Proc. Estonian Acad. Sci. Eng., 2005, **11**, 4, 329–346
https://doi.org/10.3176/eng.2005.4.06

Commonalities of model relationships within product line processes

Jari Peltonen and Maarit Harsu

Tampere University of Technology, Institute of Software Systems, P.O.Box 553, FI-33101 Tampere, Finland; {jari.peltonen, maarit.harsu}@tut.fi

Received 4 August 2005, in revised form 10 October 2005

Abstract. Product line processes use commonalities of products to provide the basis for various technological aids. From the point of view of the tool support, the most significant commonalities are the relationships among models. These relationships may cause interdependence of the model elements and imply functionality to the tools supporting the process. Similarly, as the commonalities of the relationships within product line processes provide the basis for automation, the commonalities of models and the relationships among different product line processes provide a common basis for the tool support for them. The main goal of our research is to provide customizable tool support for product line processes in general. Hence, in this paper we study the nature and commonalities of models and model relationships in various product line processes from the viewpoint of tool support.

Key words: product line process, modelling, automation.

1. INTRODUCTION

The quest for better quality and productivity is the basic reason for arranging activities as processes in organizations. If a project is carried out according to a suitable process, chances for success will grow. A suitable process conforms to the requirements of the current organization, project, product, etc. Even in the present countermovement against overly bureaucratic software development processes, in agile software development (such as [1]), process is still the key concept. It is the cost-benefit factor that counts. The used technology, such as automation of some activities, as well as tools that guide the work or make it easier otherwise, have a great influence on the cost-benefit factor.

Product line processes concentrate on defining, designing, producing, and maintaining software product families, that is, software products with some similar attributes. These commonalities provide the basis for various technological aids within product line processes. One reason for this is that common parts of the

systems can be modelled, and sometimes even implemented, before the design and implementation of the applications. Thus, possible variation (i.e. design decisions to be made) is diminished by fixing the amount of models and model relationships.

The product line processes (process models) can also be considered to form a product line, in which each process model can be considered as an application within the family. There are similarities among the processes, leading to a possibility of forming a common basis for automation of the activities of product line processes. The similarities are necessarily neither in the intended domain of a certain process nor in the activities or their actual order in the process. It is also clear that the details of the processes vary a lot. From the point of view of the tool support, the most significant similarities are in the relationships among models. Similarly, as the commonalities of the relationships within product line processes provide the basis for automation, the commonalities of the models and relationships among different product line processes provide a common basis for the tool support for product line processes in general.

In order to provide customizable tool support for product line processes in general, we study the nature and commonalities of models and model relationships in various product line processes from the viewpoint of tool support. In Section 2 we discuss some related papers to position our work. We also form a background for discussing the model relationships by considering the relationships and dependencies among models in processes in general. We explain the actual comparison with its starting point, criteria, and results in Section 3. Section 4 considers tool support and automation of the tasks in the processes, taking into account found relationships. Section 5 discusses some concrete tools, implemented by our research group and by others as well as tool integration to support the whole product line process. Section 6 finally concludes the paper.

2. MODEL RELATIONSHIPS

2.1. Model relationships as the basis for comparison of product line processes

There exist several papers, reporting comparison of product line processes. For instance, in [2] five product line processes have been compared. It is found that all the processes are different with their special goals and ideology, suiting to different purposes, and thus, the processes do not compete with each other. The purpose of the comparison was to find out the properties of the processes in order to select the most appropriate process for one's purposes, and the focus was mostly on the individual characteristics of each process.

We believe that none of the processes is suitable as such, but a process must always be customized (or even built) for the needs of the current organization, project, product, etc. Hence, instead of finding selection criteria for a process, we concentrate on finding commonalities among the processes in order to find a common basis for tool support for product line processes in general.

Similar reasoning is presented in [3]. A tool has been implemented that automatically analyses the similarity of two processes that are not necessarily product line processes. They use specific rules for formalizing similarity aspects. Also in [4] a generic product-line process framework has been generated acting as a benchmark for existing processes.

In all the above comparisons, the goals, ideology, phases, etc. have been used as comparison criteria. In contrast, we believe that from the point of view of the tool support the most significant similarities are in the relationships among models. Tool support can exploit relationships between any modelled (or meta-modelled) elements, such as UML models [5], single model elements, diagrams, as well as the source code of a program. Actually, in model-centric software development processes everything is considered as models. The relationships may suggest dependencies among the elements and imply functionality to the tools, supporting the process. Examples of such functionality are the tasks of checking and generating models, as well as various tools for guiding the model-building tasks.

In many product line engineering processes, an ordinary software engineering process is used when an application (within the product line) is developed. To form the basic understanding of the model relationships, in Sections 2.2 and 2.3 we briefly discuss relationships within software development processes in general.

2.2. Relationships within software development processes in general

Let us call a set of descriptions, requirements and constraints as design rules, and apply them in different concepts in software development processes. Design rules for a software development process include optional and mandatory activities (e.g. requirement-specification phase, or the task of requirement-specification documentation), and possible and required relationships between the activities in the process. Relationships between the activities include defining the order of the activities, the input and output artifacts (e.g. a UML class diagram) of an activity and relationships among the artifacts of different activities.

For a single activity, design rules include possible and required artifacts, accessed or created during the activity, as well as the possible and required relationships among artifacts within the activity. Activities are often hierarchical, that is, an activity may include a sub-process description with tasks (or sub-activities), their relationships, and input and output artifacts. The models within a single phase usually describe the same thing on the same level of abstraction. In a way, they can be seen as parts of the same model, and completing each other. Hence, also the relationships among the models are strong and imply strong possibilities for automation.

Although each diagram type, used in a process, emphasizes a particular kind of information, the information originates from the same sources, from the target domain and the target system to be specified. For example, changes in one model may imply changes in another, and a portion of one model may be inferred on the basis of another. Accordingly, in addition to the possible inheritance and composi-

tion relationships between models, there can also be more complex relationships among them. These relationships carry information about the meaning of a model in the context of some other models. That is, the process, defining the relationships among the models, can be considered to introduce semantics for the models.

In the early phases of the development process, models are typically very abstract. They will be refined in later phases when more detailed information about the system to be specified is available. The same holds true also for the relationships between the models. Thus, the possibilities of automation in early phases are usually more restricted than in the later phases.

Design rules for an artifact in a phase of a process define the elements and the relationships for each artifact. If the artifact is a UML model, the abstract syntax and well-formedness rules of the type of the model specify characteristics of a legal model instance. Nevertheless, since the semantics of UML models have not been tied, the ways of using a certain model type may vary among processes, or even from phase to phase within a single process. Therefore, it must be possible to restrict and extend the selected model types in each process. In UML, this can be done using profiles.

Regardless of how processes are described, the descriptions of the processes are also models. Thus, it is possible to find relationships also between different processes as well as between the process model and the actually realized process instance. For example, consider an organization with a basic process framework that is varied for each process, e.g. by the size, complexity or other features of the project. The situation is similar to product line architectures.

During the lifetime of a system, the system may undergo modifications to respond to changes in the environment where it is used. Hence, certain aspects in the system need to be redesigned, leading to revised models, describing the modified parts. This implies again strong relationships between the original models and the modified ones. This is a particularly important factor in iterative and incremental process models, but must be considered as a tool support for processes. This is also one of the reasons, why automation possibilities are greater in product line processes compared to ordinary software engineering processes. It may be that there are already several products in the same product family. That is, there may be a bunch of ready made models and defined model relationships existing already.

2.3. Architectural implications to the relationships

Often, the most critical relationships are implied for the selected software architecture. Thus, the architecture (architectural style) of the specified system affects heavily the relationships in the process. OMT++ [6] is an example of a process model that follows a default architectural style called MVC++, where there is a model, a controller, and a view part in object design of each component (similarly to the original MVC model [7]). This fact implies some relationships for the process. The initial version of the model part is a fragment of the analysis model. Similarly, the initial versions of the view and controller parts are generated from a fragment of a dialogue diagram. Initially, each dialogue in a

dialogue diagram is translated to a view class, a corresponding controller class is created and associated with each view class, and the controller of the main dialogue composes the other controllers.

The standard architectural style in OMT++ simplifies the automation of the process since one of the varying parameters is bound. Apart from OMT++, the software architecture is most often specified during the software development process. Thus, the relationships, implied by the architecture, are not known until in the middle of the process. Similarly, change of the architectural style would change the relationships within OMT++. This means that it should be possible to change the relationships within the process according to the selected architectural style.

In software product families the situation is similar to OMT++. Often the common product platform ties the architectural style of the applications. As a result, the application engineering process is simpler than a corresponding process, implemented from scratch, and the possibilities of automation are greater. The models (e.g. UML models) of family members have also relationships among each other. Especially, the models, specifying an application, should be consistent with the product platform.

The rules for different kinds of architectural styles can be specified as architectural profiles as is done, for example, in [8]. This would imply also relationships between the architecture and its profile.

3. IDENTIFYING COMMONALITIES AMONG PRODUCT LINE PROCESSES

3.1. Starting point for the comparison

To find out the commonalities among product line processes, we have made a commonality analysis of the model relationships of four process models: FAST [9], FORM [10,11], KobrA [12] and QADA [13]. The criterion for choosing the processes was the (satisfactory) amount of information we could find about them. These four processes also differ from each other enough to be a fair sample of product line processes. The knowledge about different product line processes is derived from literature, and more detailed version of this study is presented in [14].

Each of the processes has its own characteristics and emphasis. FAST domain engineering provides an environment (consisting of tools and languages) to be exploited during application engineering. It guides the workflow of the process in a traceable and disciplined way by determining the steps that can follow a certain step. FAST reveals mainly temporal workflow dependencies. Software process is seen as decision making, and these decisions have partial orders among each other: some decision must be made before others. For example, family design can be started when both commonality analysis and the decision model have been reviewed.

FORM domain engineering produces feature models and a reference architecture, from which reusable components can be derived during application

engineering. Features and their relationships play an essential role, for example, in grouping the features into subsystems. Composition rules ensure consistency and completeness of features. These rules inform, for example, if several features should be selected together. The features in different models (subsystem, process and module models) are organized so that closely related features belong to the same group to avoid unnecessary relations between different groups.

KobrA concentrates on components and their composition. Common components (called framework components) are produced during domain engineering and instantiated during application engineering. KobrA reveals several kinds of consistency rules. Intra-diagram rules ensure that a single (UML) diagram is complete and well-formed. Inter-diagram rules ensure that diagrams, associated with specification, realization, and implementation, are consistent with each other. Realization rules ensure that realization of a component represents a faithful representation of its specification. Containment, specialization and clientship rules are associated to containment, inheritance and clientship relations, respectively.

QADA emphasizes architectural assessment along with architectural styles and the rationale how requirements guide finding a suitable style. It does not support application engineering like the other processes. QADA supports constraint violation, checking different views (structural, behaviour and deployment views) and refinements between conceptual and concrete architecture. Moreover, these refinements must not violate the selected architectural style. QADA assessment evaluates the architecture against its requirements.

3.2. Criteria for the comparison

The processes have clearly different emphasis, but many "structural" similarities like the division into domain engineering and application engineering, as well as some common phases, are easy to detect. However, the similarities of the model relationships within the processes are not that easily detected – mainly because they are not extensively documented in the process descriptions. Thus, we had to make the comparison indirectly, through the similarity of the models.

We based the comparison on the following assumptions. The fact that each process gives semantics to its models can be reversed assuming that if the semantics of all the models in two processes are the same, the processes are essentially the same. Furthermore, if the semantics of models in two different processes are the same, it can be expected that the relationships in the models and between the models are also the same. For each model relationship that is found to be similar in two processes, it is possible to apply the same tool support.

As actual criteria for comparing the semantics of the models in the commonality analysis, we used model name, model description and the abstraction level of the model. *Names* usually attempt to express the meaning of the model, thus the model names were used as the starting point of searching for commonalities. Nevertheless, the main comparison means used were the *descriptions* of the models, because they revealed the semantics of the models more deeply. However, two descriptions may look the same, but if they are at different *levels of abstraction* (within the

process), the models cannot be considered the same. The level of abstraction refers here to the phase division of the process. The earlier the phase, the more abstract the models typically are.

It is notable that in this comparison, the tasks to be carried out during processes may be different, but they may still produce similar models. This means that the tasks, phases, etc. have no other meaning than revealing the level of abstraction of the models in the process. For this reason, we have "unified" the phases in the processes. That is, we have a single name for the phases on the same level of abstraction.

3.3. Actual comparison and its results

Table 1 shows the comparison criteria, i.e. the models of the chosen product line processes, in condensed form. The table is divided horizontally according to the "unified" phases, and each row contains the models produced during a certain phase. The derivation of the phases is based on both ordinary processes and product line processes. Due to the nature of product line processes, the phases of the ordinary process are repeated twice during product line processes: once in domain engineering and the other time in application engineering.

Each slot may contain several models (beginning with a capital) together with its content (sub-models or description, shown as an itemized list). Due to the different emphasis of each process, the models are not exactly the same (as can be seen in their names and contents), but we have tried to find as close correspondences as possible.

To keep the abstraction levels (rows) consistent, we have, for example, divided the domain model of FAST into two parts: one part produced during domain analysis and the other part during domain design. Originally, FAST has no domain design; it is included in domain analysis. Note also that all models are not present in all processes, and thus, there are empty slots in the table.

Collecting the models was not straightforward. For instance, they have hierarchical relationships, i.e. some models consist of other models (sub-artifacts). When visualizing these relationships between models in tree-like pictures, corresponding models can be found on different levels of different processes. For example, FORM has a domain scope as a part of context analysis, while QADA has a corresponding model, called product line scope, as an upper-level model. In these cases, we compared the hierarchies to find out the most common way to compose the models and tried to identify the most acceptable abstraction among the model compositions.

Table 1 is a starting point for Table 2, showing the "unified models", i.e. the most common models for the processes. Note that not all the models in Table 2 must be found for all of the processes. None of the models is considered as mandatory. Thus, there are also models like "Architectural assessment" in Table 2 since it could easily be added to any of the processes without any side effects if suitable.

The models in Table 2 are considered in greater detail in the next section, where model relationships and their possible influence to process automation are

Table 1. The models of the chosen product line processes

Phase	FAST	FORM	KobrA	QADA
Domain Requirements specification Domain analysis	Domain model economic model commonality anal. decision model	Context analysis domain scope Feature model feature diagrams composition rules decision guidance feature dictionary commonality anal. domain terminology	Context realization commonality anal. framework scope enterprise model structural model activity model interaction model decision model	Requirements spec. Context definition Product line scope economic analysis customer value anal. commonality anal.
Domain design	Domain model family design composition mapping appl. modelling lang. toolset design appl. eng. process	Reference architect. (component) model subsystem model process model module model	Framework comp. specification structural model functional model non-funct. req. spec. quality document behavioural model decision model	Conceptual architecture design structural view behaviour view deployment view Conceptual arch. assessment knowledge base styles

Table 1. *Continued*

Phase	FAST	FORM	KobrA	QADA
			Framework comp. realization structural model activity model interaction model data dictionary quality document. decision model Component containment tree spec. relationships realizat. relations.	Concrete architecture design structural view behaviour view deployment view
Domain implementa-tion	Domain implementation library generation tools analysis tools documentation	Reusable components	Framework comp. embodiment source code structural model phys. comp. model pseudo code	
Testing				Concrete architecture assessment design patterns SAAM
Application requirements specification	Customer requirements	User requirements feature selection spec.	Context realization	
Application design	Application model	Architecture subsystem model process model module model	Specification instantiation Realization instantiation	
Application implementtation	Application doc. Application code	Application software	Components	

Table 2. The most common models of the processes

	Model
Domain requirements specification/ Domain analysis	Economic model Terminology Commonality model Context analysis Decision model
Domain design	Architectural style Architectural views Toolset design Composition mapping Application modelling language Application engineering process
Domain implementation	Reusable components Framework Product line tools
Testing	Architecture assessment
Application requirements specification	Application specification
Application design	Application model
Application implementation	Application

discussed. As the result of the comparison, we can state that the processes do have a lot of models that can be considered to be essentially the same. This means that there are a great amount of commonalities also in the model relationships, and it is reasonable to consider a common "family of tool support" for product line processes.

4. IDENTIFYING POSSIBILITIES FOR PROCESS AUTOMATION

4.1. Automation possibilities implied by general model relationships

The purpose of, and methods used in domain and application engineering in product line processes differ from the purpose and methods of ordinary software engineering. Nevertheless, the nature of relationships, and thus, also the possible automation are essentially similar in both domains. Examples of tool support needed in different phases and different artifacts are collected in Table 3. The listed automating facilities are discussed in the rest of this section from the point of view of model relationships.

As mentioned earlier, relationships may lead to, or actually present a group of dependencies among models. A dependency has semantics, or a rule that specifies how the two models depend on each other. As an example, the relationship between a profile and its instance (a model) is a simple one – the model, as well as every element in it, must conform to its profile. The functionality, implied by checking of the *well-formedness* (Table 3) of the models, that is, the conformity to profiles, is typically simple, including warnings and indication of the possible

Table 3. Produced artifacts and possible tools needed in process phases

Phase	Artifact	Examples of possible tools needed														
Domain requirements specification/ Domain analysis	Economic model		•		•			•						•	•		•
	Terminology		•		•	•		•						•	•		•
	Commonality model	•	•	•	•			•						•	•		•
	Context analysis		•	•	•			•						•	•		•
	Decision model		•	•	•			•						•	•		•
Domain design	Architectural style		•	•	•			•						•	•	•	•
	Architectural views		•	•	•	•		•						•	•	•	•
	Toolset design		•	•	•	•		•			•		•	•	•	•	•
	Composition mapping		•	•	•			•	•					•	•	•	•
	Application modelling language		•	•	•			•						•	•	•	•
	Application engineering process		•	•	•			•			•			•	•	•	•
Domain implementation	Reusable components					•	•	•	•		•	•	•	•	•	•	•
	Framework			•			•	•	•		•	•	•	•	•	•	•
	PL tools					•	•	•	•		•	•	•	•	•	•	•
Testing	Architectural assessment			•	•			•		•				•	•		•
Application requirements specification	Application specification		•	•	•			•						•	•	•	•
Application design	Application model		•	•	•	•		•	•		•	•	•	•	•	•	•
Application implementation	Application		•	•	•	•	•	•	•		•	•	•	•	•	•	•
Testing			•	•	•	•		•		•	•	•		•	•		•
Deployment															•	•	•
Operation and maintenance	Updated components, tools, and applications	•	•	•	•	•	•	•	•	•	•	•		•	•		•
Project & process management	Project plan		•	•	•			•							•		•
	Project report					•		•							•		•
	Assessment & feedback	•	•	•	•			•		•				•	•		•
General	Profiles (e.g. UML prof.)		•	•	•			•									•

problems. However, further processing may vary according to made decisions, e.g. in the guided correction of the problems. In some cases, profiles can also be used in the generation of parts of the model instead of mere checking. This ideology is parallel to the use of patterns, as considered in [15].

The models within a single activity (like phase or sub-phase) of a process usually describe the same thing at the same level of abstraction and from the same viewpoint. In a way, they can be seen as parts of the same model and completing each other. Hence, also the relationships among the models are strong and imply strong possibilities for automation. Examples of this kind of relationships, requiring *consistency* (Table 3) among models, include: class diagrams of the analysis model and the corresponding data dictionary, operation list in the behavioural analysis and the sequence diagram for each operation as well as component diagrams and corresponding deployment diagrams.

The situation is somewhat similar to the above when considering the relationships among models, produced in different activities. However, in general, most of the dependencies are actually partial, more complex than the above examples, and imply highly varying functionality. The partiality of the dependencies may be problematic for preserving consistency. For instance, if a class in detailed design (in class diagram) is removed, should this affect the analysis model or not? How about adding a class or changing its name?

These kinds of decisions are carried out separately for each process or in many cases even for each inconsistency situation. Thus, there may be several possibilities for actions implied by a dependency and in many cases it is the best choice to let the designer of the models to decide which one is applied in a certain situation. Furthermore, interaction with the tool user may, at its best, enable exploitation of even the weakest relationships among the models for the purposes of tool support. It is also important to remember that inconsistency is not the only feature of software engineering process needing functionality. For example, functionalities like *dependency initialization* and *model generation* may be needed.

4.2. Domain engineering specific automation

One of the notable deviations from ordinary software engineering processes is the existence of domain analysis. If there are existing applications meant to be used as the basis of forming a product family, a tool set for finding a common basis for products (*commonality and variation detectors* in Table 3) could help domain analysis phase considerably. The basis of this analysis could be either existing or reverse engineered models of the applications. The result of the analysis could be shown, for instance, by coloring the differences of the static models of the applications. Besides revealing the similarities of the applications, the tool can also show the possible variation points in the application family.

The relationships in the later phases of a product line process are more specific and more detailed than in the earlier phases. Thus, the possibilities of automation in later phases of the process are greater. In contrast to the ordinary software development processes, the early phases of the application engineering

are in the middle, or even in the end, of the whole product line processes. On the flip side of the more concrete and bounded application engineering, the early phases (or even all of the phases) of the domain engineering are even more abstract than in the ordinary processes, since the focus is wider.

In the early models in domain engineering, there are very little or no possibilities for automation (e.g. *economic model*). The created models are very abstract – typically meant only for humans, and hence, a computer can not easily use the possible relationships among the models. Again, in the other phases (like domain analysis, design and implementation) it is possible to automatically check the consistencies and generate warnings or tasks to repair the consistency violations, as well as (possibly) automatically generate the skeleton for the successive models. Naturally, all these issues depend on the specific method at hand.

Altogether, domain engineering aims to create the basis for the automation in the actual application engineering process. For instance, marking of the variability points during domain engineering indicates on one hand the fixed parts, and on the other hand the possible choices within the application engineering. The variability points guide the work and permit automation in application engineering. They can also be incorporated in the base architecture (framework) of the product line (e.g. dynamic plug-in interfaces). Thus, a great amount of the needed automation is not necessarily visible in the process itself, but is found inside of the used *framework*.

A variation of using frameworks is to fix the architecture model in domain engineering, and use a (e.g. visual) *component composition language* or another similar mechanism in application engineering to fill in and vary features of the application. Similar ideas can be used also in testing. For example, by building a common test bench (framework) in domain engineering, it is possible to automate some parts of the testing in application engineering.

4.3. Application engineering specific automation

Application engineering benefits of the work done in domain engineering and other applications. As an example, the gathering and analysis of requirements and the domain are probably accomplished to a great extent, and the architectural style is already fixed. It is also possible to have the basic architecture (e.g. a framework) completed already, as well as some other applications of the same family. As a result, the strong relationships between the common parts of the product line (such as components and features), as well as between other applications in the family, permit application of various potential tools.

During domain engineering, it is possible to form a preset group of choices, for instance, for the basic set of requirements (or features) of the system. Each feature implies a piece of analysis, which again implies a piece of design and so on. When the designer then selects a set of requirements for an application (during application engineering) she also binds some of the features of other models (analysis, design, etc.), and possible choices in the rest of the phases. When the designer then, in later phases (like analysis), binds a variation point,

the effect is similar. This means that the possible variation in the rest of the phases is more limited. Requirements also often depend on each other. If the designer has chosen a certain requirement, the depending requirements should be taken as well. Naturally, this can be done also automatically. The same applies also to the other pieces of models.

All this also implies timely relationships among the parts of the models. It is possible to get an order to the tasks to be performed, based on the relationships modelled in the domain engineering and the choices the designer makes. This leads to a tool that would guide the work by giving tasks to the designer (fixing a variation point, etc.). When the user has completed the tasks, the model changes and implies another set of tasks. In separation to typical *workflow tools*, where there are merely predefined phases with timely relationships, the needed tasks are generated dynamically according to the defined model relationships. However, in addition to these kinds of *task generation tools*, there is also a need to explicitly pre-define the general workflow of the process.

Instead of picking the requirements or other pieces of models from the base of the product line only, it is also possible to use other applications of the same family in the similar fashion. Thus, it is possible to choose requirements, features and models from other applications, and use them to complete the selections made from the base of the product line. These choices can even guide the work and bind other choices even more than the ones made from the base of the product line, since the other application is typically more detailed and more complete than the base of the product line.

When time passes, the product line architecture evolves, and the changes are most often due to the new applications, built in the family. This means that existing applications could also be used to complete certain aspects of the base of the product line in the same way we described how the existing applications can be used as the basis of forming a product family. Again, except the similarities of the applications, it is possible to find also possible variation points in the application family. This feedback from application engineering to domain engineering is basically a reverse engineering tasks. However, at least some mechanisms for abstraction of the models must also be available. In this way, it could also be possible to form a common database, including, e.g. well-defined requirements, use cases, and pieces of analysis to be used in the future applications. This approach is especially applicable in component-based product lines.

5. EXISTING TOOLS AND THEIR INTEGRATION

Even though some of the aforementioned tools seem to be no more than wild ideas, there actually exist several tools for most of the tasks. As an example, an essential task in the context of product lines is to identify the common features either in existing applications or in the domain. Tools for these purposes typically fall into domain analysis category, and their requirements are discussed in [16]. An example of a concrete domain analysis tool is DARE [17]. Commonality

detection can be based on reverse engineering, and tools for this area can be found in [18]. Product line engineering process is often based on frameworks that are thoroughly discussed in [19]. Frameworks can be used to guide building applications by composing from common components or features [20,21].

The above tools and platforms typically address some specific tasks during product line engineering. However, separate tools are not enough, but it should be possible to integrate them into the processes in order to get the real benefit out of them. Furthermore, there should be ways to define the relationships and processes, as well as to attach arbitrary, and possibly interactive functionality to any situations or activities within the specified process. In addition, all this should be possible to integrate with the existing CASE-tools, like editors, version control tools, etc.

As examples of tool sets, we have tools and platforms, developed in our research group. One of them is a tool-independent software platform, called xUMLi [22,23]. It permits building and combining UML processing facilities and using them as integrated CASE tools like Rational Rose [24]. The very basis of xUMLi is a visual data and control flow language called Visiome [25]. A Visiome script consists of activities that are either Visiome components, created as COM components, or other Visiome scripts. Single Visiome activity gets data (typically a model or model fragment) as input, and gives another data (model) as output. Activities have arbitrary but typically relatively simple functionality, and they are combined with Visiome to gain more complex functionality.

In addition to Visiome, xUMLi provides, e.g., a tool-independent API for processing UML models that are created by various kinds of tools. Typical functionality, used in xUMLi, includes transformations between different diagram types, combining and merging the information content of two models of the same diagram types, and construction of more complicated transformation and combination operations based on more simple ones. xUMLi is also used to build more complex tools like ArtDeco architecture validation tool [8,23].

Another tool developed in our group is JavaFrames (earlier known as Fred) [26]. It is built on Eclipse [27] and it provides an extendable framework to be exploited during application development. The extension is specified as an interface, consisting of patterns. Such pattern includes roles that are interdependent and that are bound to concrete program elements during specialization, as well as rules and constraints for them. The specialization process consists of steps, having partial order among each other. The tool uses patterns to automatically generate a dynamic task list, guiding the application programmer. During specialization, new tasks are generated for missing elements and repairing tasks are generated when a constraint is detected to be violated. Examples of tasks are creation of a new class or a new operation. Performing a task may generate new tasks.

From the basis of the aforementioned tools, a tool called MADE (modelling and architecting development environment) is developed in our research group [28–30]. MADE exploits pattern-driven specialization properties from JavaFrames and its task-driven guiding mechanism. Pattern is an abstract concept without semantic contents, and thus, the pure concept can be used in different contexts. MADE

extends the JavaFrames to support also other than Java-specific pattern role types, such as UML and general text files. xUMLi again is at the moment used merely for integrating JavaFrames and Rational Rose.

As already stated above, the basic idea of MADE is to define and describe the possible and required relationships among metaelements in the form of patterns and use this information to maintain and instantiate valid models. However, MADE does not, for example, have possibilities to tie more general, arbitrary functionality to the inconsistency situations, nor exist mechanisms to define, instantiate, or otherwise support the whole software development processes. The customizability in MADE is in the relationships, i.e. patterns, instead of, for example, in functionality.

In Visiome, the models are manipulated programmatically, that is, the relationships of the models are defined and maintained with programs. Mechanism is more complex than in MADE, and there is no easy way to observe the relationships of the models separately from the functionality. However, Visiome as a programming language gives the possibility to define arbitrary functionality at reasonably high level. Furthermore, we use a specialized version of Visiome to describe the whole software development processes, i.e. the workflow, as well as the functionality needed in them. On the contrast to MADE, the customizability is in the functionality in Visiome.

We believe that by combining these two approaches and platforms, Visiome and MADE, we are able to gain the benefits of both of them and to create a basis for a customizable tool support for product line processes in general. Our project group is currently working on the integration of the platforms.

6. CONCLUDING REMARKS

In this paper, we made commonality analysis for model relationships of product line processes in order to consider common basis for tool support of the product line process. We conclude that the processes do have a lot of commonalities in the model relationships, and it is reasonable to consider a common “family of tool support” for product line processes. In our project group, we are currently setting up a project, where the goal is to implement the described customizable family of tool support for product line processes. As the starting point for the implementation, we shall use tools, languages, and platforms produced in our earlier projects, including Visiome, xUMLi, JavaFrames and MADE.

However, we still have to clarify the details of the relationships in the future. It is not enough to know, for instance, that some artifacts depend on each other and one must be completed before the other. Equally, drawing a line, no matter how much decorated, is not enough to specify the meaning of a relationship. When processes are automated, it is essential to know the exact semantics of the relationships, and to describe them in a form that the computer can understand. Furthermore, the process must be described in the way that automated tasks are attached to the appropriate places of the process.

In order to provide customizable and flexible tool support for processes, there must be means to define the relationships and rules in a way that they can be changed when the need arises. That is, any of the relationships cannot be hard coded to the tools. It should also be possible to specialize processes for various purposes. If a general support for processes is desired, a strong mechanism for specification and automation is needed. All this remains as future work.

ACKNOWLEDGEMENTS

This paper was supported by the National Technology Agency of Finland through the project called Inari (Integrated Architecting Environment), 40556/04. We also thank Kai Koskimies, Tommi Mikkonen and the anonymous reviewers for reviewing and commenting the paper.

REFERENCES

1. Beck, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, Massachusetts, 2000.
2. Matinlassi, M. Comparison of software product-line architecture design methods: COPA, FAST, FORM, KobrA and QADA. In *Proc. 26th International Conference on Software Engineering (ICSE 2004)*. Edinburgh, 2004, 127–136.
3. Ocampo, A., Münch, J. and Bella, F. Software process commonality analysis. In *5th International ICSE Workshop on Software Process Simulation and Modeling*. Edinburgh, 2004.
4. Vehkomäki, T. and Känsälä, K. A comparison of software product family process frameworks. In *Proc. International Workshop on Software Architectures for Product Families (IW-SAPF-3)*. Las Palmas de Gran Canaria, 2000, 135–145.
5. Unified Modeling Language (UML) www sites. http://www.uml.org/, 2005.
6. Jaaksi, A., Aalto, J.-M., Aalto, A. and Vättö, K. *Tried & True Object Development: Industry Proven Approach with UML*. Cambridge University Press, Cambridge, 1999.
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. *Pattern-Oriented Software Architecture: A System of Patterns*. Wiley, Chichester, 1996.
8. Selonen, P. and Xu, J. Validating UML models against architectural profiles. In *Proc. European Software Engineering Conference (ESEC'2003)*. Helsinki, 2003, 58–67.
9. Weiss, D. M. and Lai, C. T. R. *Software Product-Line Engineering: A Family-Based Software Development Process*. Addison-Wesley, Reading, Massachusetts, 1999.
10. Kang, K. C., Kim, S., Lee, J. and Kim, K. FORM: a feature-oriented reuse method with domain-specific reference architectures. *Ann. Softw. Eng.*, 1998, **5**, 143–168.
11. Lee, K, Kang, K. C., Chae, W. and Choi, B. W. Feature-based approach to object-oriented engineering of applications for reuse. *Softw. Pract. Exp.*, 2000, **30**, 1025–1046.
12. Atkinson, C., Bayer, J., Bunse, C, Kamsties, E, Laitenberger, O., Laqua, R, Muthig, D., Paech, B., Wust, J. and Zettel, J. *Component-based Product Line Engineering with UML*. Addison-Wesley, London, 2002.
13. Matinlassi, M., Niemelä, E. and Dobrica, L. Quality-driven architecture design and quality analysis method. Technical Report 456, Technical Research Center of Finland, 2002.
14. Harsu, M. and Peltonen, J. Basis for tool support for product-line processes. Report 38, Institute of Software Systems, Tampere University of Technology, January 2005.
15. Selonen P., Siikarla, M., Koskimies, K. and Mikkonen, T. Towards the unification of patterns and profiles in UML. *Nordic J. Comput.*, 2004, **11**, 235–253.

16. Succi, G., Yip, J. and Liu, E. Analysis of the essential requirements for a domain analysis tool. In *Proc. ICSE Workshop on Software Product Lines: Economics, Architectures, and Implications*. Limerick, 2000, 119–128.
17. Frakes, W., Prieto-Diaz, R. and Fox, C. DARE: domain analysis and reuse environment. *Ann. Softw. Eng.*, 1998, **5**, 125–141.
18. Di Penta, M. and Harsu, M. (eds.). *Tools for Software Maintenance and Reengineering*. FrancoAngeli, Milano, 2005.
19. Fayad, M., Schmidt, D. and Johnson, R. (eds.). *Building Application Frameworks – Object-Oriented Foundations of Framework Design*. Wiley, New York, 1999.
20. Van Deursen, A., de Jonge, M. and Kuipers, T. Feature-based product line instantiation using source-level packages. In *Proc. 2nd International Conference on Software Product Lines (SPLC 2000)*. San Diego, CA, 2002. Springer, 2002, 217–234.
21. Vanderperren, W. and Wydaeghe, B. Towards a new component composition process. In *Proc. 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*. Washington, DC, 2001, 322–329.
22. Airaksinen, J., Koskimies, K., Koskinen, J., Peltonen, J., Selonen, P., Siikarla, M. and Systä, T. xUMLi, towards a tool-independent UML processing platform. In *Proc. 10th Nordic Workshop on Programming and Software Development Tools and Techniques (NWPER'2002)*. Copenhagen, 2002.
23. Peltonen, J. and Selonen, P. An approach and a platform for building UML processing tools. In *Workshop on Directions in Software Engineering Environments (WoDiSEE 2004)*, Edinburgh, 2004, 51–57.
24. Rational Software Corporation www sites, Rational Rose. http://www.rational.com, 2005.
25. Peltonen, J. Visual scripting for UML-based tools. In *Proc. 13th International Conference on Software & Systems Engineering and Their Applications (ICSSEA)*. Paris, 2000.
26. Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A. and Viljamaa, J. Generating application development environments for Java frameworks. In *Proc. 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'2001)*. Erfurt, 2001. Springer, 2001, 163–176.
27. Eclipse www sites. http://www.eclipse.org, 2005.
28. Hammouda, I. A tool infrastructure for model-driven development using aspectual patterns. In *Model-driven Software Development – Volume II of Research and Practice in Software Engineering* (Beydeda, S., Book, M. and Gruhn, V., eds.). Springer, Berlin, 2005, 139–178.
29. Hammouda, I., Katara, M. and Koskimies, K. A tool environment for aspectual patterns in UML. In *Workshop on Directions in Software Engineering Environments (WoDiSEE 2004)*. Edinburgh, 2004, 58–65.
30. Hammouda, I., Koskinen, J., Pussinen, M., Katara, M. and Mikkonen, T. Adaptable concern-based framework specialization in UML. In *Proc. 19th IEEE International Conference on Automated Software Engineering (ASE'2004)*. Linz, 2004, 78–87.

Tarkvaraprotsessimudelite ühised omadused

Jari Peltonen ja Maarit Harsu

Tarkvaraprotsessimudelite ühised omadused on üheks tarkvaraprotsessi automatiseerimise aluseks. Arendusvahendite aspektist on kõige olulisemaks ühiste omaduste klassiks mudelitevahelised seosed. Artiklis on analüüsitud tarkvaramudelite ja mudelitevaheliste seoste ühiseid omadusi erinevates tarkvaraprotsessides arendusvahendite ühise baasi koostamiseks.