Abstract Delta Modeling: My Research Plan*

Michiel Helvensteijn
CWI, Science Park 123
1098XG, Amsterdam
the Netherlands

michiel.helvensteijn@cwi.nl
http://www.mhelvens.net

ABSTRACT

Software product lines are sets of software programs with well defined commonalities and variabilities that are distinguished by which features they support. There is need of a way to organize the underlying code to clearly link features on the feature modeling level to code artifacts on the implementation level, without code duplication or overspecification, so we can support automated product derivation. Existing approaches are still lacking in one way or another. My answer to this problem is delta modeling. My thesis will approach delta modeling from an abstract algebraic perspective called Abstract Delta Modeling. It will give a thorough formal treatment of the subject and extend it in several directions. A workflow for building a product line from scratch, a way to model dynamic product lines as well as plenty of practical examples and case studies.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architectures; F.3.3 [**Mathematical Logic and Formal Languages**]: Mathematical Logic

General Terms

Theory, Languages, Algorithms, Performance

Keywords

Product lines, delta modeling, phd thesis, development workflow, dynamic product lines, modal logic, type systems

1. INTRODUCTION AND MOTIVATION

Code duplication leads to reduced maintainability. In software product line engineering, which is concerned with maintaining a potentially large set of related software systems, this is an urgent problem. Several solutions have been suggested to this problem, but each has significant drawbacks. In my thesis I will propose the approach of Abstract Delta Modeling, which aims to make building and maintaining product lines more intuitive, more suitable for concurrent and isolated development as well as less susceptible to code duplication and overspecification.

*This work is partially supported by the EU project FP7-231620 HATS: Highly Adaptable and Trustworthy Software using Formal Models (http://www.hats-project.eu)

SPLC - Vol. II September 02 - 07 2012, Salvador, Brazil

1.1 Basic Terminology

A *software product line (SPL)* (or *software family*) is a set of software systems, called *software products*, with well-defined commonalities and variabilities [6, 24]. In software product line engineering, SPLs are developed by structured reuse in order to reduce time to market and to increase product quality. *Automated product derivation* generates individual products from the product line artifacts by a mechanical process which requires no human intervention by virtue of a sufficiently expressive code base.

Different software products are distinguished from each other by which *features* they provide. Features can be described as designated product characteristics or increments of product functionality [1]. A product is uniquely identified by a *feature configuration*, i.e., a valid selection of features. The different feature configurations that are supported in an SPL is expressed by a *feature model* [14, 30]. On the feature model level, features are merely labels [7]. To mechanically derive a product for a particular feature configuration, the code base has to be designed with a clear link between features and code. It is important that all possible products can be generated from a trivial composition of this code.

1.2 Existing Approaches

In general, approaches facilitating automated product derivation for SPLs can be classified in two main directions [16]. The first direction consists of *annotative approaches* such as conditional compilation, frames [32] or Colored Featherweight Java (CFJ) [15]. These approaches annotate the complete set of product line code with feature labels so that all irrelevant parts can be removed for generation of a specific product. The disadvantage is that these approaches generally lack a separation of concerns. The annotated code belonging to a specific feature(set) can potentially be spread across the entire code-base, making it difficult to maintain. The upside of annotative approaches is that variability can be very fine-grained. Single statements or even expressions can be conditional on some feature. Nonetheless, I am more interested in the second direction.

The second direction consists of compositional approaches which associate product fragments to features. Based on feature selection, those product fragments are then assembled to implement the product for a particular feature configu-

ration. Two prominent examples (from before I began my research) are AHEAD [1], in which a product is built by stepwise refinement of a base module with a sequence of feature modules, and delta modeling [28, 26], which worked similarly, but with a many-to-many relation between features and code fragments, rather than a one-to-one relation as in AHEAD. Both had their shortcomings, which I will now briefly discuss.

AHEAD introduced the concept of *feature module*, which consists of the code that implements a specific feature when it is applied to the base module. These feature modules are then put into a manually determined linear order, used to generate products with more than one feature. Feature modules later in that linear order can overwrite the code of earlier modules, in order to make the features work well together, so this has to be taken into account in their development. The drawback of forcing conceptually independent features into a linear order is that they can no longer be developed independently, each feature implementation depending on the one before. While this may be useful for subfeatures and other closely related features, a linear order overspecifies the relation between features, so one feature module can unintentionally and silently overwrite the code of another, leading to bugs that are hard to detect. An automated error checker cannot distinguish between intentional and unintentional overwriting of code, so cannot warn you.

Originally, the *delta model* of a product line consisted of a single core and a set of *product deltas* [28, 26]. These deltas look like feature modules, but can be associated with more than one feature at the same time. The set of feature configurations a delta is associated with is expressed through its *application condition*. So combinations of features could be implemented without overspecification. However, *conflicts* between deltas applicable for the same feature configuration were prohibited. Instead of using a linear order, delta modeling was at the other end of the spectrum, and no two deltas could be ordered at all. To express all possible products, an additional delta covering the combination of the conflicting deltas had to be created, requiring fine-grained exclusive application conditions and leading to code duplication.

Some related approaches in programming are *traits* [8, 2] and *mixins* [29]. However, traits and mixins are 'pulled in' by a class, rather than 'pushed in' from the outside. They impose the strict requirement that the core product should already know about all possible features that could be implemented on top of it. Such an approach would lack modularity and scalability. 'Pushing in' code from the outside is sometimes called *invasive composition*, and it makes sure that there is no predefined limit to the functionality that can be implemented on top of a core product.

One approach that does allow 'pushing in' of code is *aspect oriented programming (AOP)* [18, 31, 20, 23, 22]. It also allows relatively fine-grained modifications. However, something lacked by aspects as well as traits and mixins is the ability to add and remove methods, classes and modules. Being restricted to modifying on a sub-method level is quite limiting. Also, none of those three techniques have an inherent link to features from the feature model.

1.3 Abstract Delta Modeling

The approach I propose is, at its core, a mix of delta modeling and AHEAD. A compositional approach in which the deltas / feature modules may be put in an arbitrary partial order. As it resembles delta modeling the most, I decided to work together with Ina Schaefer, who has previously authored papers on delta modeling, in order to improve and extend the delta modeling approach. A partial order is a natural fit for this sort of model, as it allows the developer to order related deltas, so they can overwrite each others code, and leave unrelated deltas unordered, so the developer can be warned when they contain conflicting implementations. The partial order also allows for a novel way to resolve such conflicts. The developer can choose to order two conflicting deltas, so one of them 'wins'. But this is often not appropriate, as two features with conflicting implementations can still be conceptually unrelated to each other. This is called the *optional feature problem* [17]. One way to solve such a problem is what is referred to in [17] as a *derivative module*. The partial order supports derivative modules by using deltas that are greater in the partial order than the two conflicting deltas, developed such that they make the two conflicting deltas work together in the way intended. We call them *conflict resolving deltas*.

The first paper written about this form of delta modeling is Abstract Delta Modeling (ADM) [4], which approaches the topic from an abstract algebraic perspective. Products and deltas are no longer strictly about a specific programming language, or even about software at all, making ADM very widely applicable. Being a formal description of deltas, we were able to provide mathematically rigorous proofs of several desirable properties, such as program line unambiguity, i.e., that for every feature configuration, a single unambiguous product can be generated.

ADM will form the core of my PhD thesis. But the formalism will be extended and explored in several directions. ADM can be seen as a design pattern that can be put to different, sometimes unexpected, purposes, such as profile management on a modern mobile device. I explore different workflows intended to develop product lines with ADM, ways to model dynamic product lines, as well as a modal logic in order to reason about the semantics of deltas. There are also plans to explore type systems for delta modeling.

Concretizations of ADM have already been used by the HATS project [10]. The Delta Modeling Workflow (DMW) has been applied to an industrial scale case-study. And many smaller examples have also been created to validate the applicability of ADM.

This research plan is structured as follows: Section 2 will explain the key research questions for my thesis, grouped by direction of research. Where available, it also gives preliminary answers to those questions, as well as examples and figures to illustrate the different research topics. Section 3 gives a brief view on the research methodology I use. Finally, Section 4 gives a full overview of past, current and future work, as well as detailed plans for my thesis.

2. RESEARCH QUESTIONS & ANSWERS

I group various important research questions by the different research topics that have branched from ADM, starting with ADM itself.

Each following subsection will give a small summary of the research topic, a list of key research questions and, where available, a list of preliminary answers to those questions, possibly from my own published or submitted papers.

2.1 Abstract Delta Modeling

Abstract Delta Modeling shows how to model product lines using deltas. The abstract formalism gives a functional meaning to features from the feature model and provides

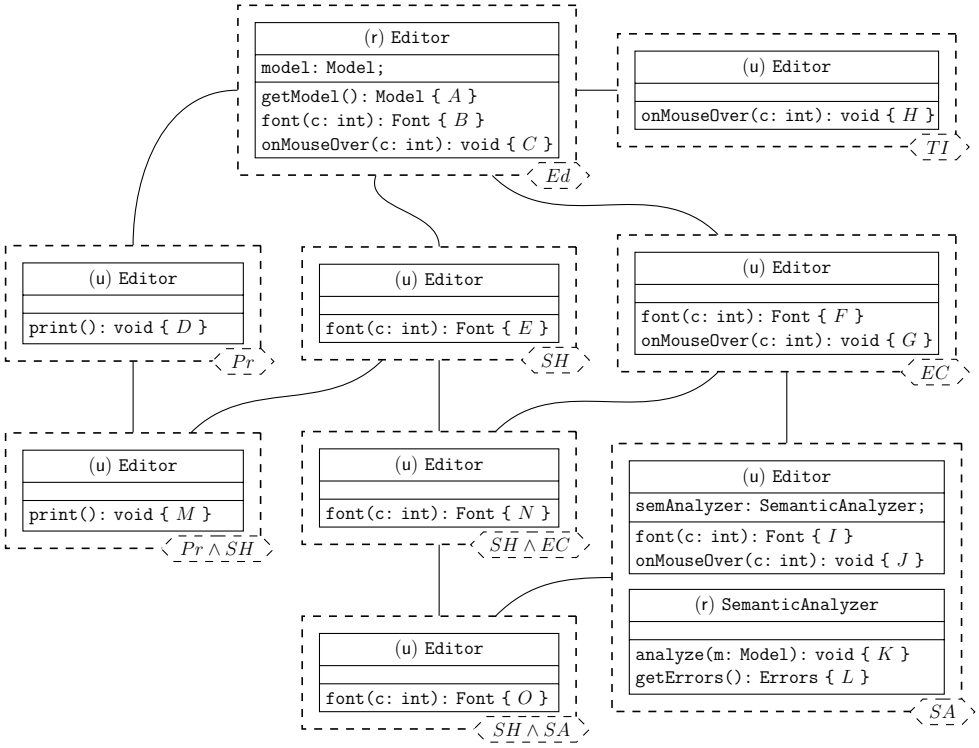

Figure 1: Graphical representation of the delta model for the Editor product line. The dashed boxes represent deltas. They are decorated with their application conditions, represented by a propositional formulas, and placed in a partial order. They contain class replacements (r) and updates (u) displayed in UML.

a novel mechanism for resolving implementation conflicts. This mechanism is first explained for single-product delta models and is then lifted to the level of full product lines. Abstract Delta Modeling, as well as the papers with that title [4, 5], pose several research questions:

1. How can the intuitive idea of a delta model and product line be formally specified, so we can perform rigorous analysis and proof?

2. How can we formally define conflicts and resolution of conflicts in our model?

3. Can we prove that a product line with no unresolved conflicts can generate a unique product for each feature configuration?

4. How does ADM compare to existing approaches for modeling software product lines?

Abstract Delta Modeling has been thoroughly treated in a paper published in the proceedings of GPCE '10 [4] and an extension of it has been accepted to appear in a special issue of MSCS [5]. The answers provided to the previous questions are roughly as follows:

1. Deltas, their composition and the empty delta are seen as a monoid $(\mathcal{D}, \cdot, \epsilon)$ acting on the left of a set of products $\mathcal{P}$. We call the 5-tuple $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$ a *deltoid*. A *delta model* is a tuple $(D, \prec)$ with $D \subseteq \mathcal{D}$ and $\prec$ a partial order over D. $\prec$ restricts the order in which the deltas can be applied. A *product line* is represented as $(\mathcal{F}, \Phi, c, D, \prec, \gamma)$ where $\mathcal{F}$ is a set of features, Φ is the set of valid feature configurations over those features (the feature model), c is the core product to which the deltas in D are applied, $(D, \prec)$ is an underlying delta model and γ is a function mapping each delta to its application condition: the set of feature configurations to which it is applicable. Figure 1 shows the example product line from [5], which has concrete object oriented setting, in a graphical representation.

2. Two deltas $x, y \in D$ are in conflict if they are non-commuting: $y \cdot x \neq x \cdot y$ and not ordered by $\prec$. A third delta $z \in D$ can resolve the conflict if it is greater than both deltas in $\prec$ and $z \cdot y \cdot x = z \cdot x \cdot y$.

3. The proof appears in [4, 5].

4. We can encode most other approaches in delta modeling. A thorough comparison appears in [4, 5].

2.2 Delta Modeling Workflow

In the vast expressive space of delta modeling, it may not be clear to a developer how to create a product line from scratch. The ADM formalism is descriptive rather than prescriptive. To that end, I proposed the Delta Modeling Workflow (DMW). I show preservation of global unambiguity and completeness in the product lines resulting from this workflow. I also show how the workflow naturally supports concurrent development and how it avoids code duplication and overspecification. The research questions are:

1. What are the useful properties we would like a product line to have?

2. Can we define a systematic workflow to build such product lines while allowing different developers to work independently and in isolation?

3. Given such a workflow, can we prove that it exhibits those properties?

The Delta Modeling Workflow has been treated in a paper published in the proceedings of VaMoS '12 [11], and its use in an industrial scale case study has been described in another paper published in the same proceedings [13], written by Peter Wong, Radu Muschevici and myself. An extension of [11] has been mostly written, and I am in search of an appropriate venue. The answers provided to the previous questions are roughly as follows:

1. A product line should be globally unambiguous, meaning that all possible conflicts inside it have been resolved. It should also be complete, in the sense that all features in the feature model have been appropriately implemented for each feature configuration.

2. The workflow is described in detail in [11]. The flowgraph in Figure 2 gives an overview. Each feature is implemented using a single delta, in the order of the subfeature relation, i.e., base features first, subfeatures later. Then any desired interaction between features is implemented using feature interaction deltas and any implementation conflicts resolved using conflict resolution deltas. The paper describes why the workflow can be followed by different developers independently and in isolation as well as why it preserves the properties listed in the answer to question 1.

3. Proof sketches of those properties appear in [11]. Full formal proofs will appear in its extension.

2.3 Dynamic Delta Modeling

In traditional application engineering a single valid feature configuration is chosen, which doesn't change during the lifetime of the product. However, there are many useful applications for product lines that change their configuration at run time. Dynamic Delta Modeling (DDM) is a new technique for generating efficient dynamic product lines from their static ADM counterparts. The research questions can be listed as follows:

1. What is a dynamic product line?

2. How do we model the behavior of a dynamic product line in the context of ADM?

3. Can a product line modeled with ADM be 'converted' into a dynamic product line? And if so, how?

4. How can we define the 'cost' of a dynamic product line? And can such a dynamic product line be optimized to be less costly?

5. Does such a(n optimized) dynamic product line behave the way we want it to? Can we prove this?

6. What is a good use-case for this kind of dynamic product line?

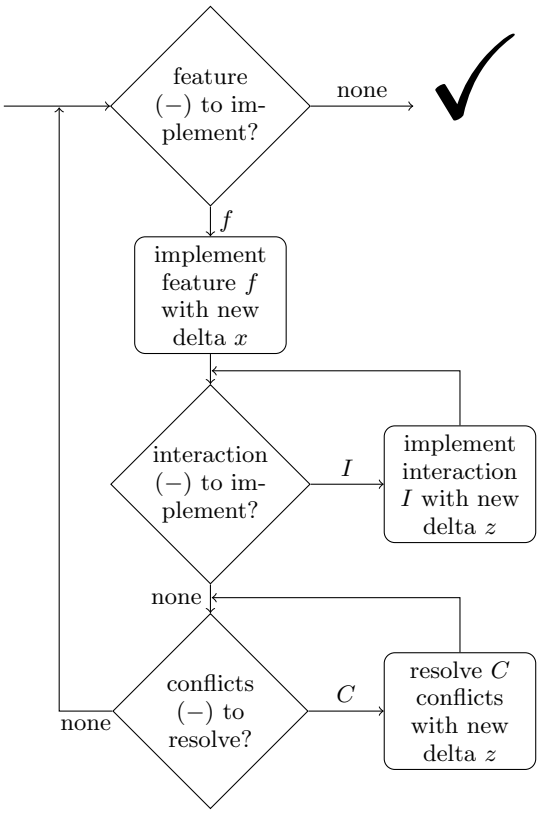

Figure 2: Overview of the Delta Modeling Workflow. After implementing a feature, interaction is implemented and implementation conflicts resolved.

A paper treating Dynamic Dynamic Delta Modeling has been accepted for publication by the DSPL 2012 workshop [12] and will have appeared in the same proceedings as this research plan. The answers provided to the previous questions are roughly as follows:

1. For the purposes of this paper, a dynamic product line is like a static product line, except that the chosen feature configuration can change at runtime. Other definitions of dynamic product lines are discussed in the related work section of the paper.

2. We model the behavior of a dynamic product line with a Mealy Machine, a finite automaton with input and output on each transition. The input corresponds to a feature that has been turned on or off and the output corresponds to the delta that can be used to bring the current product up to date. Figure 3 shows such a Mealy Machine for a product line example that is described in the paper.

3. The paper formally describes how a static product line can be converted into a dynamic one.

4. The paper describes the cost of monitoring a specific feature for change. Different features will generally have different cost. The paper describes that to optimize a dynamic product line, you can remove costly tranitions from the Mealy Machine, as long as certain reachability conditions remain satisfied.

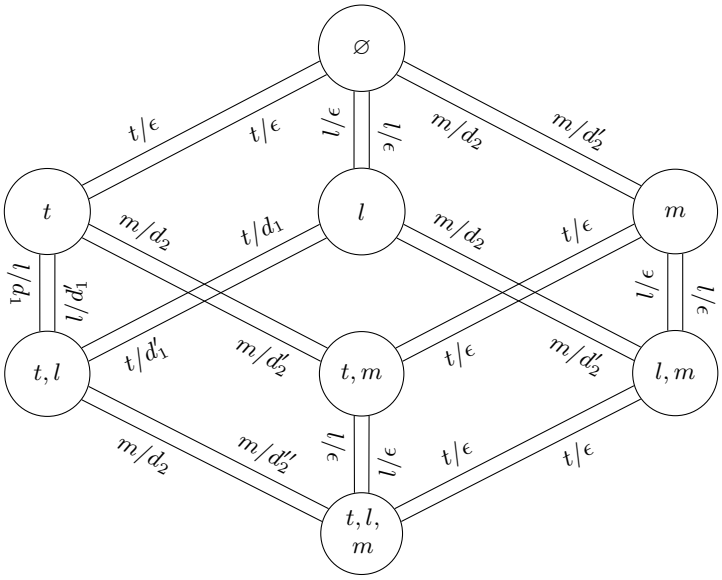

Figure 3: An example Dynamic Product Line as a Mealy Machine. Each state corresponds to a valid feature configuration. Each transition is triggered by a feature being turned on or off, and returns a delta which can be used to bring the current product up to date. Some transitions can be optimized away to reduce cost. [12]

5. Proof of this appears in the paper.

6. The paper describes Delta Profiles, an Android application which regulates the settings on your smartphone or tablet. Through a convenient user interface, the user can design deltas that modify the settings on the phone based on various conditions, such as time, gps location, connected peripherals, scheduled appointments and more.

2.4 A Modal Logic for Delta Modeling

Especially in light of the Delta Modeling Workflow (Section 2.2), there is need of a logic in which to express certain properties about the behavior of deltas and the semantics of features. So far, features have been seen as labels, but those labels actually represent some functionality we'd like a product to have. Does a delta implement a specific feature? Under which conditions? Frank de Boer, Joost Winter and myself have introduced the modal logic $\mathbf{K\Delta}$, in order to be able to reason about such notions more easily. The research questions can be listed as follows:

1. What does a Kripke frame that embodies the semantics of deltas and delta models look like?

2. What should the basic axioms of the logic $\mathbf{K\Delta}$ be?

3. Is the logic (strongly) complete? Can we prove it?

4. On the model level, can we prove completeness of a base theory containing only simple facts about individual deltas?

5. What can this logic be used for in practice?

A paper describing the logic has been accepted for publication by the FMSPLE 2012 workshop [9] and will have appeared in the same proceedings as this research plan. The answers to the above questions are roughly as follows:

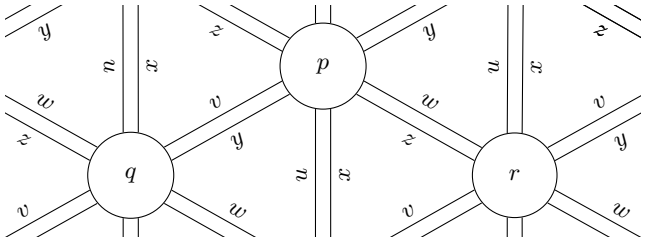

Figure 4: Example view of a delta frame with products p, q, r and deltas u, v, w, x, y, z currently visible

1. Products are worlds in this frame and deltas are relations (Figure 4). Compound deltas, such as composition, union and partial ordering (such as in delta models), form compound relations on the frame. Models based on this frame make semantic judgments about which features are implemented by specific products.

2. The axioms are the classic **K**, **Dual** (as described in [3]), as well as axiom schemata called $\mathbf{\Delta}$ describing the meaning of compound delta modalities.

3. The logic is strongly complete, as proved in the paper by a reduction to the completeness of **K** over the class of all frames.

4. We were able to prove relative completeness under the restriction of weakest precondition expressability. For more details, I refer to the paper.

5. The logic will be used in the extension of the Delta Modeling Workflow paper (as described in Section 2.2) in order to describe the process and the proofs more elegantly and succinctly.

2.5 Row Typing for Delta Modeling

Many concrete implementations of delta models fall under the semi-abstract realm of nested key-value pairs. Modules, classes, methods and fields in object oriented programming have names and implementations (values). Their names form a handle by which deltas can add, remove and modify them. The level at which elements carry a name defines how fine-grained those modifications can be. Within this semi-abstract realm, I want to introduce a type system to make sure elements are not added when they are already present or removed when they are already absent, together with Dave Clarke and Michael Lienhard, which have already done some preliminary work on this topic, applying row typing (normally used for structures in programming languages) to delta modeling [19, 21]. We will try to lift the typesystem to the level of product lines (whereas right now it has only been done without taking features into account), and try to use it as an alternate way to detect conflicts. The research questions are:

1. How can we formally define the 'semi-abstract realm of nested key-value pairs'?
2. How is the typesystem formally defined?
3. What are the facts we want to be able to prove with this type system?
4. Can we prove those facts using the type-system?
5. Can we prove that there are no unresolved conflicts using the type-system?

Previous work by Dave Clarke and Michael Lienhardt [21] describes 'hard conflicts', unresolvable by their semantics, so we also add the following question:

6. How can we reformulate those 'hard conflicts' so we can resolve them anyway?

We are now writing this paper but, as of writing this research plan, do not yet have answers to these questions.

3. RESEARCH METHODOLOGY

The research in my thesis will be largely formal and mathematical, and as such, will depend largely on definitions, theorems and mathematically rigorous proofs. Many of the results will also be theoretical. This is a consequence of working in a purely abstract setting.

However, the theory has been practically applied and validated in several ways. I am funded by the HATS project [10], which focusses on software product lines. The delta modeling approach has been widely accepted and is now a cornerstone of the HATS methodology.

More specifically I have, with the help of Peter Wong and Radu Muschevici, implemented one of the HATS industrial scale case studies - the Fredhopper Access Server [13] - using a concretization of ADM. Deltas have been added to the ABS language, and support for them has been added to the Eclipse IDE. We also added *parametrized deltas* to support more advanced aspects of the Delta Modeling Workflow.

Delta Profiles, the Android application used to illustrate Dynamic Delta Modeling, will be a major practical application of an otherwise theoretical contribution.

I am now working on an abstract software framework to support ADM principles for any instantiated programming language. This will likely take the form of an Eclipse plugin and will be another important practical contribution.

Finally, in the papers that will make up my thesis (Section 4), there are concrete examples to illustrate the theory.

A possible threat to the validity of my theory is that ADM has so far not been used on an actual industrial software product line. Also missing right now is a practical comparison of the ADM approach to older approaches such as delta oriented programming [27] or AHEAD [1].

4. WORK PLAN

In this section I present all papers: past, present and future that I have (co)authored or will (co)author for my PhD thesis, as well as the structure and content of the thesis.

4.1 Papers

The following is a list of papers that have been published or accepted. They are identified by their bibliography number:

> [4] Abstract Delta Modeling,
> GPCE 2010
>
> [5] Abstract Delta Modeling,
> MSCS 2012 Special Issue
>
> [9] A Modal Logic for Abstract Delta Modeling,
> FMSPLE 2012
>
> [11] Delta Modeling Workflow,
> VaMoS 2012
>
> [12] Dynamic Delta Modeling,
> DSPL 2012
>
> [13] Delta Modeling in Practice,
> VaMoS 2012
>
> [25] HATS Abstract Behavioral Specification:
> The Architectural View,
> FMCO 2011 Postproceedings

The following is a list of papers I am currently working on, possibly with coauthors:

> [⋆1] Row Types for Product Lines in Delta Modeling
>
> [⋆2] Delta Modeling Workflow revisited
>
> [⋆3] Delta Profiles (tool paper)

And these are papers I would still like to write (be)for(e) my thesis:

> [⋆4] A Concrete Feature Satisfaction Relation
>
> [⋆5] An Abstract Framework for Delta Modeling
>
> [⋆6] Delta Modeling Modal Logic revisited

[4] and [5] present Abstract Delta Modeling, as briefly discussed in Section 2.1.

[11] and [13] were presented at VaMoS 2012, and introduce the Delta Modeling Workflow, as briefly discussed in

Section 2.2, as well as its application to an industrial scale case study: the Fredhopper Access Server.

[25] describes the architectural view of the ABS language created by the HATS project, which includes a description of the Delta Modeling Workflow.

[12] describes Dynamic Delta Modeling, as briefly discussed in Section 2.3.

[9] describes the modal logic $\mathbf{K\Delta}$ as briefly discussed in Section 2.4.

[⋆1] extends the original work on a type-system for delta modeling by Michael Lienhardt and Dave Clarke, briefly described in Section 2.5. I am working with them on this.

[⋆2] describes an alternate formalism for the Delta Modeling Workflow, with a more detailed analysis of locality and full formal proofs using $\mathbf{K\Delta}$. I plan to extend it further and submit it to a journal, most likely IEEE Transactions on Software Engineering.

[⋆3] will be a tool paper describing the Android application which served as practical example for [12]. Figure 3 actually shows an example related to that application.

[⋆4] should describe a concrete use of the feature satisfaction relation described in [9], [11], [13] and [⋆2]. A possible direction is to use petri-nets to demonstrate feature specifications in the form of firing patterns.

[⋆5] will describe my efforts to implement Abstract Delta Modeling in an abstract framework. It will likely consist of an Eclipse plugin which can apply the principles of abstract delta modeling to any instantiated programming language.

[⋆6] was proposed by my thesis supervisor, Frank de Boer. He suggested we should write a journal version of [9].

4.2 Reading Order

There is a partial order between the topics of Section 2, which will also be a recommended reading order of my thesis:

- At the core will be Abstract Delta Modeling [4], [5]

Then there are four branches which could be read more or less independently:

- Delta Modeling Workflow [11], [13], [25], [⋆2]
- Dynamic Delta Modeling [12], [⋆3]
- The Modal Logic $\mathbf{K\Delta}$ [9], [⋆6]
- Row Types for Delta Modeling [⋆1]

[⋆4] will tie together the Delta Modeling Workflow with the Modal Logic $\mathbf{K\Delta}$. [⋆5] will be used for practical examples throughout the thesis. Also, later sections of the Delta Modeling Workflow chapter will depend on the modal logic $\mathbf{K\Delta}$.

4.3 Abstraction Level

The thesis will be 'layered' by abstraction level:

- [4], [5], [9], [11], [12] and [⋆2] and [⋆6] discuss delta modeling on a completely abstract level.
- [⋆1] will take place in the semi-abstract realm of key-value pairs.
- [13], [25], [⋆3], [⋆4] and [⋆5] are examples of delta modeling in a concrete domain (e.g. object oriented software, profile managers, petri-nets).

4.4 Perspective

A useful distinction to make when looking at these topics is the perspective they explore. The topics can be divided into two different perspectives, which may be seen as Kripke Frames:

- The frame $(D, \prec)$, which relates deltas in a delta model to each other by a developer-defined partial order. Figure 1 shows this perspective, and it is prevalent in [4], [5], [11], [13], [25], [⋆1], [⋆2] and [⋆5].
- The frame $(\mathcal{P}, \mathcal{D})$, which relates the possible products (both in and outside any particular product line) to each other by the set of deltas which can transform one product into another. Figures 3 and 4 show this perspective, and it is prevalent in [9], [12], [⋆3]m, [⋆4] and [⋆6].

4.5 Example Product Lines

In order to illustrate the highly formal theory in my thesis, I will use several example product lines. Some come from publications, some will be created purely for my thesis:

- Firstly, for everything in the abstract layer I would like one concrete example product line to tie everything together. Most likely in an object oriented domain, based on the example in [4] and [5]. I hope to use real code to illustrate the product line, based on my abstract software framework from [⋆5].
- The Fredhopper Access Server case study [13].
- Based on [12] and [⋆3], I will show a family of product lines, which represent rule-sets to govern the profiles on a mobile device.
- I also want to demonstrate the 'thesis product line', which is a product line of versions of the PhD thesis itself. The published version will be the one with all possible 'features' (topics), but it will be generated from a set of deltas which I will write. Shorter versions of my thesis can also be generated and will be downloadable from my website. These shorter versions will be sure not to reference non-existing chapters and such. Humorous effect aside, it has real practical applications. For example, the approach could be used to generate shorter versions of a textbook to perfectly fit a specific syllabus, or to generate documentation belonging to code developed using delta modeling. It will be a great illustration of the versatility of ADM.

4.6 Future Work

In the coming year, I will be working on the papers [⋆1], [⋆2], [⋆3], [⋆4], [⋆5] and [⋆6]. Out of those, [⋆3] and [⋆5] are most important, as they show the practical value of my work, which may still be unclear. I believe that any additional papers I write should explore the existing topics described in this research plan with more depth, rather than explore new topics. Otherwise there may be too great a variety of topics for a coherent thesis. I may or may not have to narrow the scope of my thesis already. In a few months I expect to start writing my thesis. Until then, I hope to get inspiration out of the SPLC 2012 doctoral symposium.

5. REFERENCES

[1] D. S. Batory, J. Sarvela, and A. Rauschmayer. Scaling Step-Wise Refinement. *IEEE Trans. Software Eng.*, 30(6), 2004.

[2] L. Bettini, F. Damiani, and I. Schaefer. Implementing Software Product Lines using Traits. In *Proc. of Object-Oriented Programming Languages and Systems (OOPS), Track of ACM SAC*, 2010.

[3] Patrick Blackburn, Johan F. A. K. van Benthem, and Frank Wolter. *Handbook of Modal Logic, Volume 3 (Studies in Logic and Practical Reasoning)*. Elsevier Science Inc., 2006.

[4] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *Proc. of GPCE*, pages 13–22. ACM, 2010.

[5] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. *Accepted to MSCS special issue*, 2012.

[6] P. Clements and L. Northrop. *Software Product Lines: Practices and Patterns*. Addison Wesley Longman, 2001.

[7] K. Czarnecki and M. Antkiewicz. Mapping Features to Models: A Template Approach Based on Superimposed Variants. In *Conf. on Generative Programming and Component Engineering(GPCE)*, 2005.

[8] S. Ducasse, O. Nierstrasz, N. Schärli, R. Wuyts, and A. Black. Traits: A mechanism for fine-grained reuse. *ACM TOPLAS*, 28(2), 2006.

[9] J. Winter F.S. de Boer, M. Helvensteijn. A Modal Logic for Abstract Delta Modeling. In *Workshop Proceedings of SPLC 2012*, 2012.

[10] R. Hähnle. HATS: Highly Adaptable and Trustworthy Software Using Formal Methods. In *ISoLA (2)*, pages 3–8, 2010.

[11] M. Helvensteijn. Delta Modeling Workflow. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.

[12] M. Helvensteijn. Dynamic Delta Modeling. In *Workshop Proceedings of SPLC 2012*, 2012.

[13] M. Helvensteijn, R. Muschevici, and P.Y.H. Wong. Delta Modeling in Practice, a Fredhopper Case Study. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-intensive Systems, Leipzig, Germany, January 25-27 2012*, ACM International Conference Proceedings Series. ACM, 2012.

[14] K. C. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Carnegie Mellon University Software Engineering Institute, 1990.

[15] C. Kästner and S. Apel. Type-Checking Software Product Lines - A Formal Approach. In *ASE*, pages 258–267. IEEE, 2008.

[16] C. Kästner, S. Apel, and M. Kuhlemann. Granularity in software product lines. In *ICSE*, pages 311–320, 2008.

[17] C. Kästner, S. Apel, S. S. ur Rahman, M. Rosenmüller, D. Batory, and G. Saake. On the impact of the optional feature problem: Analysis and case studies. In *Proc. Int'l Software Product Line Conference (SPLC). SEI*, 2009.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Videira Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *ECOOP*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[19] Michaël Lienhardt and Dave Clarke. Row types for delta-oriented programming. In *Proceedings of the Sixth International Workshop on Variability Modeling of Software-Intensive Systems*, VaMoS '12, pages 121–128, New York, NY, USA, 2012. ACM.

[20] N. Loughran and A. Rashid. Framed aspects: Supporting variability and configurability for AOP. In *ICSR*, volume 3107 of *LNCS*, pages 127–140. Springer, 2004.

[21] D. Clarke M. Lienhardt. Conflict detection in delta-oriented programming. In *Proceedings of 5th International Symposium On Leveraging Applications of Formal Methods, Verification and Validation*, 2012.

[22] M. Mezini and K. Ostermann. Variability management with feature-oriented programming and aspects. In *SIGSOFT FSE*, pages 127–136. ACM, 2004.

[23] N. Noda and T. Kishi. Aspect-Oriented Modeling for Variability Management. In *SPLC*, 2008.

[24] K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.

[25] E. Broch Johnsen M. Lienhardt D. Sangiorgi I. Schaefer P.Y.H. Wong R. Hähnle, M. Helvensteijn. HATS Abstract Behavioral Specification: The Architectural View. In *FMCO 2011 Postproceedings*, 2012.

[26] I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In *Intl. Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2010)*, 2010.

[27] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. Delta-oriented Programming of Software Product Lines. In *SPLC*, volume 6287 of *LNCS*, pages 77–91. Springer, 2010.

[28] I. Schaefer, A. Worret, and A. Poetzsch-Heffter. A Model-Based Framework for Automated Product Derivation. In *Proc. of Workshop in Model-based Approaches for Product Line Engineering (MAPLE 2009)*, 2009.

[29] Y. Smaragdakis and D. S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[30] A. van Deursen and P. Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–18, 2002.

[31] M. Völter and I. Groher. Product Line Implementation using Aspect-Oriented and Model-Driven Software Development. In *SPLC*, pages 233–242, 2007.

[32] H. Zhang and S. Jarzabek. An XVCL-based Approach to Software Product Line Development. In *Software Engineering and Knowledge Engineering*, pages 267–275, 2003.