Model-driven Planning and Monitoring of Long-term Software Product Line Evolution

Mathias Schubanz¹, Andreas Pleuss², Ligaj Pradhan², Goetz Botterweck², Anil Kumar Thurimella³

¹Brandenburg University of Technology, Cottbus, Germany, M.Schubanz@tu-cottbus.de
²Lero – The Irish Software Engineering Research Centre, Limerick, Ireland, {Andreas.Pleuss, Ligaj.Pradhan, Goetz.Botterweck}@lero.ie
³ABB Corporate Research, Bangalore, India, Anil.Thurimella@gmail.com

ABSTRACT
In order to increase the level of efficiency and automation, we propose a conceptual model and corresponding tool support to plan and manage the systematic evolution of software-intensive systems, in particular software product lines (SPL). We support planning on a high abstraction level using decision-making concepts like goals, options, criteria, and rationale. We extend earlier work by broadening the scope in two dimensions: 1) in time, supporting continuous planning over long periods of time and many releases, and 2) in space, supporting traces from high-level decisions down to the implementation. We present a metamodel which allows to represent these concepts, corresponding prototypical tool support, and a first example case using data extracted from an open-source project, Eclipse SWT.

Categories and Subject Descriptors
D.2.13 [Software Engineering]: Reusable Software; D.2.2 [Software Engineering]: Design Tools and Techniques

General Terms
Design, Algorithms, Management

Keywords
software product lines, software evolution, software maintenance, software release planning

1. INTRODUCTION
Complex software-intensive systems represent a considerable long-term investment. In order to realise an economic return-on-investment on the effort that went into design and development, companies that deal with such systems often take a long-term perspective and plan their product portfolio strategically over many months or even years ahead. This applies in particular to software product lines [1, 6], which require additional upfront investment to be established but promise payoff due to expected cost savings, productivity gains and increased quality when deriving products in a more efficient fashion.

So far there is insufficient support to perform the evolution efficiently and in a systematic fashion [5]. To address this problem, we apply model-driven techniques to planned software evolution and release planning, i.e., we aim to (1) represent all relevant information in models and (2) provide automation and/or interactive tool support to support the corresponding activities.

In this paper (1) we extend and refine the metamodel presented in earlier work [7] with concepts to represent dependencies, time, changes in plans, as well as links to implementation-oriented elements. Further, (2) we present a prototypical tool that implements these concepts and supports the analysis and planning of evolution steps. (3) We also address the integration of the high-level plan with implementation-level concepts. (4) To illustrate the approach and provide a first evaluation by analysing the SWT part of the Eclipse Project of the past years in terms of its requirements development down to the corresponding code history.

2. BACKGROUND AND RELATED WORK
This work extends our previous work in [5, 7] on model-driven support for proactive planning and management of SPL evolution. We first introduce the basic concepts of our approach, called EvoPL. Then, we summarize concepts for evolution planning from other areas as a basis for the remainder of the paper. For a more general overview on approaches for SPL evolution please refer to [5].

2.1 The EvoPL Framework
In our previous work [5] we focus on product line evolution on the level of feature models. The evolution of a feature model is specified as “variability over time” using feature model concepts itself: Parts of the model that are added or removed during the same evolution step are clustered into fragments. Each fragment has a unique name and is stored together with a context root node specifying its location within the overall feature model. This way, each feature model version at a certain point in time can be described as a composition of fragments. The hierarchy of fragments (corresponding to the feature model’s tree hierarchy) is specified using a specific “evolution feature model” called EvoFM. A feature in EvoFM represents a fragment. Hence, a configuration of the EvoFM (called EvoConfiguration) results in a selection of fragments which compose a feature model. Ad-
ditional change operators are used to specify changes within the fragments, like adding a cross-tree constraint, changing a feature from optional to mandatory, or moving a feature within the model. Change operators are specified as a “feature” in EvoFM as well, where selecting a change operator in an EvoConfiguration means that it becomes active (i.e., the change operator is applied). The evolution of a feature model can hence be described by a sequence of EvoConfiguration instances.

The modelling concepts above can be represented by a visualization, which we call an evolution plan. An evolution plan represents all EvoConfigurations over time, similar to a product matrix in a product line, but representing evolution steps instead of products (similar to Figure 5).

In [5], we present a model- transformation which creates an initial EvoFM and evolution plan from a given sequence of versions of a feature model. This evolution plan can then be extended (by adding new fragments and change operators) and used to plan future versions of the feature model by specifying EvoConfigurations. As demonstrated in [5], these abstractions can lead to a significant reduction of complexity in terms of the number of model elements required to specify the evolution of a large feature model. However, as these concepts alone are not sufficient to support evolution planning, we introduce in the following other research areas that address planning and decision-making.

2.2 Rationale Management

Rationale management is a discipline supporting decision-making and managing tacit knowledge behind these decisions. A survey on the management of rationale in practice can be found in [9]. Most approaches are based on QOC (Questions, Options and Criteria) [3]. In QOC, decision-making is handled by specifying the available options, judging the options by criteria, and finally selecting one of the options. These concepts are applied to product lines in [11] to decide on variability points and in [10] to address product line evolution.

2.3 Modelling Goals and Requirements

Modelling goals and requirements is a common task in the area of requirements engineering. Van Lamsweerde [12] provides an overview on goal-oriented requirement modelling approaches like KAOS [2]. In KAOS, goals are specified in a goal refinement graph that refines them down to the actual requirements for the system to be developed. The relationships between goals and requirements can be associated with qualitative weights (like contributes, contributes strongly, conflicts, conflicts strongly) to support selection among alternatives. In addition, constraints like an OR can be specified to indicate that two sub-goals are alternative options to satisfy a parent goal.

2.4 Release Planning

The area of release planning [8] aims to support systematic prioritization of requirements to select those to be addressed by the next release(s). Therefore, candidate requirements (e.g., proposed by different stakeholders) are rated by stakeholders according to different criteria like development cost, market value, or strategic benefit. In addition, constraints can be specified like the available resources or dependencies between requirements. On this base, approaches like EVOLVE [4] automatically propose candidate release plans.

Figure 1: Proposed integrated planning process.

3. PROBLEM AND PROPOSED SOLUTION

As shown in Section 2, the literature provides various concepts to support systematic planning and decision-making. However, most of the literature focuses on one particular aspect (e.g., deciding on the next release) while a broader integrated view (e.g., tracing the decisions made) is beyond their scope. The goal of our work is to provide such an integrated view for long-term evolution. This requires broadening the scope in two dimensions: 1) in time, supporting continuous planning over long periods of time and many releases, and 2) in space, supporting traces from high-level decisions down to the implementation artefacts.

Figure 1 illustrates a planning process integrating the concepts from the related work. On a high abstraction level are goals, which can be specified according to goal-oriented requirements modelling concepts (Section 2.3). Goals are refined into concrete requirements and features to fulfill them. As pointed out by research on release planning (Section 2.4), the identified requirements or features are augmented by additional features proposed by other stakeholders (e.g., customers) and constitute the set of candidate requirements and features. In terms of rationale management, the candidate requirements, features, and change requests can be considered as the available options.

The next planning step is to decide which of the candidate requirements/features are targeted by the upcoming release(s). These decisions can be made in a systematic way based on defined criteria (e.g., development cost, perceived value, strategic benefit). This concept is common to both areas, release planning and rationale management. As pointed out by rationale management approaches, all decisions should be documented by rationale to preserve the knowledge for future decisions to be made. The result from this process step is a release plan which has to be realized by an implementation.

The planning process, as described so far, applies to a certain period in time, e.g., a particular release. When addressing long-term evolution the temporal dimension has to be considered. The implementation development has to be monitored as over time there might be deviations from the plan, which require to evolve the plan accordingly. Moreover, developing and running the implementation usually
results in change requests and new requirements caused by detected defects or side-effects. Such new requirements and the latest actual status of the system (potentially deviating from the plan) have to be considered for the next planning process cycle. Thus, decision-making can also benefit from knowledge about previous decisions documented as rationale. In parallel, the high-level information used as a base for decision-making (goals and criteria) can evolve as well in practice.

In [7] we extended the EvoFM concepts using ideas from goal-oriented requirements engineering and rationale management. In this paper we aim to extend this approach to address the needs for a long-term perspective as discussed:

1. In the long run, not only evolution of the system itself must be supported but also evolution of concepts for decision-making like goals, requirements, criteria, and rationale. For this, we present modelling concepts and a prototypical visual editor in Section 4.

2. In addition, the plan itself must be updated according to the actual status of the implementation, which requires monitoring evolution by tracing high-level artefacts down to the implementation. Initial tool support for this is provided by the visual editor as a user interface (Section 4) and "extractors" which provide metadata available in implementation-oriented data sources like bugtracking systems or repositories (Section 5).

4. MODELLING EVOLUTION

In this section we describe modelling concepts for long-term evolution planning, extending previous work in [7].

4.1 Basic Time and Versioning Concepts

As the main goal of our model is supporting the temporal perspective, we first introduce concepts to model time and a basic versioning mechanism. Figure 2 shows an extract of the metamodel for time concepts. Time is managed by a Timeline which contains multiple instances of AbstractPointInTime. A PointInTime can either be a concrete Date or Times-tamp or be abstract like a Milestone or a Release (to be extended by other custom types). An AbstractPointInTime can be associated with a concrete date (e.g., "Release 1" is on "31/12/2012").

We also support versioning within the model. This has several advantages compared to external versioning as it supports a fine-grained tracing of modified elements and enables to add information like change rationale. We provide a simple mechanism by an abstract superclass VersionedElement which is subtyped by all metaclasses to be versioned. A VersionedElement refers to a point in time, to the previous version of the element (if any), and to a change rationale (see [7]).

Figure 2: Metamodel for basic notion of time.

4.2 Evolution of Goals and Requirements

As part of our planning approach, we support modelling goals and requirements using goal models (Section 2.3). Such models specify the refinement from high-level goals down to the actual requirements and further relationships between goals and requirements (e.g., conflicts, requires).

As discussed in Section 3, in long running projects not only the software system itself evolves, but also the corresponding goals and requirements are undergoing change. Since goal models have a hierarchical structure very similar to feature models we can reuse EvoFM concepts (see Section 2.1) to specify their evolution over time. This results in an evolution plan view for goals and requirements, providing an overview of which goals and requirements are active at which point in time.

Figure 3 shows an extract of the corresponding metamodel. As explained in Section 2.1, an EvoConfiguration is associated with a PointInTime and specifies which model fragments and change operators (abstracted as EvoFeature) are selected or deselected as this point in time. Here, the fragments contain elements from the goal model, i.e., goals, requirements, and relationships between them. The change operators support to move a goal or requirement within the model or to add or remove dependencies between them. In the sample cases we explored so far, we used for each goal or requirement a fragment of its own so that an EvoDecision means deciding about whether a goal/requirement is present at a certain point in time (e.g., in a release) or not. An example for an evolution plan at goals/requirements level is shown later in Figure 5 as part of our case study.

Figure 3: Metamodel extract for the evolution of goals and requirements.

4.3 Planning

To support planning, we enhance the above concepts with support for decision-making and rationale. According to the research on rationale management (Section 2.2) and release planning (Section 2.4), decision-making can be supported by first specifying the available options, rating them according to criteria, and finally selecting one of them. In our approach we even go a step further and introduce more fine-grained states as options for the goals/requirements (not only selected and deselected) which are: active (i.e., a goal/requirement is in progress), inactive (i.e., a goal/requirement has been discarded), postponed, or fulfilled. Further, we introduce the possibility to group options to indicate that multiple options belong together (see Figure 4).

The evolution of the plan itself is represented by defining EvoConfiguration as VersionedElements (see Figure 2). Moreover, an EvoConfiguration has a status itself to indicate whether it is a draft plan, the final release version of a plan or whether it shows the actual information about the implementation. Actual information about the system can result
from monitoring the system addressed in the next section.

Figure 4 shows as an example three versions of an EvoConfiguration for 2014. The leftmost version is specified as a draft plan. There are three candidate requirements (R1 - R3) to be decided. For the first one (R1, Speech interface), it has to be decided whether to make it part of the release (active), or to postpone it, or to discard it (i.e., set as inactive). For R2 and R3 it is considered to set one of them as active and postpone the other one which is specified using two option groups. All options are annotated by rationale descriptions.

To support the decision-making, each option can be rated according to criteria (not shown in the figure). The next version shown here (centre of Figure 4) is a released plan. Here, all decisions have been made already, i.e., one of the options has been selected for each requirement. Each decision is explained by rationale. The rightmost version in Figure 4 shows actual information which resulted from monitoring the development progress. It shows that there is a deviation from the original plan as it turned out that R1 had to be postponed. R2 has been fulfilled already and R3 remains as postponed as specified in the plan.

4.4 Research prototype

Tool support and integration with other tools is as a key requirement for planning approaches and rationale management. Thus, we developed a Eclipse\(^1\) plug-in based on the Eclipse Modeling Framework\(^2\) to support viewing end editing the models introduced above.

Figure 5 shows a screenshot of the tool depicting its central view, a visualization of the evolution plan. The prototype supports evolution plans on the level of features and on the level of goals and requirements as introduced in Section 4.3. As an example, the screenshot shows the evolution plan for goals (‘%’) and (%) requirements from our sample case study in Eclipse (see Section 5) for the releases 2010 and 2011 represented by a column each. To illustrate future planning, two additional future releases (2012 and 2013) are shown with arbitrary decisions (see Section 4.3).

While the prototype has still some limitations (e.g., not all parts of the model can be edited) it demonstrates the ability to visualize the evolution of goal and feature models.

5. APPLICATION TO A SAMPLE CASE

In the preceding section we presented concepts for the model-based planning of software evolution. In order to support monitoring of on-going projects we need to trace concepts in the evolution plan to the corresponding elements at the implementation level.

In this section we look at a specific sample case, the evolution of Eclipse projects over time. Concretely, we have mined data from the evolution of the Eclipse SWT project\(^3\) over four years, covering the Eclipse releases 3.2 to 3.6 and thus the time from August 2006 to June 2010. Even though Eclipse is not a product line in the strict sense, it exhibits several characteristics which make it suitable for a sample case, e.g., variability between variant products and an actively managed set of features.

To integrate implementation-oriented concepts which preexist outside of our approach, we have implemented extractors for typical datasources. In detail, we implemented tools to explore the common Bugzilla bugtracking system, the also common Git source code repository, as well as a source code analyzer also extracting Eclipse-specific meta-data information in definitions of Eclipse features and plug-ins and the dependencies between them.

5.1 Eclipse-specific Extension

As a basis for the extraction on implementation level we were able to use the generic tools and extractors as just described. For gathering information on goals and requirements level we considered Eclipse documents on objectives and release planning, i.e., “Eclipse Themes and Priorities”\(^4\) as well as “Project Plans” of Eclipse SWT\(^5\). Eclipse Themes and Priorities are set up by the Eclipse Requirements Council and reflect the overall project goals and requirements, whereas Eclipse Project Plans reside on a lower abstraction level and focus on the time-bound realization of the themes and priorities in the context of a concrete project scope. We interpreted each high level item from the “Eclipse Themes and Priorities” as a goal, each lower level item as a requirement and created refinement links between them according to the given document structure (headlines, lists, etc.).

Based on this information we performed some further analysis. We linked the high-level goals and requirements from the Eclipse Themes and Priorities to the project plan items of the SWT widget toolkit. Here, we had to manually link plan items to the corresponding superior goals and requirements. The next step towards a sufficient trace from Themes and Priorities down to the implementation, was to connect the plan items to implementation artefacts (and corresponding changes). Using the information we extracted, we were able to identify (for a subset of code commits) links from code commits to “high level” goals. This allows, for instance,

\(^{1}\)http://www.eclipse.org

\(^{2}\)http://www.eclipse.org/modeling/emf/

\(^{3}\)http://www.eclipse.org/swt/

\(^{4}\)http://wiki.eclipse.org/RequirementsCouncilThemesAndPriorities

\(^{5}\)http://www.eclipse.org/swt/R3_8/plan.html
to trace from a goal or requirement the number of open and closed work items (Bugzilla bugs) as an indicator for work progress. Further refinement is required in this direction.

5.2 Results

Even while interpreting Eclipse projects was only a very first case study, a set of first outcomes based on the corresponding analysis can be outlined as follows:

Plan evolution – The analysis of the Eclipse Themes and Priorities as a project with long-term planning perspective shows that plans themselves are evolving. For the first release of the document we extracted 28 high-level goals and 67 high-level requirements. Over time some have been modified and quite a few have been added; the last considered version of 2011 contained 105 high-level goals and 154 high-level requirements.

Modification of plan elements – Not only the overall plan does evolve by deleting or adding goals and requirements, but also plan items change. For instance, in the transition from version two (2006) to version three (2007) the goal hierarchy was restructured by renaming “Simple to use” to “Ease of Use”. Hence, change operators as introduced in Section 4.2 (rename, move, etc.) will be required to sufficiently model evolving plan elements.

Goal/Requirement states – The case study confirmed goals and requirements to reside in different states at different points in time and thus reflect evolution. For instance, in Eclipse Project Plans elements are intended to be completed within the planned release. However, in several cases such completion has been postponed. For instance, the plan item “General API for drawing standard UI components” was meant to be finished with the release 3.2 but was then postponed regularly every year up to the version 4.2.

5.3 Discussion

The choice of SWT as a basis for our case study might influence the results. We have chosen SWT since it exhibits long-term development themes, e.g., the continuous effort to support new versions of operating systems. Our finding that that release planning efforts of such a project can sufficiently be modelled and implemented with our concepts, might be biased by that choice. Other projects, with different characteristics and a shorter turnaround time for new objectives, might not be suited for our approach in the same way.

A drawback of the case study is the manual work we had to perform. First, we had to manually extract the Eclipse Themes and Priorities plan. Second, the mapping from the themes and priorities to the SWT project plan had to be done manually. This mapping and potential results might be biased by the perspective of the particular developer.

6. CONCLUSIONS

In this paper, we presented an approach for model-driven planning and monitoring of product line evolution. This includes a metamodel, corresponding tool support, and the application to a first case study. The approach broadens our previous work on evolution in two dimensions: 1) in time, modelling temporal concepts and supporting continuous planning over a long period, and 2) in space, supporting traces from high-level decisions down to the implementation.

The tool support enables tracing information from high-level planning information down to the implementation, including fine-grained development-oriented concepts like bugs and commits. While the approach still has several limitations, we could demonstrate its general feasibility.

In future work we will further improve the tool support. To enhance planning support we aim to integrate concepts from release planning like automated suggestions of release plan. We also aim to provide improved feedback based on monitoring on-going projects, e.g., the estimation of progress based on closed bugs and committed changes. Finally, we aim to provide a more formal evaluation.

7. ACKNOWLEDGMENTS

This work was supported, in part, by Science Foundation Ireland grant 10/CE/11855 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/.

8. REFERENCES