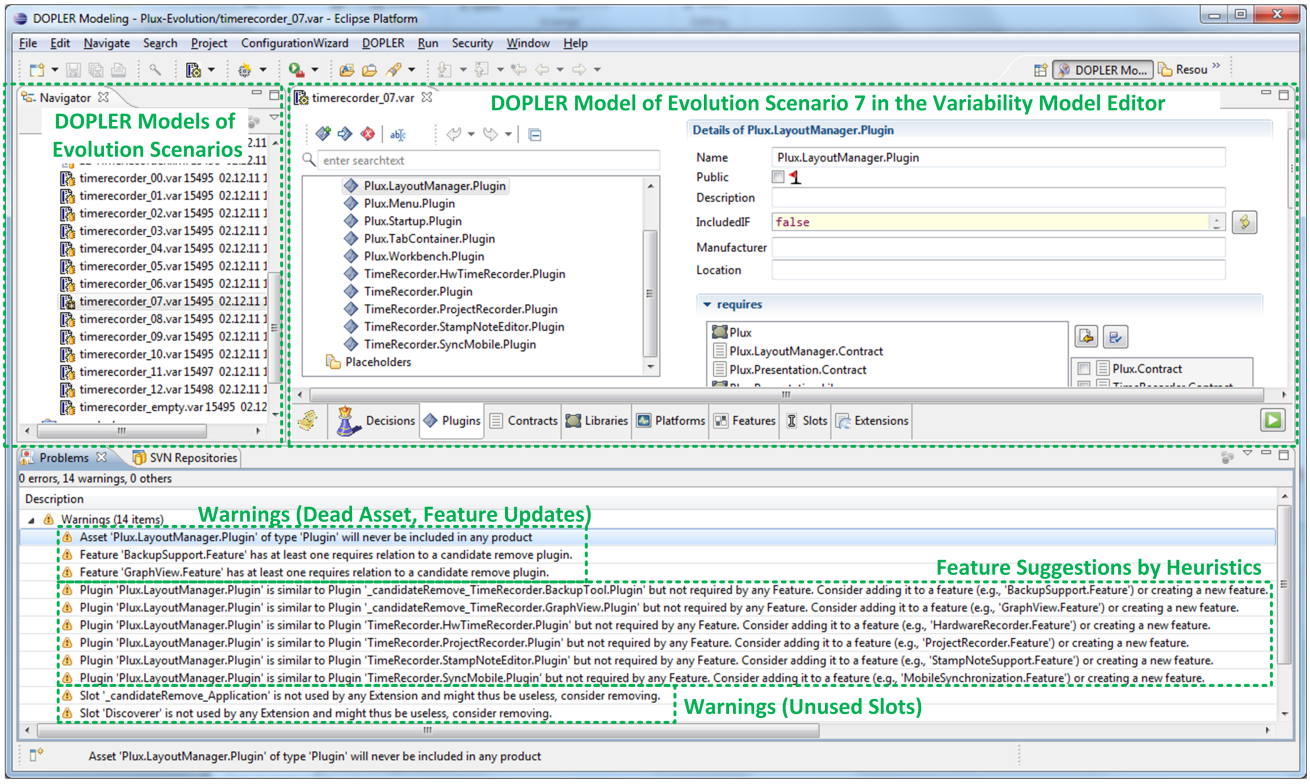

Figure 6. Warnings and feature suggestions for evolution scenario 7 in the DOPLER modeling tool.

Change Scenario 7 – New contracts, new slots, new/deleted extensions, new/deleted plug-ins. Scenario 7 involved major refactorings, i.e., a new contract (+1), new slots (+4), new extensions (+5), and a new plug-in (+1) were defined but also existing plug-ins (-2) and extensions (-4) were deleted. Our tool detected all renames (marking the original elements as candidates for removal and correctly recognizing the 'new' names) as expected. It also displayed dead asset warnings for all new elements, feature update warnings for renamed features, and feature without plug-ins warnings for removed plug-ins. In total four suggestions were provided as shown in Table I and Figure 6.

Change Scenario 8 – More generic notes plug-in instead of stamp note editor; project recorder plug-in split. In this scenario the main idea was to abstract from the rather specific stamp note editor and implement a more generic notes solution. Furthermore, the project recorder plug-in was split into a recorder control and the actual recorder. This led to one added plug-in and one removed plug-in; one added contract; two added extensions; one removed extension; and one added slot. The tool showed the expected dead asset warnings for the new plug-in and a feature without plug-in warning for the *StampNoteSupport* feature (as the stamp note editor plug-in was removed).

Change Scenario 9 – SyncMobile now using other slots/extensions (no new elements, just refactoring). In this scenario the support for mobile synchronization was refactored to use other slots and extensions. In the course of these refactorings, one plug-in and one extension were renamed. No warnings or suggestions were detected as only dependencies were adapted and no elements were introduced or renamed.

Change Scenario 10 – Hardware recorder refactoring. This scenario involved renaming the string 'Hw' in a plug-in and in an extension to 'Hardware'. Also, the slot of the extension was changed from model to recorder (an already existing slot). The tool showed the expected warning indicating the unused slot, as the extension was the only one using the model slot. It also showed a dead asset warning for the renamed plug-in and a feature without plug-in warning.

Change Scenario 11 – New plug-in and extension for payroll integration; reuse of existing slots. In this scenario the unused model slot was removed and a new plug-in *PayrollIntegration* was added together with the extension *PayrollIntegration* which has the slot *DataProvider* and fulfills the (existing) slot *Control*. The approach reported a dead asset warning for the new plug-in as well as two suggestions for features to add the new plug-in to.

Change Scenario 12 – Discoverer extracted from existing plug-in; new plug-in FilesystemDetector including slot and extension. This scenario involved adding two new plug-ins, one contract, two extensions, and two slots. The tool showed expected dead asset warnings for the two new plug-ins and the new contract.

C. Results

Table I summarizes for each scenario the warnings found after creating the variability model (using our checks) and the

TABLE I
EVALUATION RESULTS (WARNINGS, SUGGESTIONS, MANUAL ACTIONS, AND REMAINING PROBLEMS) FOR THE EVOLUTION SCENARIOS 0-12.

Scenario	Warnings	Suggestions	Manual Actions	Remaining Problems
0	14 DA Warnings (without features, no assets will be included)	no features -> no suggestion	defined a Base Feature for all new plug-ins	none
1	1 DA Warning for Plug-in GraphView	no suggestion	defined Feature GraphView	none
2	1 DA Warning for Plug-in ProjectRecorder	1 suggestion (similar to Plug-in GraphView)	defined Feature ProjectRecorder	none
3	1 DA Warning for Plug-in StampNoteEditor	2 suggestions (similar to Plug-ins GraphView and ProjectRecorder)	defined Feature StampNoteSupport	none
4	1 DA Warning for Plug-in BackupTool	3 suggestions (similar to Plug-ins GraphView, ProjectRecorder and StampNoteEditor)	defined feature BackupSupport	none
5	1 DA Warning for Plug-in HwTimeRecorder	4 suggestions (similar to Plug-ins GraphView, ProjectRecorder, StampNoteEditor and BackupTool)	defined Feature HardwareRecorder	none
6	1 DA Warning for Plug-in SyncMobile	5 suggestions (similar to Plug-ins GraphView, ProjectRecorder, StampNoteEditor, BackupTool and HwTimeRecorder)	defined Feature MobileSynchronization	none
7	1 DA Warning for Plug-in LayoutManager; 2 FWOP Warnings: 'Features BackupSupport and GraphView have at least one requires relation to a candidate remove Plug-in'	4 suggestions (similar to Plug-ins HwTimeRecorder, ProjectRecorder, StampNoteEditor, and SyncMobile)	removed Features BackupSupport and GraphView; added new Plug-in LayoutManager to the Base feature; removed candidate removes	none
8	1 DA Warning for Plug-in StampNoteSupport; 1 FWOP Warning: 'Feature StampNoteSupport has at least one requires relation to a candidate remove plug-in'	no suggestions	renamed Feature StampNoteSupport to NotesSupport and added renamed Plug-in Notes; removed candidate removes	none
9	none	no suggestions	none	none
10	1 DA Warning for Plug-in HardwareRecorder; 1 FWOP Warning: 'Feature HardwareRecorder has at least one requires relation to a candidate remove plug-in'; 1 US Warning: 'Slot Model is not used by any Extension and might thus be unused'	no suggestions	added renamed HardwareRecorder Plug-in to Feature HardwareRecorder; removed candidate removes	none
11	1 DA Warning for Plug-in PayrollIntegration	2 suggestions (similar to Plug-ins LayoutManager and HardwareRecorder)	added a new Feature PayrollIntegration with the new plug-in	suggestion to add plug-in PayrollIntegration to HardwareRecorder Feature does not make sense semantically
12	3 DA Warnings for Contract Discoverer, Plug-in FileSystemDetector, and Plug-in Discoverer	no suggestions	added Discoverer Plug-in to Feature Base and defined new Feature DynamicPluginUpdateSupport with new Plug-in FileSystemDetector	approach did not recognize that in this scenario an existing plug-in was split

suggestions made by the heuristics, the necessary (manual) actions to fix the warnings, and the remaining problems which could not be automatically solved and were not identified via a warning.

For scenarios 0-6, 8, 9, and 10 the results exactly matched our expectations. For scenario 7 results matched or even exceeded our expectations: everything was updated correctly and only one dead asset warning and two FWOP warnings remained. Four useful suggestions were made by our heuristics which greatly helped with manual model maintenance. For scenario 11, the results also were better than expected: two suggestions were made by our heuristic. However, the suggestion to add plug-in *PayrollIntegration* to the *HardwareRecorder* feature does semantically not make sense – an unresolved issue we have to address in future work. For scenario 12, except for the expected dead asset warnings, the heuristic did not recognize the fact that one plug-in was split into two plug-ins. This issue will also be resolved in future work.

VI. RELATED WORK

Our work is related with research on the co-evolution of models and product lines as well as work on consistency checking.

The idea to provide different views on the product line model is based on existing research on perspectives and viewpoints.

Co-evolution of models and product lines. Managing evolution is success-critical in model-based product line approaches to ensure consistency after changes to meta-models, models, and development artifacts. Similar to existing work on product line evolution [19–21] our approach aims to avoid product line erosion and deviation from the product line model. Deng et al. [22] describe a model-driven product line approach that explicitly focuses on the issue of domain evolution and product line architectures. Mende et al. [23] present tool support for the evolution of software product lines based on identifying code that might be moved from products to the product line level. Dhungana et al. [24] describe an approach supporting the definition of model fragments to simplify the evolution of large-scale systems. This work nicely complements the presented approach regarding scalability to very large systems. Similar to Murta et al. [8] we try to ensure consistency of architecture models to implementation focusing on evolution. However, our approach also addresses variability and problem space artifacts.

Consistency checking. Numerous approaches have been developed to identify inconsistencies between arbitrary artifacts in software development. Nentwich et al. [25] present a consistency checking approach for arbitrary distributed software engineering documents encoded in XML. Egyed [26] presents an incremental approach for evaluating consistency rules after changes to arbitrary models. Blanc et al. [27] focus on structural inconsistencies between different models in large-scale industrial software systems. Similar to Egyed et al. they use an event-driven approach which enables incremental evaluation of constraints. Consistency checking is also receiving a lot of attention in product line engineering research. For example, Czarnecki and Pietroszek [28] present a feature-based approach based on model templates that uses constraints defined in OCL. Our approach currently does not use a generic approach to consistency checking. However, we presented algorithms which can easily be used in such frameworks as we have shown in [9].

Perspectives and Views. Models are powerful as they allow separating concerns in software development. However, the size and complexity of models requires mechanisms to create and work with views. Creating views and perspectives is not a new idea and has been proposed almost 20 years ago [29, 30]. Our approach aims to provide views for three key stakeholder groups in product line engineering. Developers, product managers and customers need different representations when working on a software product line. We use a product line model to integrate these views while allowing each group to use the most appropriate enviroment.

VII. Summary and Conclusions

In this paper we presented a tool-supported approach to support maintaining models of component-based product lines. The approach automatically updates the development view in product line models and checks the consistency between the development, product management, and customer views. It further supports model maintenance by suggesting changes to product managers. Our approach is extensible and new checks and heuristics can easily be added. Using evolution scenarios from a time recorder product line we have demonstrated the usefulness of the approach for model maintenance.

In future work we plan to improve our heuristics for feature suggestions by taking into account semantics. This will allow us, e.g., to detect if adding a certain software-related plug-in to a hardware-related feature does not make sense. We also plan to improve the detection of common evolution patterns with the heuristic, e.g., to recognize that a plug-in has been split and that related features need to be split accordingly. Both improvements will require to take historical data on evolution into account. We also plan to identify further possible warnings and suggestions and develop new algorithms and heuristics to support these. This will require investigating other systems to generalize from the concepts used in Plux.

Acknowledgment

This work has been supported by the Christian Doppler Forschungsgesellschaft, Austria, Siemens VAI Metals Technologies GmbH, and BMD Systemhaus GmbH.

References

[1] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering, Addison-Wesley, 2001.

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

[3] K. Czarnecki, P. Grünbacher, R. Rabiser, K. Schmid, and A. Wasowski. Cool features and tough decisions: A comparison of variability modeling approaches. In *6th Int'l WS on Variability Modeling of Software-Intensive Systems*, pp. 173–182. ACM, 2012.

[4] A. Metzger, P. Heymans, K. Pohl, P.-Y. Schobbens, and G. Saval. Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *15th IEEE Int'l Requirements Engineering Conf.*, pp. 243–253. 2007.

[5] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Software Eng.*, 26(1):70–93, 2000.

[6] R. Johnson, J. Höller, and A. Arendsen. *Professional Java Development with the Spring Framework*. Wiley Publishing, 2005.

[7] M. Svahnberg and J. Bosch. Evolution in software product lines: two cases. *J. of Software Maintenance: Research and Practice*, 11(6):391–422, 1999.

[8] L. G. P. Murta, A. van der Hoek, and C. M. L. Werner. ArchTrace: Policy-based support for managing evolving architecture-to-implementation traceability links. In *Int'l Conf. on Automated Software Engineering*, pp. 135–144. IEEE, 2006.

[9] M. Vierhauser, P. Grünbacher, A. Egyed, R. Rabiser, and W. Heider. Flexible and scalable consistency checking

on product line variability models. In *25th Int'l Conf. on Automated Software Engineering*, pp. 63–72. ACM, 2010.

[10] D. Dhungana, P. Grünbacher, and R. Rabiser. The dopler meta-tool for decision-oriented variability modeling: A multiple case study. *Automated Software Engineering*, 18(1):77–114, 2011.

[11] R. Wolfinger. *Dynamic Application Composition with Plux.NET: Composition Model, Composition Infrastructure*. Ph.D. thesis, Johannes Kepler University, Linz, Austria, 2010.

[12] R. Wolfinger, S. Reiter, D. Dhungana, P. Grünbacher, and H. Prähofer. Supporting runtime system adaptation through product line engineering and plug-in techniques. In *Proceedings 7th Int'l Conference on Composition-Based Software Systems, ICCBSS 2008), February 25-29, Madrid, Spain*, pp. 21–30. IEEE Computer Society, 2008.

[13] R. Wolfinger, M. Löberbauer, M. Jahn, and H. Mössenböck. Adding genericity to a plug-in framework. In *Int'l Conf. on Generative Programming and Component Engineering*, pp. 93–102. 2010.

[14] M. Jahn, M. Löberbauer, R. Wolfinger, and H. Mössenböck. Rule-based composition behaviors in dynamic plug-in systems. In *17th Asia Pacific Software Engineering Conf.*, pp. 80–89. 2010.

[15] D. Birsan. On plug-ins and extensible architectures. *Queue*, 3(2):40–46, Mar. 2005.

[16] R. Rabiser. Flexible and user-centered visualization support for product derivation. In *2nd Int'l WS on Visualisation in Software Product Line Engineering*, pp. 323–328. Lero TR, 2008.

[17] P. Grünbacher, R. Rabiser, D. Dhungana, and M. Lehofer. Model-based customization and deployment of Eclipse-based tools: Industrial experiences. In *Int'l Conf. on Automated Software Engineering*, pp. 247–256. ACM, 2009.

[18] ECMA. *Int'l Standard ECMA-335, Common Language Infrastructure (CLI)*. 2006.

[19] S. Deelstra, M. Sinnema, and J. Bosch. Variability assessment in software product families. *Information and Software Technology*, 51(1):195–218, 2009.

[20] S. Johnson and J. Bosch. Quantifying software product line ageing. In *WS on Software Product Lines: Economics, Architectures, and Implications*, pp. 27–32. 2000.

[21] H. Siy and D. Perry. Challenges in evolving a large scale software product. In *WS on the Principles of Software Evolution)*, pp. 29–32. 1998.

[22] G. Deng, J. Gray, D. Schmidt, Y. Lin, A. Gokhale, and G. Lenz. Evolution in model-driven software product-line architectures. In P. Tiako (Ed.), *Designing Software-intensive Systems*, pp. 1280–1312. Idea Group Inc. (IGI), 2008.

[23] T. Mende, F. Beckwermert, R. Koschke, and G. Meier. Supporting the grow-and-prune model in software product lines evolution using clone detection. In *12th European Conf. on Sw. Maintenance and Reengineering*, pp. 163–172. IEEE CS, 2008.

[24] D. Dhungana, P. Grünbacher, R. Rabiser, and T. Neumayer. Structuring the modeling space and supporting evolution in software product line engineering. *J. of Systems and Software*, 83(7):1197–1122, 2010.

[25] C. Nentwich, W. Emmerich, A. Finkelstein, and E. Ellmer. Flexible consistency checking. *ACM Transactions on Software Engineering Methodology*, 12(1):28–63, 2003.

[26] A. Egyed. Instant consistency checking for the UML. In *28th Int'l Conf. on Software Engineering*, pp. 381–390. ACM, 2006.

[27] X. Blanc, I. Mounier, A. Mougenot, and T. Mens. Detecting model inconsistency through operation-based model construction. In *30th Int'l Conf. on Software Engineering*, pp. 511–520. ACM, 2008.

[28] K. Czarnecki and K. Pietroszek. Verifying feature-based model templates against well-formedness OCL constraints. In *5th Int'l Conf. on Generative Programming and Component Engineering*, pp. 211–220. ACM, 2006.

[29] B. Nuseibeh, J. Kramer, and A. Finkelstein. A framework for expressing the relationships between multiple views in requirements specification. *IEEE TSE*, 20(10):760–773, Oct. 1994.

[30] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.