Increasing Software Product Reusability and Variability using Active Components: a Software Product Line Infrastructure

Bas Geertsema
Information and Computing Sciences
University Utrecht
Utrecht, Netherlands
mail@basgeertsema.net

Slinger Jansen
Information and Computing Sciences
University Utrecht
Utrecht, Netherlands
slinger@slingerjansen.nl

ABSTRACT

Software Product Lines are typically used to support development of a software product family and not a software product population, which denotes a broader and more diverse range of software products. We present a Software Product Line Infrastructure (SPLI) that has been designed to increase the reuse of software efforts in product populations. The SPLI takes a bottom-up approach by structuring product features in highly reusable software components called Active Components which contain different types of artefacts. Variability is expressed using domain-specific models and formal variability models. Variability is bound during product derivation by executing model-to-artefact transformations. Components are active because they are invoked during the derivation process, thereby empowering the component. The SPLI enables step-wise refinements of applications by allowing specialization and composition of models before variability is bound. A prototype of the SPLI has been created that was used to develop and evaluate an experimental software product line. It is concluded that within the context of our experimental software product line the SPLI improves software reuse in software product populations.

Categories and Subject Descriptors

D.2.13 [**Software Engineering**]: Reusable Software—*Reusable libraries, Reuse models*

General Terms

Design, Experimentation

Keywords

Software Product Lines, Active Components, Variability, Components, Model-Driven Development

ECSA 2010, August 23-26, 2010, Copenhagen, Denmark.

1. INTRODUCTION

Software reuse is considered one of the most powerful ways to address the challenges in developing increasingly more complex software systems [16]. By reusing software, development efforts are amortized which leads to reduced development costs. Other goals of software reuse are increased quality and faster development time [9, 16, 17].

Software product lines (SPL) take a systematic approach to multi-system development and are an embodiment of software reuse [20]. In contrast with single-system development, a software product line is set up to exploit commonality between features of a software product family. The production of software systems is made more economical because building a new software system becomes mere a matter of integration and configuration rather than development from scratch [9]. Members of a software product family are by definition distinct which means that there is always variability that must be addressed during product derivation. In a software product line commonality is captured in reusable software artefacts that expose variability to adapt the artifact to the requirements of a specific software product. It is this explicit notion of reusability and variability that marks a software product line approach.

Compared to a software product family, a software product population [18] encompasses an even more diverse set of software products; a family of software product families. Product populations can be seen at software vendors that develop archetypical types of software systems for a wide range of markets. The company where we did our research develops typical administrative business applications for many customers operating in different markets. Each market and client has its own peculiarities and the software is adapted accordingly, both in features and application design. For example, not every customer requires the administration of financial contracts (a feature), but for those that do, the requirements for financial contracts are often specific to that customer (design). Software product populations can also be expected at large (online) application development platforms or with applications that are backed by large component stores. When a SPL has to support a software product population, the SPL scope increases and the SPL variability becomes greater in order to be able to derive all products in the population. An engineering challenge indeed, especially considering that the development of all these components is likely subject to distribution over multiple (independent) development teams. The support for

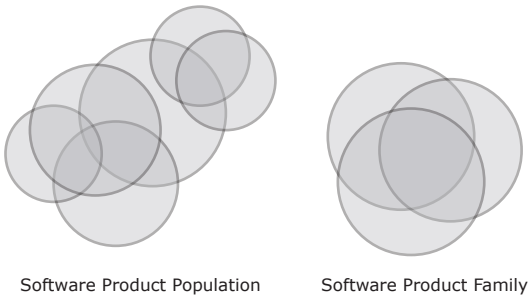


Figure 1: Overlapping features.

a product population is attractive however, as software efforts are amortized over a wider range of software products.

We recognize that the line between a software product family and a software product population is not a rigid one. However, for our research we assume that software product populations show the following characteristics: a) there is no central, or median software product in a population although some products will have more weight and will be more reused than others and b) a feature is reused in more heterogeneous software products. The implied challenges are the increased complexity due to the larger number of features and the increased variability within each feature. In a product population a software product variant cannot be defined relative to a central all-encompassing software product using negative variability. The strategy that we propose in this paper is tackling this in the following ways: 1) reduce complexity by employing domain-specific models and structure both implementation and design in hierarchical components, 2) increase variability by empowering components and using model-driven generative variability mechanisms and 3) specify software products using positive variability through composition and step-wise refinement.

We present a Software Product Line Infrastructure (SPLI), an SPLE approach that is designed to support software product populations. Based on the aforementioned strategy, we do this by taking a bottom-up approach in which product features are structured in highly reusable software components called Active Components. Using the Model-Driven Engineering (MDE) paradigm[15], variability, or application design, is expressed in domain-specific models and formal variability models. Variability within components is achieved by executing model-to-artefact transformations during product derivation. Active Components are composed hierarchically and can be refined step-wise. The components are 'active' because they are actively queried and invoked during the derivation process. Distinguishing features are the notions of model specialization and model refinement and the derivation process in which a global application model is established before variability is bound within components.

The structure of this paper is as follows. First we will explain the research method in Section 2. In Section 3 and 4, we elaborate on the design rationale and describe the structural and behavioral design of the SPLI. A prototype and an experiment are described in Section 5 followed up by an overview and discussion of the results in Section 6. Related work is discussed in Section 7 and we conclude in Section 8.

2. RESEARCH METHOD

The research was triggered by a software development company that used a software product line approach with a strong emphasis on code generation. Their main concern was the limited reuse of software efforts done for specific clients. More specifically, they recognized that they were developing a software product population on their software product line and that they needed to increase reuse of software efforts by structuring product features and application design into reusable components. The research was conducted on-site and there was continuously interaction between the researchers and the developers at the company. The Software Product Line Infrastructure (SPLI) was evaluated by developing a prototype and a small experimental software product line. The prototype is an implementation of the SPLI. It evaluates the feasibility and practicability of the design. The feasibility determines whether the design can indeed be translated into a working and functional software system upon which software product lines can be developed. The practicability aspect includes the experienced development time and difficulty of the development of the prototype. This provides insight for software development companies in the attainability of adopting the SPLI. The experimental software product line was developed on the prototype and evaluates the design goals of the SPLI, most notably the support for a broad range of software products and the use of step-wise refinements of applications. This is done by developing reusable core artefacts, Active Components, and two final applications that share common application design and common features but vary are at the same time considerably in their user interface.

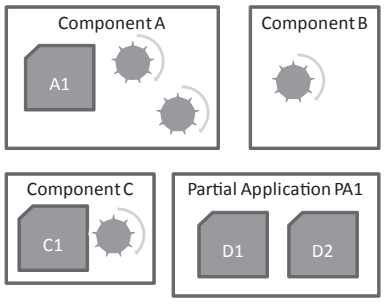
3. DESIGN RATIONALE

The Software Product Line Infrastructure (SPLI) consists of structural and behavioral aspects. Its design is guided by the challenges associated with software product populations: 1) Reduce complexity by structuring features into reusable and self-describing components that can be composed hierarchically, 2) Increase variability of components by employing domain-specific models and generative variability mechanisms and 3) Allow step-wise refinements of application design by using composition and specialization of domain-specific models.

3.1 Active Components

The increased product line scope when dealing with a software product population leads to extra complexity. Ommering and Bosch [19] suggest that this can be reduced by moving from pure variation to both variation and composition. I.e., there is not a top-down architecture-driven approach, but a bottom-up component driven approach. By moving features from a single architecture to multiple independent components the architecture is not becoming more complex with every developed component; only the selected and relevant components for a software product together make up the features of a software product. The design of the SPLI is based on the premise that this idea of divide and conquer is essential to manage the involved complexity when a wide variety of software products must be supported by the software product line.

A product feature can span multiple types of artefacts. Besides source code files there can also be e.g. models, configuration files, data files and images. A component that




```
PA1 = B(A)
Application = C(PA1) = C(PA1(B(A)))
⇒ApplicationModel = C1(D1(D2(A1)))
⇒Input for transformations
```

Figure 2: Components and partial applications.

represents a feature must therefore also contain multiple types of software artefacts at possibly different abstraction levels. The consistent propagation of variability between all artefacts is the responsibility of the component which is why transformations are defined by components themselves; an Active Component must ensure internal integrity upon reuse.

Application design in the context of the SPLI equals to the expression of variability in domain-specific models. An Active Component can provide both implementation (e.g. source-code, libraries, generators) and/or design (e.g. entity models, state machine models, variability models). A component that only supplies design in the problem space is considered a partial application, as it is intended for further refinement during application engineering, but it does in itself not provide a feature in the solution space. The distinction between components and applications is narrow here: an application can be considered a component that is intended to describe a final software product. We have opted to distinguish between components and application however as it is easier to understand and fits in better with the overall SPL paradigm. An example can be seen in Figure 2 where there are three components (A, B and C) and one partial application (PA1). The partial application it defined by its composition (A, B) and defines two domain-specific models that expresses design specific for that application (D1, D2). The 'final' application, i.e. the software product that is derived from the SPL, is defined by the composition of the partial application PA1 refined with the third component C. The resultant application model is the merging of all models defined in both the components as well as the partial application. This application model is subsequently used to drive the transformations contained within the components A, B and C.

Active Components are noted active because an Active Component provides executable specifications that are invoked during the derivation process, thereby creating polymorphic behaviour. This operational interface must be implemented by all Active Components. We have opted for this approach as components in a product population can be used in very heterogeneous software products and are possible developed independently. By allowing participation during derivation, an Active Component developer can check the validity and integrity of the application design that is specified by the application engineer. It is for example possible to do a static analysis on the application model before all transformations are executed during the derivation process and errors and warnings can be returned to the application engineer. This choice has been made to empower the component developer. In a bottom-up approach there is no central authority, or a central architecture, that constraints or checks the derivation. This responsibility is transferred from the architecture to the components.

3.2 Variability

As SPL variability inevitably increases when software product line scope increases, the need to manage and specify this additional variability is emphasized. We turned to the paradigms of Model-Driven Engineering (MDE) [15] and the closely related Generative Programming (GP) [10] for this. In MDE models are used to capture system design. During development these models can automatically be transformed to other software artefacts, such as source code files. Goals of model-driven development are: development speed, software quality, separation of concerns, reusability and manageability of complexity through abstraction [22]. These are all qualities that are also of importance in a software product line and therefore the use of MDE in software product lines seems like a natural fit. Indeed, this combination has been researched quite extensively in other approaches [7, 12, 13, 14, 23]. Models can be either a one-size-fits-all model, e.g. UML, or domain-specific models. The SPLI makes use of the latter, as it can be divided in multiple models which are better for dealing with complexity [24].

The primary variability mechanism used within Active Components is that of model-to-artefact transformations such as templates. The transformations operate on the merged single application model as input and can have different types of software artefacts as output. A common output is that of code files, i.e. code generation, but configuration files or data files can also be the output of a transformation. Variation specifications in domain-specific models can have much richer semantics compared to configuration settings that is found in variability models and can therefore drive flexible variability mechanisms.

Domain-specific models are at an abstraction level that is close to the problem domain, meaning that variability can be specified in a precise and intentional way. The use of domain-specific models to express variability is in contrast with SPL approaches that use a single type of model, often a feature model or variability model, as the only way to express variability[2, 21, 25]. In these models, product features or variation points can be enabled or disabled to match the requirements of a specific software product. These decisions are used during product derivation to construct a tailor-made application. Variability models are orthogonal to the software system and describe the variation points and variants in a SPL. A variability model can take on many forms, ranging from a mere informal diagram to a formally defined model that can be used as input for automated product derivation [3]. Variability models have the benefits of making variability explicit and easily configurable. We believe that it should be used as a valuable software artefact in the engineering process; a variability model in a SPL should not only be a formalized diagram, but rather be integrated in the software product line engineering infrastructure. Because of the prevalence of variability models and its ease

of use for product configuration we have opted to include the use of variability models in the SPLI. Active Components thus rely on both domain-specific models and variability models to express variability. For example, an Active Component that generates a data access layer for databases can make use of variability models to expose variants that have different performance characteristics and also make use of Entity-Relation Diagrams, a domain-specific model, to adapt the component to the entities and relations as defined in the model.

A definitive variability meta-model is not defined by the SPLI; there are multiple meta-models that can be used to describe variability models, e.g. [21] or [25]. The SPLI dictates, however, that once a meta-model is chosen that it is consistently used by all Active Components. I.e., the variability meta-model becomes a standard on the SPL and describes the models in which components must define their variation points and variants. The benefit of using a single meta-model is that during application design all variation points and variants can be combined and presented to the application developer in a uniform way.

3.3 Step-wise Refinements

Step-wise refinement is a powerful paradigm for developing a complex software system by incrementally adding features [6]. A feature in the context of the SPLI includes implementation artefacts as well as application design artefacts such as models. As application design is captured in domain-specific models, it must be possible to refine these models as well. This is similar to for example cascading style sheets (CSS) in web design. A technique called model superimposition has been used for UML by [1]. The SPLI does not suggest a definitive technique on model specialization. For instance, in our prototype we worked with explicit references to model elements, rather than specialization based on syntax as used in model superimposition.

As Active Components contain both implementation and models, a refinement of a component is refinement of the contained models as well. I.e. for component A and B containing models A.M and B.M the composition A(B) is implicitly also the composition A.M(B.M). A software product is defined by the composition of its component, thus upon derivation all models within these components are merged according to the composition of the components into a single application model. In the SPLI models can be specialized and refined with the same ease as for example source code.

A software product line can be based around a central all-encompassing software product. In such a setup, software product variants are derived by excluding features, i.e. negative variability. We assume that there is no central all-encompassing application in a software product population, and therefore applications in our SPLI are defined by 1) the composition of components (features) and 2) the variability decisions captured in either the variability model or domain-specific models. Features and variability decisions are incrementally added in this setup, i.e. positive variability. We call this application design and it is beneficial when this design can also be reused and refined because a lot of efforts might go into defining it.

4. SPL INFRASTRUCTURE

The SPL infrastructure (SPLI) follows the traditional distinction between domain engineering and application engineering. During domain engineering the Active Components are developed and during application engineering application design is captured in domain-specific models, variability models and component composition. An overview of the design can be seen in Figure 3. We will review the structural aspects first and continue with the behavioral aspects.

4.1 Structure

Product features are structured in Active Components. A product feature can span multiple software artefacts and therefore an active component can contain a coherent set of software artefacts that are related to a feature or feature set. Active Components are developed during domain engineering and are always intended for reuse. An active component can include multiple types of artefacts on multiple abstraction levels. In our SPLI we defined the following non-exhaustive list of software artefact types: meta-models, domain-specific models, model constraints, variability models, transformations, programming libraries and static artefacts.

Meta-models define the problem domain in which the component operates. This essentially allows component developers to supply the application developer with meta information on how to use this component, i.e. which kind of models to supply.

Models within components are developed during domain engineering and are intended as default models or as models that need to be specialized during application engineering. For instance, a web-driven security framework might have a core model in which a user account table schema is defined. Common application design can be captured by models in components and reused by applications. Reuse is in this approach not only on the code level; it is just as important that models can be reused as well.

Model-to-artefact transformations are defined in Active Components to generate software artefacts during derivation that are integrated in the final application. The input of these transformations is the global application model. Transformations are essential since they are the primary way of propagating variability that is expressed in domain-specific models into the derived software artefacts. Typical transformations are model-to-code transformation in which source code files are generated based on models. However, also configuration regular text files or xml data files can be generated. Transformations can take on different forms; a common form is that of templates in which text is outputted and an integrated programming language can be used to control output based on the model that is used as input.

Model-to-model transformations are executed prior to the model-to-artefact transformations. They allow a component developer to make last-minute changes to the application model.

Variability models provide a way to specify variation points and variants within the component. During application engineering specific variants can be enabled or disabled. During derivation, these variant choices can be queried during execution of transformations and upon selection of artefacts for integration.

Programming libraries are traditional libraries that can be used during runtime. Examples are math library or database access drivers. Programming libraries do not expose variability that bind during derivation. Variability is possible, e.g. by sub classing in object-oriented libraries, but typically

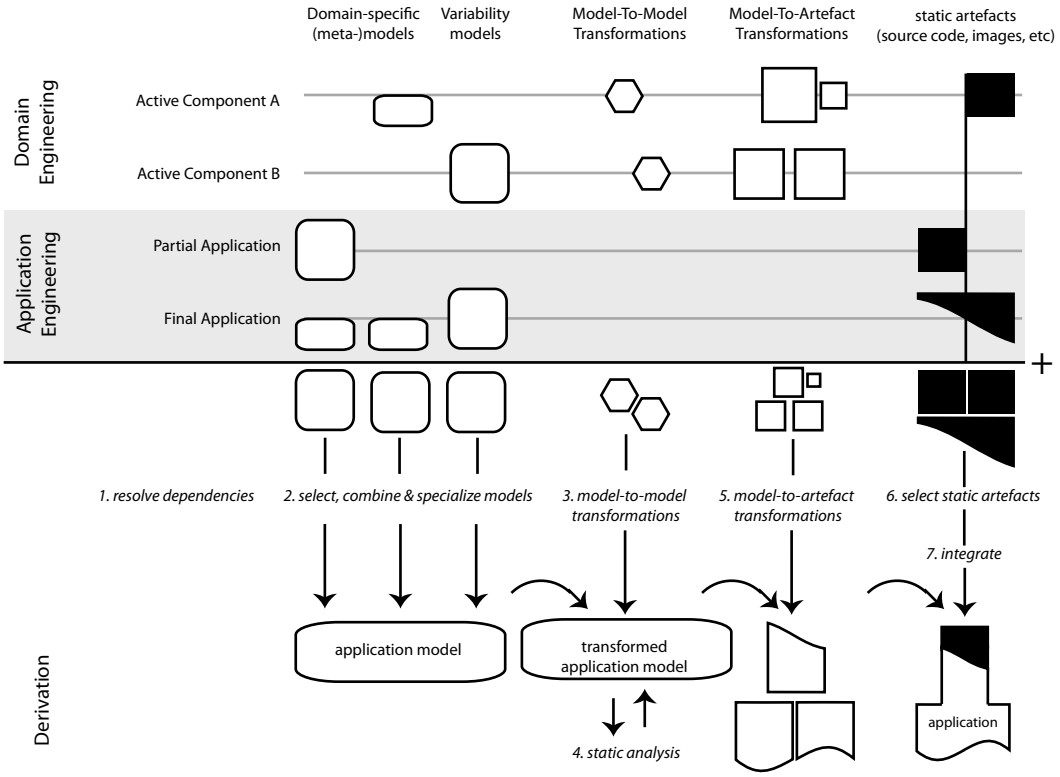


Figure 3: SPL Infrastructure.

only during compilation or at runtime.

Static artefacts do not expose any internal variability and can only be integrated in the software product by including the whole artefact. Static artefacts can be excluded from integration depending on the application model, however, as is the case with all software artefacts.

Applications can be reduced to component composition and variability decisions in models. The reduction of software products to these two specifications means that specification and implementation are decoupled. Applications are not limited to these specifications however. It is allowed to use other software artefact types in the software product, such as source code files. These artefacts must be application specific and are never intended for reuse. By allowing other software artefacts we do not bound the application developer to only the variability that is exposed by the SPL.

4.2 Behavior

An Active Component must implement a well defined interface that is specified by the SPLI. This interface includes behavioral operations that are invoked during derivation in order to adapt and integrate the component into the application. The derivation process itself is merely an orchestrator that invokes the components and integrates the output into the application. During derivation the following tasks are performed:

1. Resolve all Active Components
2. Select, combine and specialize models
3. Execute model-to-model transformations
4. Static analysis on application model
5. Execute model-to-artefact transformations
6. Select static software artefacts
7. Integrate software artefacts into the application
8. Compile

References to active components are defined in the software product's component composition. All active components are resolved to ensure that all selected active components are available for construction.

The second step is that all models, both from the components and software product, are merged into a single monolithic application model. The merging takes into account the specialization of models. The output is a single application model that specifies the application. The application model can contain models from different meta-models, i.e. many type of models are combined. The application model is the input for all subsequent selections and transformations. The SPLI dictates that all models are merged into a single model before executing the steps that can generate derived artefacts. It is assumed that the application model is a coherent unit and should be the same for all components that act upon it. Indeed, it is this merging of the models that allows components, in combination with generative transformations, to adapt to a very heterogeneous set of software products in the product population.

In the third step, the global model can be transformed by the active components. The active components can analyze the model and perform modifications. In this step active

components have the possibility to do last-minute updates of the model to satisfy any component's requirements.

After all model transformations have been executed, a static analysis on the model is performed by the active components. This check is to ensure that the model is semantically valid and all the requirements are met. If errors occur in this phase, the construction stops and the developer is informed about the error or warning.

The transformations are executed that take the application model as input and have software artefacts as output. Each active component is given the control to execute the required transformations and return the software artefacts that are generated during transformations.

During selection, each active component is queried which selection of source-code, libraries or other static artefacts are to be integrated in the software product. The component can use the application model as an information source to decide which artefacts are selected. During the selection, the variability model, the standard model that is specified by the framework, is of special importance; the use of variation points and variants allows for easy selection of features or components.

All selected software artefacts are integrated in the final application. This typically consists of the (physical) copying of the static and generated software artefacts. The final step is the compilation of all programming code, both static and generated source code.

Of special interest is how the selection of artefacts and the transformations are decided by the component itself, rather than a central process. The SPLI merely invokes the components and copies the outputs. This allows the component to decide on the artefacts to be transformed and selected until the last possible moment where it can base its decisions based on the global application model. This allows for more complex variability mechanisms that can be used by the component.

5. EVALUATION

The Software Product Line Infrastructure (SPLI) was evaluated using two different methods: a prototype and an experimental software product line.

5.1 Prototype

A prototype of the SPLI has been developed upon which software product lines can be created. The prototype is implemented on the Microsoft .NET platform and takes the form as a Visual Studio 2010 extension. It uses the Microsoft Visualization & Modeling SDK for the modeling environment and the T4 templating technology for model-to-artefact transformations. The operational contracts of components are defined as code interfaces in C#. Each Active Component has to implement this interface before it can be used in the software product line infrastructure. The model-to-artefact transformations are implemented using the Microsoft T4 template technology. T4 provides a flexible templating language that can query and iterate over model elements to generate textual output adapted to the input model. The model-to-model transformations are expressed in C#. An Active Component in the prototype is effectively a dynamic link library (dll) that contains the implementation to the interface as expected by the SPLI. An Active Component can either be a project in Visual Studio that is being referenced in the application, or simply a dll that is referenced. Models and meta-models can be queried and modified by most languages on the .NET platform, e.g. C#. Upon derivation, all models are discovered, merged and specialized. An application also takes on the form as a typical Visual Studio project. Active Components can return warning and error messages which are appended to the error and warning lists in Visual Studio. This gives the developer valuable feedback on errors and warnings that occurred, along with filenames and line numbers, if applicable.

5.2 Experimental Software Product Line

An experimental software product line was built on our prototype to evaluate the design goals of the SPLI. The software product was inspired by one of the main software products at the company that drove the research. The experimental SPL was developed for research purposes and implemented only a small subset of the features of the company's software products. The applications developed on the experimental software product line provide Create, Read, Update, Delete (CRUD) functionality for a Customer Relationship Management (CRM) system.

The software product line consists of six Active Components, one partial software product and two final software products. The active components are: 1) Data Access Layer, 2) Business Logic Layer, 3) Ribbon Meta-Model, 4) Business Entity Meta-Model, 5) WPF Ribbon and 6) Web Ribbon.

To demonstrate step-wise refinements of software products there are two final applications that share the same partial CRM application. The partial CRM application captures common CRM system design in Business Entity models and Ribbon user interface models. The two final applications refine this partial CRM application by providing two different interfaces and specialize the business entity models, i.e. adding and changing business entities. One application has an interface build upon Windows Presentation Foundation (WPF), i.e. a desktop application. The other application has a web interface and is build upon ASP.NET. Both provide a ribbon user interface that can be seen in for example Word 2007. The difference in implementation of the user interface, WPF versus web based, is significant because it would not suffice to capture the variability between a web interface and a windows client interface in only a variability model. This demonstrates the use of a bottom-up approach with components instead of pure variation.

The business entities are defined in a Business Entity models which is quite similar to an Entity-Relationship Diagram (ERD) and the user interface of the application is expressed in a domain-specific interface model. The meta-models of both the Business Entity Model and the Interface Model are defined in Active Components and are part of the experimental SPL, not of the infrastructure itself; they are developed solely for the experimental software products. A variability model was specified in the Active Component that implements the WPF ribbon. It contained variation points that determined which ribbon tabs were visible.

6. RESULTS AND DISCUSSION

The prototype was developed to evaluate the feasibility of the SPLI. Although being a prototype, all essential design elements of the SPLI were implemented and behaved as intended by the design. The development of the prototype took about one month for an experienced developer. It demonstrates that the software product line infrastructure

we propose can be developed using current technology and tools.

The experimental software product line is admittedly quite small; the number of components (6), partial applications (1) and final applications (2) does not represent a significant product population. A more encompassing software product line would better show the possibility of developing more diverse software products. However, the experimental SPL does demonstrate how two very different software products that share common functionality but with different user interfaces, can be developed on a single software product line. The variability between the two interfaces, a web-interface and a windows-interface, is such that just variability within a component would not suffice. In this the strength of composition shows: composing components are a way of feature selection and thus specifying variations. An architecture-centric SPL approach would have to specify all this variability into a single architecture. The experimental software product line furthermore demonstrated the possibility of step-wise refinements of applications by defining a partial application that defined common system design that was refined in two final applications.

The lack of a central product architecture is also one of its weaknesses. Because all components agree to an implicit product architecture there is no central authority to check whether all components adhere to this architecture. The final integration and compilation phase would discover real structural architectural mismatches, but there could be more subtle mismatches that are not uncovered by compilation alone. We also recognize that a downside of using transformations is the lack of traceability of variability. Transformations can potentially be very rich in the mapping between variability input and the output. It is therefore hard for the application engineer to trace the impact of variability decisions. As the architecture cannot be enforced by the infrastructure, it will have to be enforced using other mechanisms. A related disadvantage is that components are invoked during the derivation and can base their transformations and integration decisions on both the application model and all other runtime information that a GPL can offer. This makes the impact of variability decisions even less transparent. The trade-off here is between flexibility and adaptiveness on the one hand and formality and rigidity on the other hand.

7. RELATED WORK

The use of software components to support the development of software product populations is inspired by Ommering and Bosch [19] who see components as a vital way to broaden the product scope. The divide and conquer strategy towards application design in the SPLI by using partial domain-specific models is valued by Warmer and Kleppe [24] who state that partial models are a good way of maintaining complexity. This is in contrast with big monolithic UML models which can be hard to maintain. They recognize that models can be treated in the same way as regular source code to a large extent; an approach that has been adopted in our design as well. However, they only do direct transformations between partial models and source code. In the SPLI proposed here, all partial models are combined into one big model prior to transformations. Model specialization for UML, rather than domain-specific models, has been researched by Apel et al. who refer to it as model superimposition [1].

The autonomity and adaptiveness of components is inspired by active libraries as coined by Czarnecki and Eisenecker [11]. Active libraries also guide the user of the library, hence active, and adapts to the requirements at hand. Their research focuses on a single GPL (C++) however. In contrast, Active Components convey multiple type of artefacts. Another difference is that active libraries adapt during compilation whereas Active Components adapt during derivation, i.e. before compilation.

The idea of step-wise refinements of system design can be traced back to GenVoca [5] which is a model for hierarchical software system design and construction. It expresses an individual software system as an algebraic equation. Batory et al. introduced Algebraic Hierarchical Equations for Application Design, or AHEAD, that works not only for individual programs but can synthesize multiple programs, including noncode representations [6]. The SPLI can be seen as a specialization of AHEAD as it is designed with model-driven software product lines in mind. AHEAD is a very generic method and as such it makes no difference between artefact types nor does it define the composition operators that work on these artefact types. In the SPLI these gaps are filled in by defining structural and behavioral aspects of domain-specific models. Domain-specific models contain application design which is assumed to be cross-cutting features. E.g., An entity model can affect both a Data Access Layer feature as well as a GUI Layer feature. The SPLI dictates, in contrast with AHEAD, that all models are composed first into a single model, prior to intra-component derivations, i.e. model-to-artefact transformations. A similar effect could be achieved in AHEAD, using the Origami matrix, as explained in [4], however we opted to enforce this decision in our SPLI as we believe that this behavior is well suited for model-driven software product lines where it is to be expected that a single composed application model must be the input for the derivations. Other differences between the SPLI and AHEAD are the explicit notion of application and domain engineering and the explicit notion of a variability model. In AHEAD, whole features are composed at once, whereas in the SPLI variability can be achieved within Active Components by using the variability model. Similar is the polymorphic behavior of composition operators: in the SPLI Active Components are empowered with this task and they control the derivation of artefacts.

The SPLE approach as presented by Voelter and Groher [23] uses models combined with aspects for cross-cutting concerns. Their notion of meta product lines, a product line of product line architectures, can be seen as another way to look at specifying software products for which we use partial applications. Our SPLI relies more on step-wise refinements for product specification. Also the notion of model specialization and merging before product derivation is not mentioned. Their notion of library components with a combination of models and generators is the closest match to our Active Components. Their approach uses an explicit architecture, which is lacking in our approach.

An approach similar to the SPLI has been taken by Childs et al. [8] with their Cadena platform, which is a 'a highly adaptive type-centric modeling framework with robust, flexible and extensible tool support'. Although there are similarities, their focus is more on specifying component interconnectivity using models, rather than using models as a way to capture application design.

8. CONCLUSIONS

The results of the prototype and experimental software product line are positive towards the feasibility and utility of the designed Software Product Line Infrastructure (SPLI). Software efforts reuse in software product populations is improved in two ways: 1) an Active Component can be reused in the wider software product spectrum as variability is increased by using domain-specific models to express variability and executing model-to-artefact transformations to derive implementations and 2) application design is reused by using model specialization and allowing step-wise refinements of applications.

For future research we intend to perform a case study at an ISV with a larger software product line. The research will emphasize the scalability of the SPLI. More specifically we will focus on the adaptiveness of Active Components, software integrity in larger component compositions and the role of a light-weight top-down software product line architecture. We expect that these aspects are of special interest when a bottom-up approach is taken in a software product line.

9. REFERENCES

[1] S. Apel, F. Janda, S. Trujillo, and C. Kästner. Model superimposition in software product lines. In *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 4–19, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] T. Asikainen, T. Männistö, and T. Soininen. Kumbang: A domain ontology for modelling variability in software product families. *Adv. Eng. Inform.*, 21(1):23–40, 2007.

[3] F. Bachmann, M. Goedicke, J. Leite, R. Nord, K. Pohl, B. Ramesh, and A. Vilbig. A meta-model for representing variability in product family development. In *Software Product-Family Engineering*, pages 66–80. 2004.

[4] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 48–57, New York, NY, USA, 2003. ACM.

[5] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. Softw. Eng. Methodol.*, 1(4):355–398, 1992.

[6] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering*, 30:355–371, 2004.

[7] T. Bures, P. Hnetynka, and F. Plasil. Sofa 2.0: Balancing advanced features in a hierarchical component model. In *SERA '06: Proceedings of the Fourth International Conference on Software Engineering Research, Management and Applications*, pages 40–48, Washington, DC, USA, 2006. IEEE Computer Society.

[8] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. Calm and cadena: Metamodeling for component-based product-line development. *Computer*, 39(2):42, 2006.

[9] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Professional, 3rd edition, August 2001.

[10] K. Czarnecki. Overview of generative software development. pages 313–328, 2004.

[11] K. Czarnecki, U. Eisenecker, and K. Czarnecki. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, June 2000.

[12] K. Czarnecki, S. Helsen, and U. Eisenecker. Staged configuration through specialization and multilevel configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005.

[13] S. Deelstra, M. Sinnema, J. van Gurp, and J. Bosh. Model driven architecture as approach to manage variability in software product families. Technical report, University of Twente, 2003.

[14] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, August 2004.

[15] S. Kent. Model driven engineering. In *IFM '02: Proceedings of the Third International Conference on Integrated Formal Methods*, pages 286–298, London, UK, 2002. Springer-Verlag.

[16] C. W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, 1992.

[17] W. C. Lim. Effects of reuse on quality, productivity, and economics. *IEEE Softw.*, 11(5):23–30, 1994.

[18] R. C. v. Ommering. Beyond product families: Building a product population? In *IW-SAPF-3: Proceedings of the International Workshop on Software Architectures for Product Families*, pages 187–198, London, UK, 2000. Springer-Verlag.

[19] R. C. v. Ommering and J. Bosch. Widening the scope of software product lines - from variation to composition. In *SPLC 2: Proceedings of the Second International Conference on Software Product Lines*, pages 328–347, London, UK, 2002. Springer-Verlag.

[20] K. Pohl, G. Böckle, and F. J. v. d. Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[21] M. Sinnema and S. Deelstra. Industrial validation of covamof. *J. Syst. Softw.*, 81(4):584–600, 2008.

[22] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.

[23] M. Voelter and I. Groher. Product line implementation using aspect-oriented and model-driven software development. *Software Product Line Conference, International*, 0:233–242, 2007.

[24] J. Warmer and A. Kleppe. Building a flexible software factory using partial domain specific models. In *Sixth OOPSLA Workshop on Domain-Specific Modeling (DSM'06)*, pages 15–22, Jyvaskyla, October 2006. University of Jyvaskyla.

[25] D. L. Webber and H. Gomaa. Modeling variability in software product lines with the variation point model. *Sci. Comput. Program.*, 53(3):305–331, 2004.