A Model-Driven Traceability Framework to Software Product Line Development

André Sousa[1], Uirá Kulesza[1], Andreas Rummler[2], Nicolas Anquetil[3], Ralf Mitschke[4], Ana Moreira[1], Vasco Amaral[1], João Araújo[1]

[1] CITI/DI/FCT, Universidade Nova de Lisboa, Portugal,
als12171@fct.unl.pt, {uira,amm,ja,vasco.amaral}@di.fct.unl.pt
[2] SAP Research, Dresden, Germany,
andreas.rummler@sap.com
[3] Ecole des Mines de Nantes, INRIA, France,
nicolas.anquetil@emn.fr
[4] TU Darmstadt, Germany,
mitschke@st.informatik.tu-darmstadt.de
WWW: http://www.ample-project.net

Abstract. In this paper, we present a model-driven traceability framework to software product line (SPL) development. Model-driven techniques are adopted with the aim to support the flexible specification of trace links between different kinds of SPL artefacts. A traceability metamodel is defined to support the flexible creation and storage of the SPL trace links. The framework is organized as a set of extensible plug-ins which can be instantiated to create customized trace queries and views. It is implemented based on the Eclipse platform and EMF technology. We illustrate a concrete instantiation of the framework to support the tracing between feature and use cases models.

1 Introduction

Software product lines (SPLs) methods and techniques [1], [2], [3], [4] aim at producing software system families with high levels of quality and productivity. A system family [5] is a set of programs that shares common functionalities and maintain specific functionalities that vary according to specific systems being considered. A software product line (SPL) [1] can be seen as a system family that addresses a specific market segment. In order to improve the productivity and quality of SPL development, the proposed methods and techniques motivate the specification, modeling and implementation of a system family in terms of its common and variable features. A feature [2] is a system property or functionality that is relevant to some stakeholder and is used to capture commonalities or discriminate among systems in SPLs.

The development of SPLs [6] is typically organized in terms of two main processes: domain and application engineering. The domain engineering focuses on: (i) the scoping, specification and modeling the common and variable features of a SPL; (ii) the definition of a flexible architecture that comprises the SPL

common and variable features; and (iii) the production of a set of core assets (frameworks, components, libraries, aspects) that addresses the implementation of the SPL architecture. In application engineering, a feature model configuration is used to compose and integrate the core assets produced during the domain engineering stage in order to generate an instance (product) of the SPL architecture.

Despite the advantages and benefits of current SPL methods and techniques, most of them do not provide automatic mechanisms or tools to address the traceability between the produced artefacts in both domain and application engineering processes. This is fundamental to guarantee and validate the quality of SPLs development and to allow a better management of SPL variabilities. On the other hand, current traceability tools do not provide support to address the new artefacts (variability models) or processes (domain and application engineering, product management) of SPL development. As a result, they do not allow the explicit modeling and management of SPL features.

In this context, this paper proposes a model-driven traceability framework for SPL development. Our framework aims to support forward and backward tracing of SPL artefacts using model-driven engineering techniques. It proposes to support the automatic management and maintenance of trace links between SPL artefacts of domain and application engineering. It is implemented as a flexible framework in order to allow its customization to different SPL traceability scenarios.

The remainder of this paper is organized as follows. Section 2 gives an overview of a survey of existing traceability tools developed in the context of a joint project. Section 3 details our model-driven traceability framework to software product line development by presenting the framework implementation architecture and illustrating its instantiation to support the tracing between features and use cases. Section 4 discusses implementation issues and further steps of the framework development. Finally, Section 5 concludes the paper.

2 Analysis of Existing Traceability Tools

A survey on existing traceability tools was conducted in the context of the AMPLE project [7]. The objectives of this survey were to investigate the current features provided by existing tools in order to assess their strengths and weaknesses and their suitability to address SPL development. The tools were evaluated in terms of the following criteria: (i) management of traceability links; (ii) traceability queries; (iii) traceability views; (iv) extensibility; and (v) support for Software Product Lines (SPL), Model Driven Engineering and Aspect-Oriented Software Development (AOSD). We believe that these criteria are crucial for this kind of tools since they provide the basic support to satisfy traceability requirements (creation of trace information and querying it) and other important concerns regarding the evolution of these tools and SPL development. These are explained in detail as follows.

The management of traceability links criterion was adopted to analyze the capacity of each traceability tool to create and maintain trace links (manual or automatic) and which kind of trace information is generated. The traceability queries criterion analyses which searching mechanism to navigate over the artefacts and respective trace links is available from the tools, varying from simple queries to navigate over the related artefacts to more sophisticated queries that support coverage analysis and change impact analysis. The traceability view criterion characterizes the supported views (tables, matrix, reports, graphics) that each tool provides to present the traceability information between artefacts. The extensibility criterion evaluates if any tool offers a mechanism to extend the tools functionalities or to integrate with any other software development tools. Finally, the support for SPL, MDE and AOSD development criterion indicates if a tool adopts any mechanism related to these new modern software engineering techniques.

The conclusions that were drawn from our survey were that none of the investigated tools had built-in support for SPL development, and a vast majority of them are closed, so they cannot be adapted to deal with the issues raised by SPL. The surveyed tools were also not developed for the Eclipse platform, and only a few had some sort of mechanism to support software development in that environment [8], [9]. TagSEA is the only tool that was developed as an Eclipse plug-in, allowing the developer to insert specific tags in source files providing some traceability support. However, TagSEA is still in an experimental state and does not cover traceability up to a degree that is desirable.

There is some recent progress in providing traceability support for product lines. Two of the leading tools in SPL development, *pure::variants* [10] and GEARS [11] have defined some extensions to allow integration with other commercial traceability tools. *Pure::variants* includes a synchronizer for CaliberRM and Telelogic DOORS that allows developers to integrate the functionalities provided by these requirements and traceability management tools with the variant management capabilities of *pure::variants*. Similarly, GEARS allows importing requirements from DOORS, UGS TeamCenter, and IBM/Rational RequisitePro.

However, these *external* tools (e.g. DOORS, RequisitePro) handle traceability for traditional systems. Apart from their individual weaknesses (see Table 1), they all lack the ability to deal explicitly with specificities of SPL development, for example, dealing with variability. They do not support advanced and specific support to deal with change impact analysis or requirement/feature covering in the context of SPL development.

Table 1 summarizes some key aspects of the evaluation of the tools that may be integrated with pure::variants or GEARS. In terms of trace links management, the tools allows defining them manually, but offers the possibility to import them from other existing documents, such as, MS-Word, Excel, ASCII and RTF files. CaliberRM and DOORS allow the creation of trace links between any kind of artefacts. RequisitePro focuses only on the definition of trace links between requirements.

The RequisitePro provides functionalities to query and filtering on requirements. CaliberRM allows querying requirements and trace links. DOORS provides support to query any data on the artefacts and respective trace links. Regarding advanced query mechanisms, Caliber RM allows detecting some inconsistencies in the links or artefacts definition, and DOORS offers impact analysis report and detection of orphan code. The traceability tools offer different kinds of trace views, such as, traceability graphical tree and diagram, and traceability matrix. All of them also allow navigating over the trace links from one artefact to another.

In terms of extensibility, CaliberRM allows specifying new types of reports and DOORS allows creating new types of links and personalized views. The three tools also provide support to save and export trace links data to external database through ODBC. DOORS integrates with many other tools (design, analysis and configuration management tools). None of the investigated tools has explicit support to address SPL, MDD or AOSD technologies.

	RequisitePro	CaliberRM	DOORS
(i) Links Management	Manual	Manual	Manual + Import
	Between requirements	Complete life-cycle	Complete life-cycle
(ii) Queries	Query & filter on requirements attributes	Filter on requirements & links	Query & filter on any data (including links)
	–	Links incoherence	Impact analysis, orphaned code
(iii) Views	Traceability matrix, traceability tree	Traceability matrix, traceability diagram, reports	Traceability matrix, traceability tree
(iv) Extensible	–	–	Creation of new type of links
	Trace data saved w/ ODBC	Trace data saved w/ ODBC	Integrates w/ > 25 tools (design, text, CM, ...)
(v)SPL, MDD, AOSD	Not Supported	Not Supported	Not Supported

Table 1. Summary of the comparison of three requirement traceability tools according to our evaluation criteria (see text for explanation)

3 A Model-Driven Traceability Framework to SPL

In this section, we present and discuss the main topics regarding our SPL traceability framework. The framework's architecture is presented, as well as the class diagram for the main modules. Finally, an instantiation of the framework allowing the management of trace links between features and use cases is also shown.

3.1 Traceability Metamodel

The traceability metamodel which is is the basis of the frameowrk discussed in this article is shown in figure 1. It is centered around the assumption that all trace information can be represented by a directed graph.

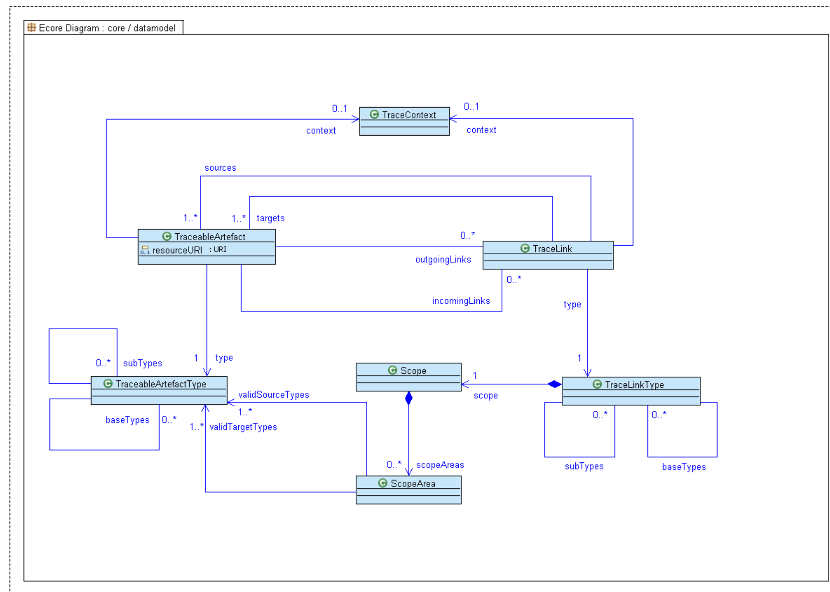

Fig. 1. Traceability Metamodel

The main elements of the metamodel are the following:

- A *TraceableArtefact* represents a (physical) artefact that plays a role in the development cycle. The complexity of such an artefact is arbitrary, it may represent a requirement, an UML diagram, an element inside a diagram, a class or a method inside a class. An artefact is unambiguously identified by a locater (resourceId), which describes where this artefact is located (such as in a file or a directory) and how it may be accessed.
- A *TraceLink* is the abstraction for the transition from one artefact to another. An instance corresponds to an hyperedge linking two artefacts in the trace graph. A transition is always directed, therefore a from-to-relation between artefacts is created by a trace link (between source and target artefacts).
- During the process of tracing information about the design of a software system, different artefacts of different types must be taken into account. For this reason each TraceableArtefact has an instance of *TraceableArtefactType* assigned. This type separates artefacts from each other. Artefact types may

be grouped in a hierarchical manner, which mimics the concept of multiple inheritance known from object orientation.
- Analogous to the type of an artefact each link does have a type because the relation between two artefacts may differ. Examples for such types would be *contains*, *depends of* or *is generated from*. For this reason each instance of TraceLink is assigned an instance of *TraceLinkType*.
- The existence of an artefact or the relation from one artefact to another may be justified in some way. Not all artefacts and transitions would require such a justification, for example a "contain" transition is rather self explanatory. The attachment of additional information to artefacts and links can be modelled by attaching a *TraceContext* to relations and/or artefacts.
- Links of a certain type may only be valid between artefacts of a certain type. A link of type *contains* may be valid between a *Method* and a *Class*, but not between two *Architectural Models*. The narrowing of validity area of link types is modeled via the introduction of the elements *ScopeArea* and *Scope*.

3.2 Traceability Framework Overview

Our traceability framework aims to provide an open and flexible platform to implement trace links between different artefacts from SPL development. In order to address this aim, the framework is being designed and implemented based on the use of model-driven techniques. A traceability metamodel allows specifying different kinds of trace links between SPL artefacts. All the trace links stored must follow the guidelines established by this metamodel. The framework requires the definition of a variability model to allow the tracing of SPL common and variable features along the domain and application engineering stages. The variability model is used in our approach as the main reference to trace the SPL artefacts. However, the design of the framework is generic, so it may be applied outside SPL development.

The following main functionalities are provided by our framework to support the tracing of SPL artefacts: (i) creation and maintenance of trace links between a variability model and other existing artefacts (UML models and source code); (ii) storage of trace links using a repository; (iii) searching of specific trace links between artefacts using pre-defined or customized trace queries. Trace queries can be executed over the trace links in order to select interesting traceability information to help the SPL development or evolution; and (iv) flexible visualization of the results of trace queries using different types of trace views, such as, tree views, graphs, tables, etc.

The architecture of the proposed traceability framework is being defined in terms of four main modules. Each of them is directly responsible to implement the framework main functionalities. Figure 2 shows the traceability framework architecture with its respective modules:

- **Trace Register** - this module provides mechanisms to create and maintain (update, remove and search) trace links between a variability model and other artefacts;

- **Trace Storage** - defines the storage mechanisms to persist the trace links between SPL artefacts;
- **Trace Query** - this module allows to create and submit queries to search specific trace links previously stored; and
- **Trace View** - it is used to specify visual representation of trace links between artefacts resulted from a submitted trace query.

Figure 2 also shows how the different framework modules are connected. The **Trace Register** and **Query** modules use the services provided by the **Trace Storage** module to store and search the trace links between SPL artefacts. The **Trace Query** module can invoke basic trace link queries methods provided by the **Trace Storage** module. Each trace query returns a set of trace links of interest between the artefacts under tracing. After the execution of a trace query, the **Trace Query** calls the **Trace View** module to allow the visualization and navigation over the resulted trace links and respective artefacts.

3.3 Traceability Framework Structure

The traceability framework is structured as an object-oriented framework that defines an infrastructure to provide basic services to search and store trace links and it also offers a set of extension points to create specific SPL traceability functionalities (trace queries and views). Figure 2 shows the general structure of the framework. The `TraceRegister`, `TraceQuery` and `TraceView` abstract classes represent the extension points of the framework main components represented as UML packages. Each of them must be instantiated and customized to address specific traceability scenarios in SPL development. Next we give an overview of these framework classes.

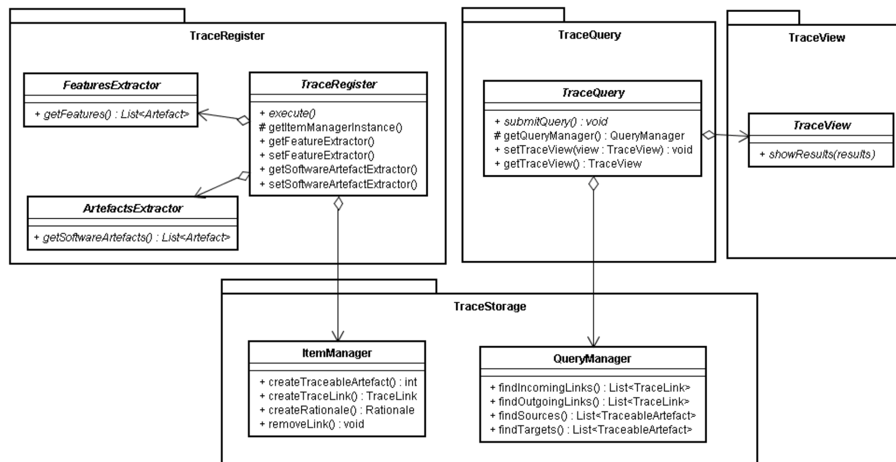

Fig. 2. Traceability Framework Architecture

The `TraceRegister` abstract class must be specialized to create specific ways to create and store trace links between artefacts. The `execute()` abstract method is implemented with this purpose. The trace links are stored using the services provided by the `ItemManager` class of the **TraceStorage** component. The framework does not specify the concrete ways that the trace links must be obtained. This functionality can be provided, for example, by specifying a strategy to automatically identify possible trace links between artefacts or by providing a graphical interface (e.g. check boxes) to allow the SPL developers to create explicitly the trace links. The `FeaturesExtractor` and `ArtefactsExtractor` abstract classes must also be specialized to provide specific ways of extracting features and other desired software artefacts. For instance, if the user wants to extract requirements from a certain requirements modeling tool, then an appropriate class inheriting from `ArtefactsExtractor` must be created and the corresponding `getSoftwareArtefacts()` abstract method must be implemented, which will be responsible for parsing the input file and extracting the desired elements.

The `TraceQuery` abstract class establishes the general structure to implement traceability queries. It uses the services of the `QueryManager` class from the **TraceStorage** component to search trace links of interest. After that, it delegates the resulted trace links from its query to an associated trace view by calling the `showResults()` method. The trace views are implemented as subclasses of the `TraceView` subclass. The current implementation of the **TraceStorage** component provides basic traceability services to retrieve and query basic trace links between specific artefacts. Our proposal in the framework is to create more advanced traceability queries (such as, requirements/feature coverage, change impact analysis, product variants tracing) from these basic ones.

3.4 Framework Instantiation: An Example

In this section, we present an instantiation of our framework that addresses the tracing between feature and use case models. Our aim is to illustrate how the framework can be used and extended to address concrete scenarios of traceability in SPL development. All the framework extension points, presented in Section 3.2, are illustrated in this instantiation.

The feature and use case models used to specify the commonalities/variabilities and requirements of a SPL were created using the following plug-ins: (i) the Feature Modeling Plug-in (FMP) [12] - that allows to create feature models and (ii) the MoPLine tool [13] - a model-driven tool being developed in the context of the AMPLE project to support the process of domain analysis of SPLs.

Figure 3 shows the classes codified during the process of instantiation of our traceability framework. Two concrete extractor classes (`FMPFeatureExtractor`, `MoPLineUseCasesExtractor`) were defined to get the information about the use cases and features defined for a specific SPL. These extractors parse the model files produced by the FMP and MoPLine plug-ins and retrieve the list of features and use cases encountered there. Figure 3 also shows the `FeatureToUseCaseTraceRegister`

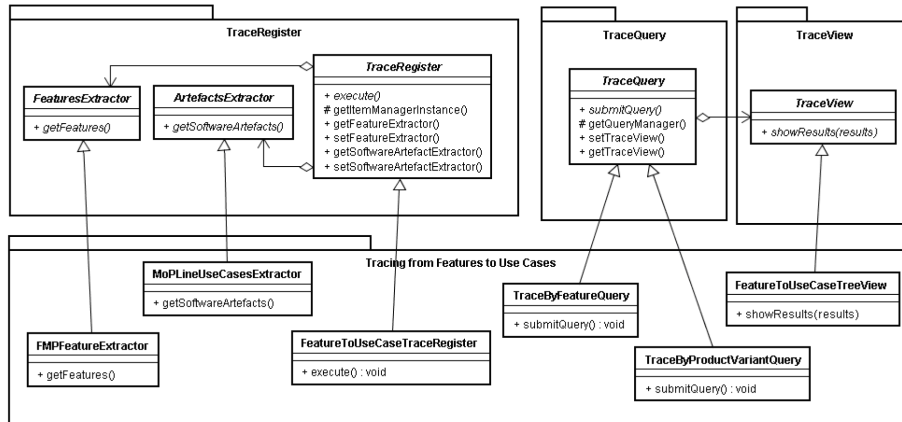

Fig. 3. Framework Instantiation Scenario - Tracing from Feature to Use Case

class that represents an instance of the `TraceRegister` abstract class. It is used to create trace links between features and use cases.

Figure **??** shows the visual representation of the `FeatureToUseCaseTraceRegister`. As we can see, it presents the features and use case (and respective steps) that were collected from the FMP and MoPLine models. The `FeatureToUseCaseTraceRegister` allows defining specific trace links between the SPL feature and entire or partial steps of use cases. After the software engineer defines the trace links, this information is persisted using the `ItemManager` class from the **TraceStorage** component.

In this framework instantiation scenario, we created different trace queries and associated views. Figure 3 shows two subclasses of `TraceQuery`: `TraceByFeatureQuery` and `TraceByProductVariantQuery`. The first one is used to trace the use cases that are connected to a specific feature. The other one is used to obtain the set of use cases related to a set of features that represents a product from a SPL. Both these trace queries can be associated to the `FeatureToUseCaseTreeView` class that represents a specialization of the `TraceView` class.

Figure **??** shows the visualization of an instance of the `FeatureToUseCaseTreeView` that shows the results of tracing a set of features representing a product. The different use cases associated to those features is showed in the tree view graphical component. Although in this framework instantiation example, we have only illustrated the tracing from features to use cases, it is also possible to define trace queries and views to show the tracing from use cases (or other SPL development artefact or model) to features.

4 Implementation Issues and Further Steps

In this section, we present and discuss preliminary lessons learned from our experience of development of the SPL traceability framework.

4.1 Framework Implementation on Top of the Eclipse Platform

The Eclipse has been chosen as the main platform to implement our traceability framework. The justification behind is easily explained: an established MDD-based technology already exists in this environment, called Eclipse Modeling Framework (EMF) [14]. The EMF provides a base platform for model-driven development. On top of this framework, several reusable components that are interesting in this context are also provided, such as, EMF Query, EMF Search and Teneo.

The traceability metamodel which is at the core of the framework has been implemented using EMF. It is a representative implementation of the OMGs EMOF standard. Other technologies used, obviously, have to be integrated with EMF.

A traceability framework has to be able to handle arbitrary sized sets of data - for this reason, it is fundamental to be supported by a database. Eclipse Teneo is used as the abstraction layer between the EMF and the actual database layer to persist EMF model instances. Teneo already supports storage, caching and retrieval of EMF object, although the layer is currently at version 0.8, which means it has not reached its final release state.

Queries on top of EMF models are formulated using EMF Query, which provides a Select-From-Where mechanism that allows basic statements to be executed quite easily. This provides the basis for more advanced queries explained in the next section.

On top of the implementation of the **TraceStorage** component, a collection of UI services can be provided for users and developers. In section 3.3, we have illustrated specific instances of the `TraceRegister`, `TraceQuery` and `TraceView` classes that were used to trace from features to use cases. We are currently organizing these specific subclasses to define a set of customizable classes that facilitates the process of development of new register, query and views. An example is the implementation of a tree view that allows tracing from features to any other SPL artefact. It is being defined based on our development experience on the tree view presented in Figure **??**.

4.2 Advanced Traceability Queries

Basic traceability queries as explained in section 3 are only a part of functionality required. Although basic traceability queries may be used to answer simple questions (i.e. which artefacts have been derived directly from a certain artefact), users are most likely not interested in such queries. Rather, more advanced questions derived from the concrete traceability use case reside in the foreground. In this context, we are currently working in three kinds of advanced queries that are of particular interest of the AMPLE project:

- *Requirements/Feature Coverage*: query that some certain feature is really covered in the applied architecture and/or in the implementation of the system. This could also be extended to test cases;

– *Change Impact Analysis*: discover possible side effects when changing a certain artefact. Such analysis is interesting in forward and backward direction: What would be the result of a change of some requirement on the actual architecture? What would be the impact of the substitution of a certain component by another on some feature?
– *Product Variants Tracing*: perform a trace to all included artefacts in a certain product. This is closely related to Orphan Analysis, where artefacts should be discovered that are included in some product variant, which provide functionality that is not really needed in that variant.

The authors rely on the assumption, that such advanced queries may be composed entirely of a set of basic queries. While basic queries may be expressed in a dedicated query language, a frontend for the advanced query module may be a dedicated user interface guiding a user through the querying process, thus providing an easier access to the application.

5 Conclusions and Future Work

In this work, we presented a model-driven framework to SPL traceability. The main aim of our framework is to support forward and backward tracing of SPL artefacts using model-driven engineering techniques. We have been developing our framework in the context of the AMPLE project. This paper also discussed the results presented in a traceability tools survey. The conclusions drawn from this survey were that the existing traceability tools do not provide a sufficient support to address the traceability problem in SPL development. We presented our proposal for a traceability framework to address traceability in for product lines. Finally, we illustrated the current architecture and implementation of the framework, and provided an example of an instantiation that supports the definition of trace links between features and use cases. Implementation, open issues and further development steps were also discussed.

We are currently working in the evolution of the framework to address other different scenarios of traceability in SPL development, such as: (i) tracing from features to architectural, design and implementation models/artefacts; (ii) analysis of feature interactions; and (iii) feature covering and change impact analysis.

6 Acknowledgements

This work has been partially funded by the European Union in project AMPLE, FP6 IST SO 2.5.5.

References

1. Clements, P., Northrop, L.: Software Product Lines: Practices and Patterns. Addison-Wesley, Boston, MA, USA (2002)

2. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration using feature models. In Nord, R., ed.: Proceedings of the 3rd International Software Product Line Conference (SPLC 2004). Volume 3154., Springer, Boston, MA, USA (2004) 266–283
3. Gomaa, H.: Designing Software Product Lines with UML: From Use Cases to Pattern-Based Software Architectures. Addison-Wesley (2004)
4. Pohl, K., Böckle, G., van der Linden, F.J.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag, New York,USA (2005)
5. Parnas, D.: On the design and development of program families. In: IEEE Transactions on Software Engineering. Number 2. IEEE (1976) 1–9
6. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Tools, and Applications. Addison Wesley Longman (2000)
7. AMPLE: Project AMPLE: Aspect-Oriented, Model-Driven Product Line Engineering. `http://ample.holos.pt`
8. Esterel Technologies: Scade suite. `http://www.esterel-technologies.com/products/scade-suite`
9. Telelogic: Telelogic doors. `http://www.telelogic.com/products/doors/index.cfm`
10. pure-systems GmbH: pure::variants. `http://www.pure-systems.com/Variant_Management.49.0.html`
11. BigLever Software Inc.: Software product lines - biglever software. `http://www.biglever.com`
12. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: Feature Modeling Plug-In for Eclipse. In: Proceedings of the Workshop on Eclipse Technology eXchange (OOPSLA 2004), ACM, Vancouver, BC, Canada (2004) 67 – 72
13. Universidade Nova de Lisboa: Di-fct/unl: Mopline tool. `http://ample.di.fct.unl.pt`
14. Budinsky, F., Brodsky, S., Merks, E.: Eclipse Modeling Framework. Pearson Education (2003)