A Product Line Asset Management Tool

Stefan Bellon, Jörg Czeranski, Thomas Eisenbarth, and Daniel Simon

Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart, Germany
{bellon,czeranski,eisenbarth,simon}@informatik.uni-stuttgart.de

Abstract. When developing a software product line, software engineers are confronted with the task of configuration and revision management for the product line as a whole. Furthermore, both on domain and product level explicit variation management has to be provided for. While there are partial solutions to these tasks, there is no integrated support for the product line developers.
In this paper we present a tool for the integrated management of software assets for software product line development. We address the problems of configuration and revision management, explicit variation point handling, and the differences in domain and product development. In our approach, the available solutions to specific tasks are integrated to provide a new solution.

1 Introduction

Companies developing software in a product line face at least three major problems: first of all, product line development makes the task of configuration and revision management much more difficult and error prone compared to the development of a single product. Secondly, explicit variation management in (hierarchical) software product line architectures is required. At last, we have to distinguish between domain development on one side and product development on the other.

There are already partial solutions for these tasks available. But up to now, there is no integrated approach to solve these problems. In this paper, we introduce a tool named Product Line Asset Manager (PLAM) that reflects the roles in software product line engineering as suggested in the STARS Two Life Cycle Model [1]. By the means of these roles we differentiate between the domain and product aspects of the development. The PLAM tool can integrate an arbitrary revision management system that provides basic operations for version control. The tool manages components, variants, and products with their complete revision history and explicitly handles architectural variation points. The PLAM tool serves as an extensible platform for the integration of various available methods for specific tasks in software product line development. The tool covers the whole product line life cycle.

The rest of the paper is organized as follows. In Sect. 2 we give an overview of our PLAM tool. The basic operations of the tool are introduced in Sect. 3.

We then show how to combine the basic operations in order to complete more complex tasks in Sect. 4. Next, in Sect. 5 we give generic examples of product line development processes that are supported by the PLAM tool. Section 6 reports on related work and we conclude the paper in Sect. 7.

2 Tool Architecture

In this section we describe the architecture of our PLAM tool and the architectural model for product lines. In Sect. 2.1 the roles involved in product line development are defined. Sect. 2.2 introduces our generic architectural model. The architectural views of the different roles are presented in Sect. 2.3. Finally, Sect. 2.4 outlines the data storage and management used by the PLAM tool.

As a running example we use a product line of compilers with a standard architecture taken from the compiler domain. As usual, the compiler is composed of a front-end, a middle-end, and a back-end interconnected via a symbol table. The product line contains many variation points. The front-end is implemented either in a monolithic or modular way and analyzes either C or Ada95. The back-end produces code for either RISC or CISC processors and optionally provides an optimizer. The optimizer for the CISC back-end is restricted to the C language. The high level structure of the compiler product line architecture is visualized in Fig. 1 and explained in detail in Sect. 2.3.

2.1 Roles

In the context of our PLAM tool we make use of the following three roles:

Programmer (P) A person responsible for the implementation of the software artefacts.

Product Engineer (PE) / Core Asset Engineer (CAE) A person responsible for a single product or some set of core assets respectively. He is responsible for the coordination of programmers and product specific changes to the architecture.

Product Line Engineer (PLE) A person responsible for the whole product line and its architecture. He coordinates activities concerning the product line as a whole and therefore collaborates with product engineers and core asset engineers.

Any person can occupy several roles at different times. It might be necessary for more than one person to fill a role so that the organization's structure is reflected. For each product line, there is one product line engineer; for each product or core asset there is one product engineer or core asset engineer and several programmers. The roles presented above cover the activities of quality assurance engineers, technical writers, software architects, product line designers, test engineers, and so forth. We do not restrict our model to any specific organizational structure of software product line development. With our approach we provide generic support for the various organizational models and for the management of software product line artefacts that can be found in [2].

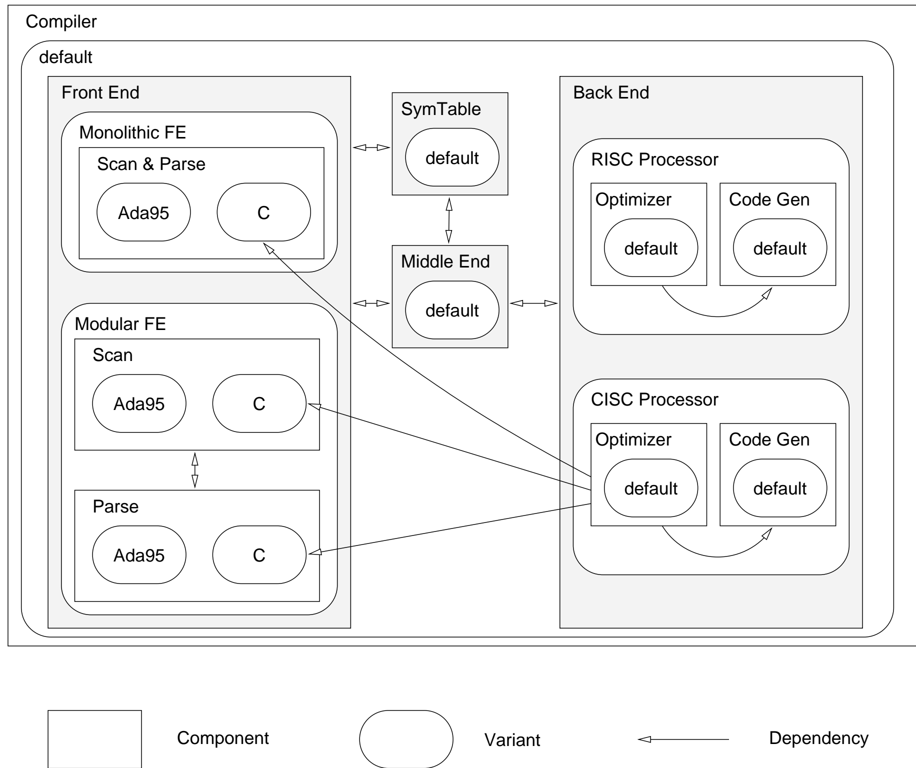

Fig. 1. The Compiler product line as seen from the product line engineer's view.

2.2 Architectural Model

In this paper we describe the architecture of a software product line in a uniform and abstract way by defining the architectural elements and relations between them as follows:

Component A component consists of one or more variants.

Variant A variant either consists of further components (thereby providing the means for hierarchical decomposition), or it consists of releases.

Release A release consists of a set of objects in a state in which they can be deployed as a part of the product.

Object An object is a software artefact such as source code, documentation, or test plans.

Version Objects are available in different versions. Version control can be provided by some standard version control system such as CVS, RCS, or any other available revision management tool.

Action All of the above elements can be associated with actions that are executed for certain operations on these elements.

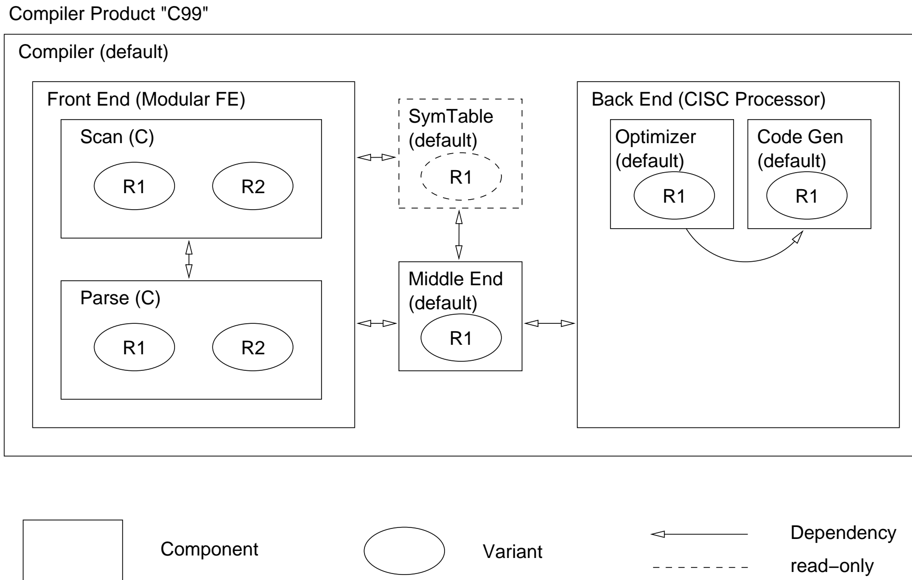

Fig. 2. The C99 product as seen from the product engineer's view.

Because of the alternating structure of components and variants, adding variants where there has not been a variation point before is simple.

Components and variants may have binary relations expressed by dependency arrows in our drawings. Those relations can be "depends on", "excludes", "suggests", etc, but we do not specify the relations in further detail here and leave it to the user of the model to specify relations and consistency checks. Variants of one component implicitly exclude each other, i. e., one has to select at most one of its variants when instantiating a component for a product. If no variant of a component is selected for the initiation of a product, the variant has to be implemented within the product development process. Variants of components model variation points that are bound at product instantiation time. Other kinds of variation points can be captured by the architecture and the implementation of the components, e. g. plug-in mechanisms. Based on those relations and a user specified decision function, the creation of products from the product line can be guided.

All architectural elements and their complete version history are stored in repositories. To each variant of a component a product engineer or a core asset engineer is assigned as responsible for development and maintenance of that variant. Moreover, components can be product-specific and do not show up in the product line architecture at all. Product parts which are considered core assets and which are directly taken from the product line can be marked as read-only in the product engineer's view. Only the assigned persons are allowed to make modifications to the repository objects.

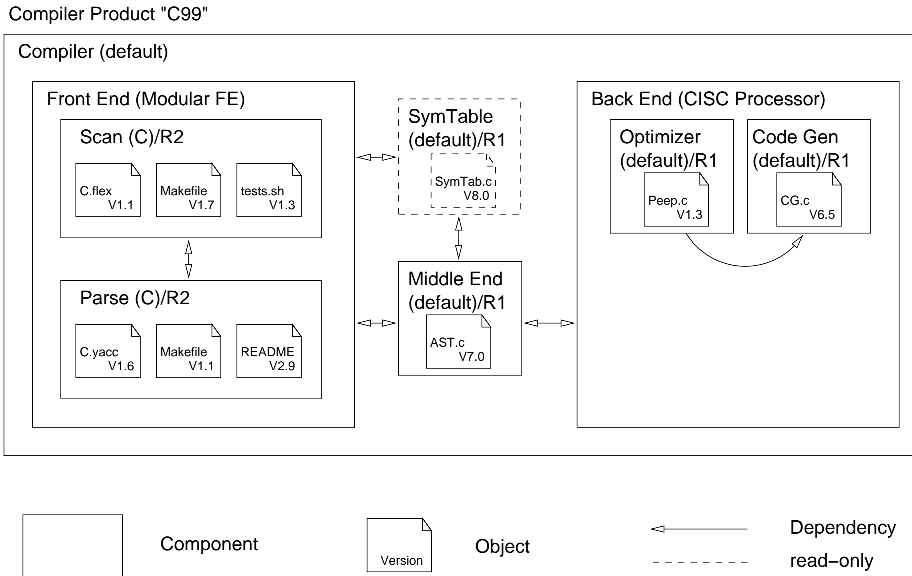

Fig. 3. The C99 product as seen from the programmer's view.

For all architectural elements, actions can be specified for each repository operation. These actions are executed whenever the corresponding repository operation on the architectural element is carried out. In cases when there are several actions to be performed, the user has to provide the order of execution. Actions can be used to implement the binding of compile time variation points. They can be specified using e. g. shell scripts or other generative methods. We give an example for an action in Sect. 4.

We do not specify the architectural elements in more detail here because we believe that the concrete representation can be left to the implementation. In our experience, concrete representations can be mapped onto our model.

2.3 Architectural Views

A software product line architecture can contain all of the architectural elements described in Sect. 2.2. Depending on the role of the user of the PLAM tool some of the information is blinded out. The three different views are illustrated in our product line example in Figs. 1, 2, and 3.

The view of a product line engineer does not reveal the details below the variant level. Components, variants, and dependencies between those architectural elements are visible to the product line engineer. The releases and the versions of objects are hidden in his view. In our example in Fig. 1, the top level component Compiler represents the whole compiler product line. Furthermore there is a component Front End which is implemented by the two variants Monolithic FE and Modular FE.

The view of a product engineer flattens variants relevant for the product into their enclosing components whereas irrelevant variants are simply omitted. In addition, all releases of the components are displayed. E. g. in the view of a product engineer responsible for the development of a compiler C99 with a modular C front-end and an optimizing back-end for CISC processors the architecture is shown in Fig. 2. The component SymTable is marked read-only for the C99 compiler product.

The programmer's view is basically the same as the product engineer's view. However, for each component a release has been selected and the objects of that release are displayed along with their version. In Fig. 3 object C.yacc is checked out in version V1.6 and being part of component Parse (C) in release R2.

Note that the hiding of architectural elements affects the workspace of the particular role. Any person can take a look at a different role's view on the product line for informational purposes.

2.4 Repositories

As depicted in Fig. 4, the PLAM tool manages a number of repositories, one for each product or core asset development group and one for the whole product line. The product line engineer has a repository for storing the product line data, e. g., the product line architecture, the core assets, the product instances of the product line, and the product catalogue. For the product instances, the product engineer and the programmers of a product share a common product repository containing the release catalogue, the product architecture, product specific components, hot fixes, etc. Likewise, the core asset engineers share a core asset repository with their particular core asset programmers.

The PLAM tool reflects the three roles introduced in Sect. 2.1. For each person filling the role of the product line engineer, a product engineer, a core asset engineer, or a programmer exists a workspace where the person can change the architectural elements of his view. Depending on the view the tool provides a number of operations to manipulate the objects in the repository and means for communication between the roles. These operations are described in Sect. 3.

Product Catalogue The product catalogue is administrated by the product line engineer and contains all data required for building each of the deployable products of the product line. This data consists among other things of the product configuration, i. e., the product architecture containing selected variants, product specific components, the access permissions, the parameters of actions, and so on. Furthermore, it contains links to the product repository and core asset repositories that are necessary for the build of the product.

Release Catalogues Upon notification by a programmer the product engineer can create a new release of a component by selecting a releasable version of each object belonging to the component. For the release process, the product engineer can alter his view to make visible the releases and versions of his product's

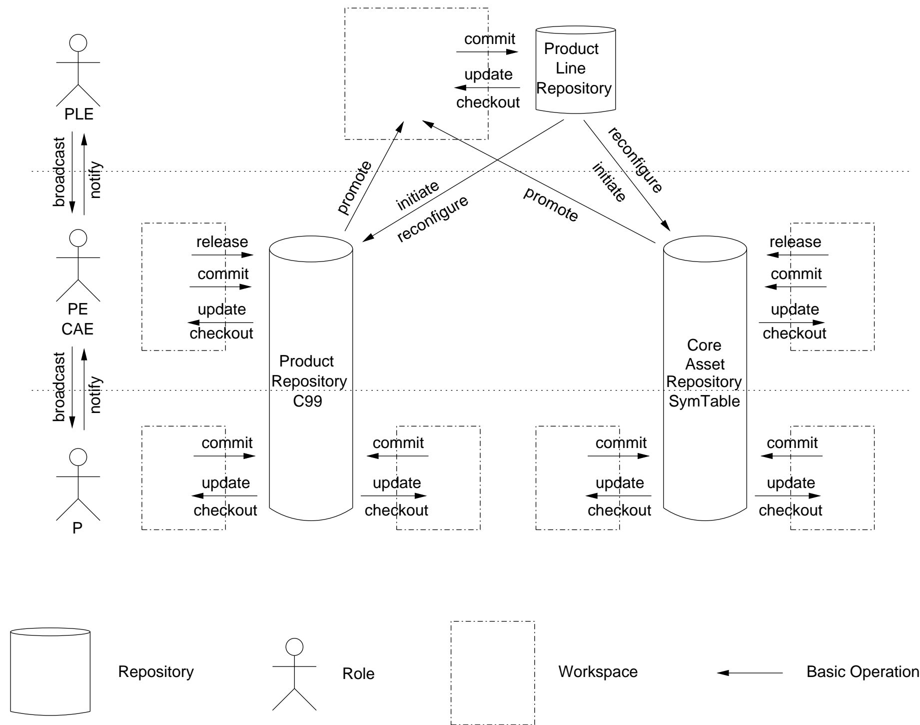

Fig. 4. An overview of the PLAM tool architecture.

components. The selected versions comprise the release which is stored in the release catalogue. The release catalogue contains releases for all components in the product repository. This way, it is possible to recover all deployable product releases. The same procedure applies to core asset development groups and their core assets.

In Fig. 5, the history for the objects of the compiler front-end in the C99 product is shown. Small dashes indicate object versions and the circles mark the releasable versions of the objects. In order to create a release of the front-end at time T2, the product engineer specifies a release $\boldsymbol{r_2} =< a_1, b_1, c_3, d_1, e_1, f_2 >$.

The releases of components propagate through the product line as indicated in Fig. 6. A programmer of the SymTable core asset development group commits his changes to the object SymTab.c frequently. Some of his versions are considered releasable by his core asset engineer. Whenever the core asset engineer creates a release of SymTable the product line engineer can promote it to the product line. Afterwards, the releases of SymTab.c are available to the product engineer of C99. A programmer of C99 can update his local copy of the object as soon as his product engineer reconfigures the product.

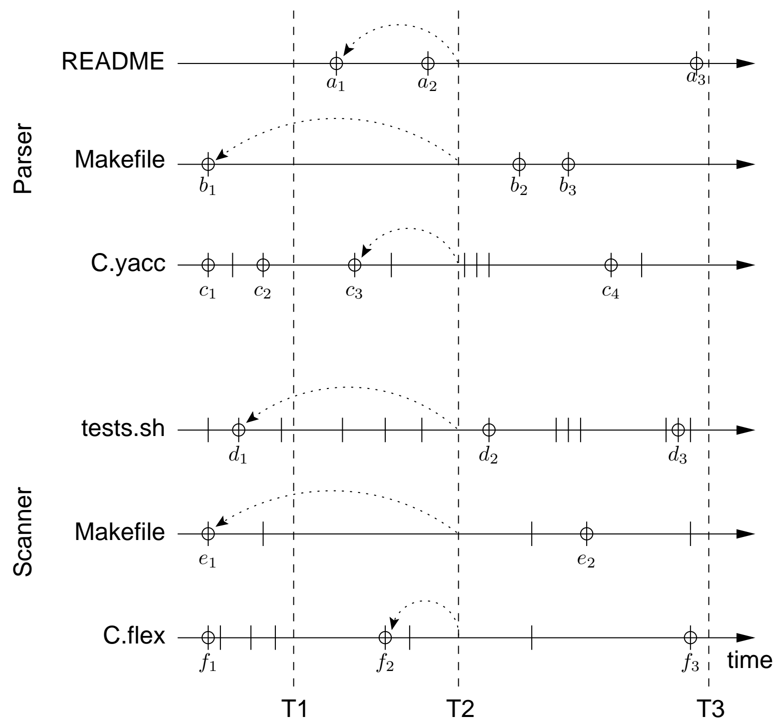

Fig. 5. The history of some objects in the C99 product repository.

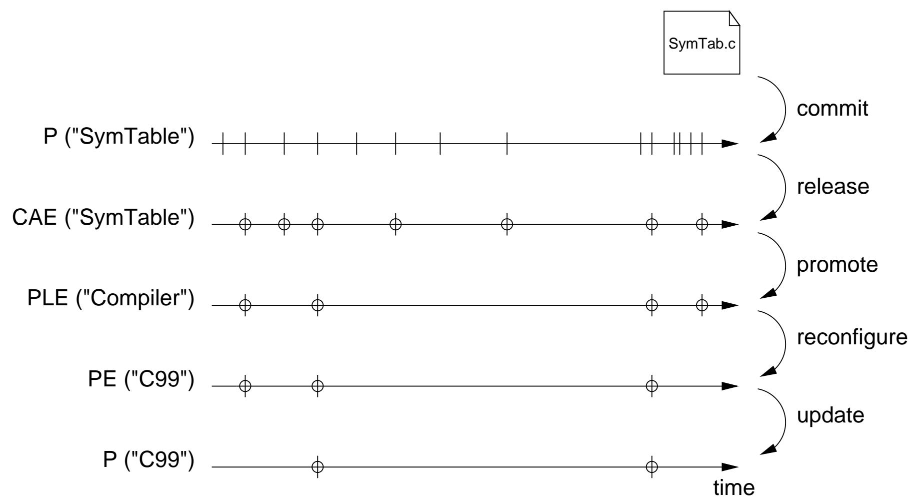

Fig. 6. The propagation of changes to an object through the product line.

3 Basic Operations

The basic operations are divided into three categories. The first category affects the respective workspaces only. The second category of operations interacts with the role's repository. The third category is used for interaction between roles and their repositories. The basic operations of the latter two categories can trigger actions attached to the architectural elements.

We discuss the details of basic operations shown in Fig. 4 in the following:

1. Local
   - **change** Changes the checked out information in the workspace at each level in the tool architecture. This basic operation is not shown in Fig. 4 since it just affects the workspace.
2. Horizontal
   - **checkout/update** Checkout obtains the initial copies of architectural elements from the repository. Update gets newer versions for already obtained architectural elements.
   - **commit** Commit publishes the changes of an architectural element to the repository.
3. Vertical
   - **initiate/reconfigure** The product line engineer and a product engineer jointly create or update a product repository from the product line repository. These operations create the product architecture and verify its dependencies. If initiating a new product or reconfiguring a product due to the replacement of a variant, the variants for components have to be selected. The selection can be supported by a decision model based on the product line architecture.
   - **promote** The product line engineer publishes changes in products or core assets to the product line repository by adopting releases of and architectural changes to the product or core asset.
   - **release** The product engineer accepts specified versions of a set of releasable objects to be part of a released component.
   - **notify** After a programmer commits a version of an object he can notify his product engineer so that the product engineer can release it. Likewise, after a product engineer creates a new release or commits architectural changes to the product he notifies his product line engineer so that the product line engineer can promote them.
   - **broadcast** After a product line engineer promotes changes he broadcasts this information to all the product engineers and the core asset engineers of the product line. If a product engineer considers the changes in the product line useful for his product, he reconfigures the product repository and broadcasts the information about the change to all his programmers.

The first two categories correspond to standard version control operations whereas the last category provides additional functionality required for parallel management of variants and versions in a product line. Examples for the usage of the basic operations are described in Sect. 4.

4 Use Cases

In this section, we show how the users of the PLAM tool can combine the basic operations in Sect. 3 to get their daily work done. We present some of the fundamental use cases in the following. The details such as the appropriate contents delivered by notify and by broadcast, have to be defined by the organization using the PLAM tool. The names for the architectural elements used in this section refer to the examples presented in Figs. 1, 2, and 3.

Initiate new product Suppose we want to build a new product C99 in our Compiler product line. First, the product line engineer and a product engineer assigned to the new product have to initiate the product by choosing appropriate variants for all required components of the C99 compiler. Afterwards, the product repository for C99 is created and the product engineer can start to make product specific changes to the architecture in his workspace. When the product engineer has implemented the product architecture, the programmers can start checking out the sources and developing the product.

Update variants from product line After changes originating from a development group are promoted to the product line by the product line engineer, they have to be obtained by any product engineer who needs them. The product engineer reconfigures the appropriate parts of the product line into his product repository. If necessary he updates the architecture of the product in his workspace and broadcasts the information of the change to his programmers.

Actions associated with the updating of the objects are executed. For example, code generation could be implemented by an action. In our compiler C99 a front-end generator can be invoked on a grammar specification for the C99 language to generate the scanner and parser source code.

New release for component When a programmer of C99 develops and changes components, he commits them to the product repository. If he believes that his objects are releasable he notifies his product engineer. The product engineer approves them and creates a new release of the component. He notifies the product line engineer of the new release.

The existence of a disciplined approval step to validate the changes is an organizational issue. It can be implemented using a reflexion model checker [3]. The source code produced by the programmers can automatically be checked for conformance to the architectural constraints specified by the product line engineer and the product engineer using a reflexion model checker. If one does not have automated reflexion model support, one can perform a review step with similar results instead.

Promote component release to product line If the product engineer of C99 wants to get changes promoted into the product line he commits his architectural changes, creates a new release consisting of the appropriate versions of

the components' objects, and then notifies the product line engineer. The product line engineer approves the changes and can promote a release of a product engineer's component either as a new release of the variant or as a new variant of the component. Furthermore, if the changes to the variant are specific to C99 and not useful for the whole product line, the new release is declared product specific and is not published to the product line.

When the product line engineer finishes the approval, he commits his changes to the product line repository and broadcasts the changes to the product engineers.

Request for bug fix Once a programmer of C99 detects a bug in a component that he is not permitted to change (e. g., the component SymTable), he notifies his product engineer that a bug fix is needed. In turn, the product engineer notifies the product line engineer and then dispatches the bug report to the responsible core asset engineer.

In the unfortunate case of all programmers of the SymTable core asset group being on holidays (or unavailable due to other circumstances), the programmers of C99 can decide to hot fix the problem by overriding the permissions of the SymTable component in their repository. However, such hot fixes are not encouraged since they will be overwritten with the next reconfiguration of the product repository.

When the bug in SymTable is fixed by the core asset development group, the fix is released by the core asset engineer. Afterwards the product line engineer promotes the fix and broadcasts its availability. Then the product engineer of C99 can reconfigure his product repository.

5 Applying the PLAM tool

As we claimed in the beginning of our paper, the PLAM tool does not restrict the development process for the software product line in any way. In this section, we provide generic examples of product line development processes that are supported by the PLAM tool.

First, we take a look at the general setup of product line development processes. There are at least two dimensions to be considered:

Product Line Kick-off In the kick-off phase there is a choice between evolutionary and revolutionary introduction of product lines [4]:

- **Evolutionary** The evolutionary introduction starts with one or more existing products and synthesizes a product line from existing assets by applying reengineering methods.
- **Revolutionary** The revolutionary introduction discards all existing products (if any such products exist) and starts developing the product line from scratch. Nevertheless, in a real-world setting one would apply reengineering methods (such as described in [5, 6]) for asset mining from existing products even in an introduction scenario that is supposed to be revolutionary.

Product Line Operation The second dimension is the choice of how to operate the ongoing development process during the life cycle of a product line:

- **Proactive** Operating the product line in a proactive manner means that there is a planning process for the integration of additional requirements into the product line as a whole and implementing them on the domain level.
- **Reactive** The reactive operation introduces additional requirements in single products as the requirements emerge. Afterwards, a reactive process takes actions on the product line level when a change is considered to be valuable for the product line as a whole. The decision when to integrate the change into the product line can be based on various facts, for example when the same requirement shows up for the second product. Architectural gardening [7] is an example for a reactive process.

We believe that in most cases the development of a product line will be an incremental process. Only product lines that are created once and never changed again are not incremental. All maintenance and evolutionary changes need incremental processes anyway. We found that almost all product line approaches use some extractive process for asset mining or the migration from product specific assets to the product line. Even reusing an architecture of an existing product might involve some extractive reverse engineering step for architectural reconstruction. Note that the terms proactive, reactive, revolutionary, evolutionary, incremental, and extractive are used inconsistently and sometimes even contradictorily by various authors [8–11]. However, the PLAM tool supports each of these flavors of product line development.

There is yet another aspect of classification for product line development:

Domain focussed In this case, there is a full-fledged core asset base where all products are created by configuration of the needed assets and no product specific code is needed.

Product focussed In this case, the only activity carried out in the domain engineering part of the development is modeling. All products share a common architecture but all code assets have to be implemented specific to a product.

For example, the Space Command and Control Architectural Infrastructure (SCAI) in the STARS Two Life Cycle Model [1] is product focussed. A domain focussed product line cannot be managed in a reactive way because there are no changes at product level. A real product line development will be somewhere in between the two extremes of domain and product focussed development. Whatever approach is taken it is supported by the PLAM tool.

As an example for the applicability we demonstrate the fitness of the PLAM tool for an evolutionary and reactive development. In Fig. 7, there is an initial product line containing the source of the legacy product. If there are multiple initial products we create multiple variants of the sole component and try to factor common parts of the product variants in an evolutionary manner. In [12,

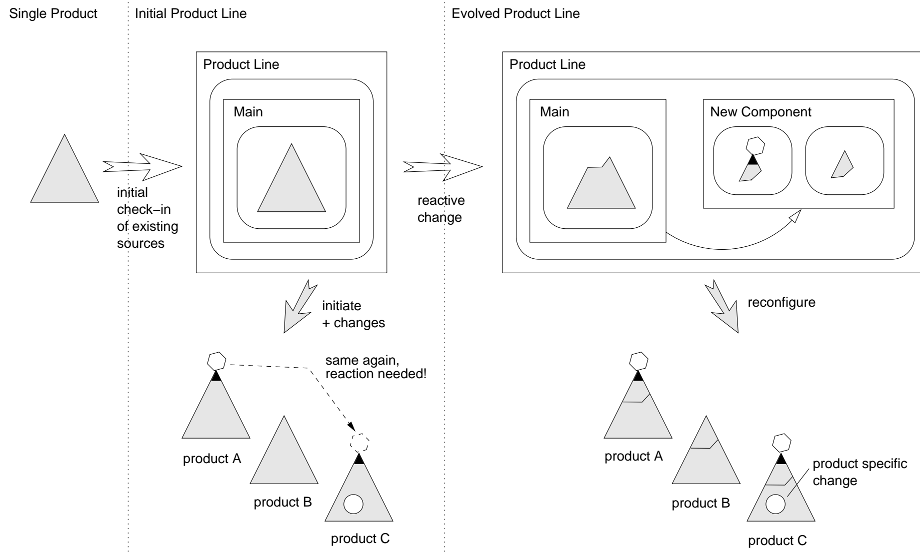

Fig. 7. An evolutionary and reactive development process with the PLAM tool.

13], we present a reverse engineering technique capable of identifying common parts of multiple products.

After the legacy product is imported as the sole component into the initial product line, three products which are based on that component are initiated. In these products, two new requirements show up and are implemented. One of the requirements is needed twice, thereby triggering the reactive process: the implementation of the requirement is factored out into a new component in the evolved product line. Then, the products have to be reconfigured and the product development group of product C has to integrate its local product specific changes into the reconfigured product.

6 Related Work

Clements and Northrop [14, 15] discuss the role of configuration management in the context of software product line development in general. The need for specific tools for software product line development has been recognized in the software product line community [16].

General considerations of the organizational structure of software development companies can be found in Bosch's book [8]. In this book, there is also a discussion of processes for the evolution of software product lines and their artefacts. In Bosch's paper [2] an overview of different approaches to the reuse of software artefacts within an organization is presented.

Krueger [17] describes the variation management as a number of multidimensional configuration management problems. The problems are addressed with

lightweight solutions that help to manage the risks, costs, and time for initiating and running a software product line. The proposed solutions can be realized in the context of our PLAM tool. Krueger [11] presents a taxonomy for characterizing the different software product line approaches.

7 Conclusion and Future Work

In this paper, we presented a tool for the management of software product line assets. Among other things, components, variants, and versions of software artefacts can be administrated with our tool. The PLAM tool is independent of underlying processes and development strategies. It provides hooks for the integration of tools and methods of other developers.

The idea we presented has been implemented by two groups of students totalling 18 persons. One group integrated the PLAM tool as a plug-in for the Eclipse platform [18], the other group extended our existing Bauhaus reengineering tool suite [19] with the necessary functionality. Both implementations have recently been completed.

As future work we want to reconcile some ideas we did not discuss in this paper as soon as we have gained enough experience with the tool in practice. Some of the interesting issues are for example how to manage objects shared amongst different variants or how actions exploit architectural information to generate glue code for plug-in mechanisms. Further challenges we are already addressing are what kind of decision model can be used in practice, how to deprecate components and variants, and how dependencies between releases can be modeled.

References

1. Bristow, D.J., Bulat, B.G., Burton, D.R.: Product-Line Process Development. In: Proc. 7th ASTC, Salt Lake City, UT, USA (1995)
2. Bosch, J.: Maturity and Evolution in Software Product Lines: Approaches, Artefacts and Organization. In: Proc. SPLC2, San Diego, CA, USA (2002) 257–271
3. Koschke, R., Simon, D.: Hierarchical Reflexion Models. In: Proc. 10th Working Conference on Reverse Engineering, Victoria, BC, Canada (2003) 36–45
4. Simon, D., Eisenbarth, T.: Evolutionary Introduction of Software Product Lines. In: Proc. SPLC2, San Diego, CA, USA (2002) 272–283
5. Bergey, J., O'Brien, L., Smith, D.: Mining Existing Software Assets for Software Product Lines. Technical Report CMU/SEI-2000-TN-008, SEI, CMU (2000)
6. Smith, D., O'Brien, L., Bergey, J.: Using the Options Analysis for Reengineering (OAR) Method for Mining Components for a Product Line. In: Proc. SPLC2, San Diego, CA, USA (2002) 316–327
7. Faust, D., Verhoef, C.: Software product line migration and deployment. Software: Practice and Experience **33** (2003) 933–955
8. Bosch, J.: Design & Use of Software Architectures. Addison-Wesley and ACM Press (2000)

9. Bühne, S., Chastek, G., Käkölä, T., Knauber, P., Northrop, L., Thiel, S.: Exploring the Context of Product Line Adoption. In: Proc. 5th Int. Workshop on Product Family Engineering, Sienna, Italy (2003)
10. Clements, P., Krueger, C.W.: Being Proactive Pays Off — Eliminating the Adoption Barrier. IEEE Software **19** (2002) 28–31
11. Krueger, C.W.: Towards a Taxonomy for Software Product Lines. In: Proc. 5th Int. Workshop on Product Family Engineering, Sienna, Italy (2003)
12. Eisenbarth, T., Koschke, R., Simon, D.: Aiding Program Comprehension by Static and Dynamic Feature Analysis. In: Proc. International Conference on Software Maintenance, Florenz, Italy (2001) 602–611
13. Eisenbarth, T., Koschke, R., Simon, D.: Locating Features in Source Code. IEEE Computer Society Transactions on Software Engineering **29** (2003) 210–224
14. Clements, P., Northrop, L.: Software Product Lines—Practices and Patterns. Addison-Wesley (2001)
15. Northrop, L.: A Framework for Software Product Line Practice. Available online at `http://www.sei.cmu.edu/plp/framework.html` (2004)
16. Bass, L., Clements, P., Donohoe, P., McGregor, J., Northrop, L.: 4th Product Line Practice Workshop Report. Technical Report CMU/SEI-2000-TR-002, SEI, CMU (2000)
17. Krueger, C.W.: Variation Management for Software Product Lines. In: Proc. SPLC2, San Diego, CA, USA (2002) 37–48
18. Eclipse Consortium: Eclipse IDE. Available online at `http://www.eclipse.org/` (2004)
19. Plödereder, E., Bellon, S., Czeranski, J., Eisenbarth, T., Koschke, R., Simon, D., Vogel, G., Zhang, Y.: The New Bauhaus Stuttgart. Available online at `http://www.bauhaus-stuttgart.de/` (2004)