Delta Modeling in Practice[*]

A Fredhopper Case Study

Michiel Helvensteijn
CWI, Amsterdam, The Netherlands
Leiden University, The Netherlands
michiel.helvensteijn@cwi.nl

Radu Muschevici
Katholieke Universiteit Leuven, Leuven, Belgium
radu.muschevici@cs.kuleuven.be

Peter Y.H. Wong
Fredhopper B.V., Amsterdam, The Netherlands
peter.wong@fredhopper.com

ABSTRACT

Delta modeling is a method for modeling software product lines (SPL), which supports the automated derivation of products. ABS is a recent modeling language and accompanying toolset that implements delta modeling as its core paradigm for developing variable systems. Due to its novelty, delta modeling has so far seen little practical application. However, only practical evaluation can indicate to what extent the delta modeling methodology is suited for the efficient and accurate modeling and implementation of SPLs. This paper reports on the development of an industrial scale product line in ABS following a workflow that guides the application of delta modeling in practice. By following the delta modeling workflow (DMW), we show how conflicting feature functionality can be systematically reconciled, and how DMW guides the implementation towards a globally unambiguous and complete product line. We further explain how this experience has been used to refine the workflow and its support by the ABS language.

1. INTRODUCTION

A *software product line (SPL)* (also known as a *software family*) is a set of software systems, called *software products*, with well-defined commonality and variability [7, 15]. Variability between these systems is expressed by *feature models* [13, 20], which distinguish software products by the features they provide.

A recently proposed method for organizing a code-base in a way that supports automated product derivation is *delta modeling* [16], wherein the code-base is split up into *deltas* that can modify a *core product*. Deltas are annotated with an *application condition*, stating for which sets of features a given delta should be applied.

Clarke et al. [4] put deltas in a partial order in a paper that abstracts away from software and generalizes the concepts of delta model and product line in an abstract algebraic setting, known as *Abstract Delta Modeling (ADM)*. The partial order offers more flexibility in how to compose deltas and how to resolve conflicts between them. *Conflict resolving deltas* were introduced, which can be applied after a conflicting pair of deltas to 'equalize' the two possible orderings between them. That is, it makes them commute again. The absence of unresolved conflicts leads to a single unambiguous product for each feature configuration. ADM also introduced efficient conditions for checking the unambiguity of a product line as a whole.

Current work on ADM concentrates on its practical applicability within SPL engineering. Helvensteijn [11] introduces a development workflow, which we refer to as the Delta Modeling Workflow (DMW). DMW aims to give step-by-step instructions on how to systematically design a product line from scratch using delta modeling, and ensure that it is globally unambiguous and complete by construction.

DMW is described in the same abstract setting as ADM [4]. We recognize that an abstract formalism does not tell you how these techniques would play out in practice. So in this paper we evaluate the DMW by modeling the industrial case study of the Fredhopper Access Server (FAS) product line. FAS, developed by Fredhopper B.V. (www.fredhopper.com), is a distributed service-oriented software system for Internet search and merchandising. In particular we consider FAS's *replication system*, which ensures data consistency across the FAS deployment. The FAS product line is modeled using the *Abstract Behavioral Specification (ABS)* language [12, 5]. ABS is being developed as part of the HATS project [10].

This paper gives an account of modeling an industrial SPL using the ABS language by following the Delta Modeling Workflow. We show how DMW assists in resolving conflicts that arise when features modify the code base in incompatible ways, and how it guides the implementation towards a globally unambiguous and complete product line. We further explain how this experience has been used to refine the workflow and the ABS language to better support systematic SPL development.

The paper is organized as follows. In Section 2, we introduce the development workflow. Section 3 describes the relevant parts of the ABS language and the changes we made to it in order to implement the workflow. The FAS case

---

study is described in Section 4 and its implementation in Section 5. In Section 6 we list the implications of applying this workflow in practice. Section 7 discusses related work. Finally, Section 8 gives a short summary and concludes.

2. DELTA MODELING WORKFLOW

Abstract delta modeling [4] (ADM) describes the possible ways a product line code base can be structured by delta modeling so that it supports automated product derivation. It puts deltas, which can modify (software) products, into a partial order which restricts their application. If delta x comes earlier in this order than y, then x has to be applied before y. If there is no order between x and y, and their changes are incompatible, a third delta can be used to mediate and resolve the conflict. The deltas carry application conditions which specify for which feature configurations they should be applied.

The Delta Modeling Workflow (DMW) [11] guides developers through the process of building a software product line step by step.

We assume that the process starts with a product line specification $(\Psi, \vDash)$. The structural feature model Ψ is a 5-tuple $(B, \multimap\bullet, \multimap\circ, \oplus, \blacktriangleright)$, where B is a set of mandatory base-features, $\multimap\bullet$ is the mandatory subfeature relation, $\multimap\circ$ is the optional subfeature relation, $\oplus$ indicates those features that may not appear together in one product and $\blacktriangleright$ indicates which features require which others. These relations should all be disjoint. Φ is used to refer to the set of feature configurations that are allowed by the feature model. Φ can be trivially deduced from Ψ (which contains more information). The feature satisfaction relation $\vDash$ is a relation between a product p and a set of features F. $p \vDash F$ indicates that p satisfies the specifications of the features in F as well as any desired interaction between the features in F.

Following the workflow should yield a product line implementation $(c, D, \prec, \gamma)$, where c is the core product that the deltas will be applied to. It is usually the empty product, although it is permissable to implement mandatory features in c. D is the set of deltas, $\prec$ is the partial order between these deltas and γ is a function mapping each delta to the set of feature configurations it is applicable for with $\gamma(x) \in \Phi$ for all $x \in D$. We start with $D = \prec = \gamma = \varnothing$. We fill in D, $\prec$ and γ while following the workflow.

The product line resulting from this workflow is guaranteed to be globally unambiguous, meaning that each feature configuration generates a unique product. If certain local guarantees are met, it is also guaranteed to be complete, meaning that each product satisfies the specifications of the features it is supposed to implement.

Figure 1 shows the workflow as a flow-chart. In each iteration a single feature is implemented, while also implementing necessary feature interactions and resolving conflicts that this feature might have introduced. We now explain these steps in more detail.

Feature left to implement?

In this stage, we choose the next feature to implement. The choice is made by following the partial order introduced by the transitive closure of $\multimap\bullet \cup \multimap\circ$. Given a feature diagram (such as the one in Figure 3), we work on it in a topological order from top to bottom, implementing first the base features and then their subfeatures. This is because deltas implementing subfeatures often need to make assumptions
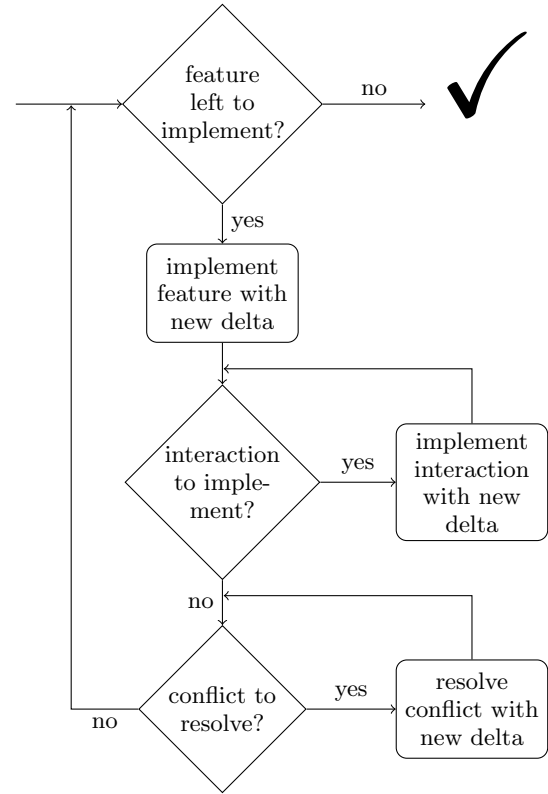

Figure 1: Overview of the development workflow

about – and changes to – the implementation of the base features.

This is the main stage where different developers can work on the product line concurrently and in isolation. Many features can be worked on simultaneously, so long as they are independent. While working on a feature, it is not necessary to consider possible conflicts, since there will be an opportunity to resolve them later in the workflow. (It will help, of course, if code is written in a modular way, which lends itself better to conflict resolution in the future.)

If there are no more features to implement, and for each previously implemented feature, the steps were properly followed, the product line is now finished.

Implement feature with new delta

Having chosen a feature, we need to develop a new delta in D to implement it. This delta has to be applied exactly when its feature is selected, which should be reflected in γ. Its place in the partial order $\prec$ should mirror the feature's place in the feature diagram. So, it should be greater than the delta that implements its superfeature and incomparable to all other deltas currently present. In fact, the deltas that implement features, linked by the transitive reduction of $\prec$, will form a graph that is isomorphic with the feature diagram. D has to be implemented such that it introduces the new feature, but does not remove the functionality that was introduced by superfeatures.

Interaction to implement?

At this point, we need to know if, by introducing a new feature, there are now sets of features that require extra work to make them interact properly. We will implement any such desired interaction in the next step. Different interactions may be implemented concurrently and in isolation.

Implement interaction with new delta

Given a set of features whose interaction we need to implement, we develop a new delta in D to do it. It has to be applied exactly when the interacting features are selected, which should be reflected in γ. It should be greater in the partial order $\prec$ than the deltas that implement the interacting features, as well as the deltas that implement interactions of subsets of the interacting features.

Conflict to resolve?

After implementing our feature and any desired interaction related to it, we now look for any conflicts we might have introduced in this iteration. We have to consider conflicts involving the delta that implemented the feature, all deltas we used for implementing desired interaction and all deltas we used for resolving earlier conflicts in this iteration. Formally, a conflict occurs between two deltas. However, when a there is a set of deltas with many (related) conflicts, we will also want to introduce conflict-resolving deltas for larger sets that can be applied together according to γ, so all combinations are covered. Conflict-resolving deltas for different conflicts may be implemented concurrently and in isolation.

Resolve conflict with new delta

Given a set of deltas whose conflict we need to resolve, we develop a new delta in D to do it. It has to be applied exactly when all conflicting deltas are applied, which should be reflected in γ. It should be greater in the partial order $\prec$ than all conflicting deltas and it should resolve all conflicts between them.

3. ABS LANGUAGE

The Abstract Behavioural Specification Language (ABS) [12, 5] is a concurrent, multi-paradigm modeling language. It combines functional, object-oriented, and concurrent programming. ABS is particularly suited for developing systems with a high degree of variability and supports software product line development through delta-oriented programming [17]. Syntax-wise, ABS resembles standard programming languages like Java. In this section we give an overview of ABS and focus on the language constructs needed for modeling variable systems. For a comprehensive description we refer to the ABS reference manual [1].

3.1 Functional Programming

ABS supports first-order functional programming with algebraic data types. Functional code is guaranteed to be free of side effects. Having such a functional core makes it possible to describe large parts of a software system in a side-effect-free way to simplify reasoning.

3.2 Concurrent Programming

The concurrency model of ABS is based on the concept of *Concurrent Object Groups* (COGs), which generalizes the model of single concurrent objects [19]. COGs are autonomous runtime components that are executed concurrently, share no state and communicate via asynchronous method calls. A typical ABS system consists of multiple, concurrently running COGs at runtime.

3.3 Sequential Object-Oriented Programming

ABS supports class-based, object-oriented programming with standard imperative constructs. ABS has a nominal type system with interface-based subtyping. ABS does not support class inheritance or overloading.

Interfaces.

Interfaces define types for objects. They are nominal, i.e., have a name, and define a set of method signatures, i.e., the names and types of callable methods. Syntactically, ABS interfaces look like Java interfaces.

Classes.

Classes define the implementation of objects. Classes do not define a type, in contrast to many other languages, such as Java. Classes can implement arbitrarily many interfaces, which then define the type of a new instance of that class. A class has to implement all methods of all its interfaces. In addition, a class can define *private* methods which do not appear in any interface. Such methods can then only be invoked on **this**. Instead of constructors, classes in ABS have *class parameters* and an optional *initialization block*.

Statements and Expressions.

ABS has standard statements and expressions known from languages such as Java with identical syntax. Beside side-effect-free expressions on built-in data types, expressions can be method invocations (`x.m(a)`), object creation (**new** `C(a)`), and field and variable reads and assignments (`x = this.y`). There is also a conditional statement, a while loop, and a **skip** statement, which does nothing.

3.4 Delta Modeling

ABS provides language constructs and tools for modeling variable systems following SPL engineering practices. A feature model of the SPL is specified using the *Micro Textual Variability Language* μTVL [5]. A *product selection* identifies individual products that are of particular interest to the project. *Deltas* define sets of changes to existing ABS code; they are applied incrementally to an ABS program to adapt its behaviour to conform to the specification of a particular product. Finally, the *configuration* associates features to deltas, enabling the generation of ABS source code for individual products simply by naming a product.

3.4.1 Feature Model

Different software products are distinguished from each other by which *features* they provide. Which feature combinations are supported in an SPL is then expressed by *feature models* [13]. Valid feature combinations uniquely identify the *products* of an SPL. In addition to features, ABS supports the specification of feature attributes. Whereas a feature can assume a boolean value (reflecting whether it is selected or not), attributes can also have integer or string values. By assigning a value to each feature and feature attribute such that the feature model is satisfied, one denotes a product.

ABS feature models are encoded in μTVL, a textual fea-

ture modeling language based on TVL [6]. Feature diagrams can be translated to μTVL in a straightforward manner. The following example shows a feature model of a system with features A, B and C, where two of its products are specified as P1 and P2.

```
root system {
    group [1..*] {
        A,
        B,
        C {Int size in [10..40]}
    }
}
product P1 (A, B);
product P2 (A, C{size=32});
```

The cardinality indicator [1..*] denotes that at least one feature from the group has to be present in a valid product. The feature C has an attribute size, which, when C is selected, needs to be assigned a value from the given range.

3.4.2 Deltas

ABS feature models describe the variants of a system in abstract terms using features. The system itself is implemented using delta-oriented programming [17]. Following this methodology, the code is divided into a *core* and a set of *deltas*. The core usually consists of the classes that are common to all products. Deltas describe how to change the core to obtain new products, by adding new classes and modifying (or removing) existing ones.

ABS deltas can add new methods and fields to classes, as well as remove existing methods and fields. In addition, existing methods can be *modified*, as shown in the following example.

```
delta D3 (Int size) {
    modifies class Buffer {
        modifies String toString() {
            String orig = original();
            return orig + " of size " + size;
        }
    }
}
```

The delta D3 above provides a new implementation for class Buffer's toString() method by defining a so-called method modifier. This is introduced by the **modifies** keyword and followed by the method signature and a block of code with the method's new implementation.

Original calls.

Notable within the above implementation block is the **original()** method call. Calling **original()** makes is possible to access a method's previous behaviour. This is similar to calling super to access the superclass behaviour of a method in a language with class inheritance such as Java. To accommodate the DMW, we enhanced ABS to also support *targeted* **original()** calls, making it possible to invoke a *particular* implementation of a method. We require this to avoid ambiguity, as the order of delta application is not always total, as well as to invoke multiple previous implementations in one method. A targeted original call is prefixed with the name of a delta, or with the keyword **core**. Omitting the target prefix means that the implementation from the most recently applied delta will be used.

Delta parameters.

Deltas take an optional list of parameters. These are used to pass on configuration information defined in the product selection to the implementation level. Products assign a boolean value to each feature (true if selected, false otherwise), and a value to each feature attribute (of features that are selected). In the above example, delta D3 takes one parameter Int size. Any occurrence of the integer variable size inside the delta is replaced with the concrete value of the feature attribute C.size upon delta application. The boolean values of features can be accessed in similar fashion, as shown in the example below. Delta parameters must be immutable objects, such as booleans, integers, or strings.

```
delta D4 (Bool a, Bool b, Bool c) {
    modifies class C {
        adds Bool featureA = a;
        adds Bool featureB = b;
        adds Bool featureC = c;
    }
}
```

3.4.3 Configuration

A *configuration* links feature models to deltas and guides the code generation by ordering the application of the deltas. Features and deltas are associated through *application conditions*. If a delta's application condition is true, then the delta is *applied*, i.e. the core is modified according to the changes described by the delta. Which deltas to apply is determined by feature configuration for the desired product.

A configuration example follows that links the feature model from Section 3.4.1 with the deltas from Section 3.4.2.

```
productline Example {
    features A, B, C;
    delta D1 when A;
    delta D2 when B;
    delta D3(C.size) when C;
    delta D4(A,B,C) when A and B after D1, D2;
}
```

Each **delta** line contains the name of the delta to be applied and a **when** clause with an application condition. An optional **after** clause establishes a partial order of application for the deltas. Ordering the application of deltas is used to mediate conflicts. Two deltas can be in conflict if their specified modifications do not commute. This is the case, for instance, when they both modify the same method or field in a different way. Delta D4 above may be used to resolve a conflict between D1 and D2. For this to succeed, it has to be applied **after** D1 and D2 have been applied.

4. FREDHOPPER ACCESS SERVER

The Fredhopper Access Server (FAS) is a component-based and service-oriented distributed software system. It provides search and merchandising services to e-Commerce companies such as large catalogue traders and travel agencies. Each FAS installation is deployed to a customer according to the FAS deployment architecture. Figure 2 shows an example setup. A detailed presentation of FAS's individual components and its deployment model can be found in the HATS project report [8].

A FAS deployment consists of a set of live and staging environments. A live environment processes queries from client
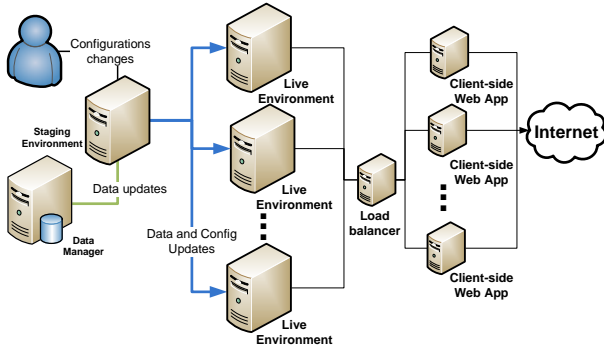
Figure 2: An example of a FAS deployment

web applications via web services. FAS aims at providing a constant query capacity to client-side web applications. A staging environment is responsible for receiving data updates in XML format, indexing the XML, and distributing the resulting indices across all live environments according to the replication protocol.

Implementations of the replication protocol are provided by the *replication system*. A replication system consists of a set of computation nodes, one of which is the synchronization server residing in a staging environment, while all other nodes are synchronization clients residing in the live environments. The synchronization server determines the schedule of replication, as well as the content of each replication item. The synchronization client is responsible for receiving data and configuration updates. A replication item is a set of files representing a single unit of replicable data.

The synchronization server communicates to clients via connection threads that serve as the interface to the server-side of the replication protocol. On the other hand, synchronization clients schedule client jobs to handle communications to the client-side of the replication protocol.

As part of the FAS product line, there are several variants of the replication system. We refer to these variants as the replication system product line and express them as features. Figure 3 shows the feature diagram of the replication system. The feature diagram has three main features: `Job Processing`, `Replication Item` and `Load`.

The feature `Job Processing` requires an alternative choice between the two subfeatures `Seq` and `Concur`, capturing the choice between sequential and concurrent client job processing, respectively. The feature `ReplicationItem` allows choosing between three replication item types represented by the features `Dir`, `File` and `Journal`. The `Dir` feature is mandatory, that is, all versions of the replication system support replicating complete file directories. The `File` feature is optional and is selected to support replicating a file set, whose files' name matches a particular pattern. the `Journal` feature is optional and is selected to support replicating database journal. In particular, the `Journal` feature requires the feature `Seq` which means that variants of the replication system that support database journal replication may only schedule client jobs sequentially.

The feature `Load` is an optional feature that configures the load of the replication system. It offers subfeatures `Client`, `CheckPoint` and `Schedule`. The feature `Client` changes the default number of synchronisation clients, and defines the constraint that if client job processing is sequential, the num-

ber of clients must be less than ten. The feature `CheckPoint` changes the default number of updates allowed per execution and defines the constraint that if the client job processing is sequential, the number of updates must be less than five. The feature `Schedule` configures the number of locations in the file system at which changes to different replication item types are monitored. It is an optional feature that offers subfeatures `DSched`, `FSched` and `JSched` to record the number of locations for directory, file set, and journal replication respectively. Note that `FSched` and `JSched` cannot be selected unless features `File` and `Journal` are selected respectively.

5. MODELING THE CASE STUDY

In this section we present how to model the members of the replication system product line using the DMW. To realize the implementations of the product line, we turn to the ABS language, which has been described in Section 3. We use a `monospaced` font to denote delta-names and ABS constructs, and a *cursive* font to denote feature-names and mathematical constructs from the DMW.

Based on the DMW, we let $(\Psi, \vDash)$ be the specification of the replication system product line, where Ψ is the structural feature model of the replication system and Φ is its representative set of feature configurations. Figure 3 shows the diagrammatic representation of Ψ. The following shows its corresponding μTVL representation:

```
root RS {
  group allof {
    JobProcessing { ... },
    ReplicationItem { ... },
    opt Load {
      group [1..3] {
        Client { Int c in [1 .. 20]; Seq -> c < 10; },
        CheckPoint { ... },
        Schedule {
          group [1..3] {
            DSched { Int s in [1 .. 5]; },
            FSched { Int f in [1 .. 5]; require: File; },
            JSched { Int l in [1 .. 5]; require: Journal;}
        }}}}}}
```

$\vDash$ is the feature satisfaction relation. It specifies when a product satisfies the specifications of a set of features. In FAS, the feature specifications are mostly in the form of descriptions (as described in Section 4) and use-cases. Noteworthy in this product line is that the features *Client* and *JSched* require some extra implementation effort to make them interact properly. Or, formally:

$$\exists p: \ p \vDash \{Client\} \land p \vDash \{JSched\} \land p \nvDash \{Client, JSched\}$$

Also, the features *Load* and *Schedule* are only subcategorizations with no semantics at all. Or, formally:

$$\forall p: \ p \vDash F \implies p \vDash F \cup \{Load, Schedule\}$$

For reasons of space and to focus on the application of the DMW, we consider only the modeling of the features *RS*, *Load*, *Client*, *Schedule*, *DSched*, *FSched* and *JSched* as the representative parts of the replication system variability. These features are shaded in the feature diagram of Figure 3. As a result, the μTVL model above is also only partially shown and we use ellipses to omit parts of the model in order to focus on those that are important for this paper. So we only consider a subset of this product line specification,
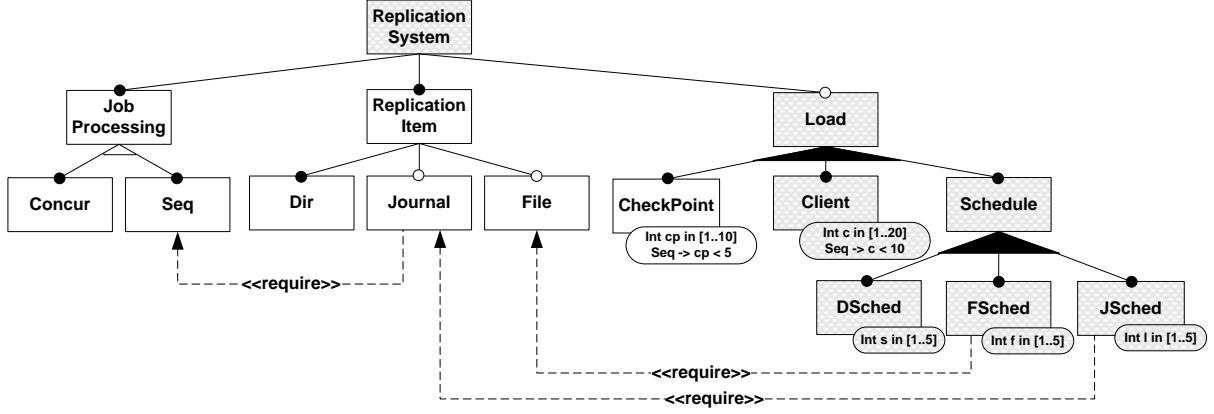
Figure 3: Feature diagram of the replication system

denoted as $(\Psi', \vDash')$, where Ψ' is defined as follows,

$$\begin{aligned}
\Psi' \;=\; &(\{RS\}, \{(RS, Load)\}, \{(Load, Client),\\
&(Load, Schedule), (Schedule, DSched),\\
&(Schedule, FSched), (Schedule, JSched)\},\\
&\varnothing, \varnothing).
\end{aligned}$$

Φ' is defined accordingly and $\vDash'$ is $\vDash$ restricted to the smaller set of features. As an example, a valid feature configuration is $\{RS, Load, Schedule, DSched\} \in \Phi'$.

To implement this product line specification we follow DMW: we consider the specification $(\Psi', \vDash')$ and start with the empty product line implementation $(c, \varnothing, \varnothing, \varnothing)$. To align with ABS implementation guidelines the core product c is defined as **class** Main {} { **new** Main();}. The main block contains one instruction to instantiate a nameless, typeless object of Main. Initially class Main provides no method and field definitions.

We implement the features in some linear extension of the transitive closure of $\multimap\!\bullet \cup \multimap\!\circ$. That is, we consider first the base features, and then their subfeatures. As such, we first work on the base feature RS. We implement that feature by the following delta RD:

```
delta RD {
  adds data SchedType = Dir | File | Journal;
  adds type CId = Int; ...
  adds class System(...) { ... }
  modifies class Main {
    adds Map<CP,Map<FId,Content>> datas = map[...];
    adds Map<SchedType,List<Schedule>> ss = map[...];
    adds Set<CId> cids = set[...];
    adds Unit run() {
      Map<CP,Map<FId,Content>> is = this.getDatas();
      List<Schedule> ss = this.getSchedules();
      Set<CId> cs = this.getCids();
      new System(is,ss,cs);
    }
    adds List<Schedule> getSchedules() {
      return lookup(ss,Dir);
    }
    adds Map<CP,Map<FId,Content>> getDatas() {
      return datas;
    }
    adds Set<CId> getCids() { return cids; }
  }
}
```

This delta adds preliminary type definitions, the class System, and modifies the definition of class Main by adding fields and methods. The delta assumes the built-in function lookup over Map data type such that lookup(ms,k) returns the value from the map ms with the key k. For brevity, we omit the full definitions of fields datas, ss and cids of Main, various type definitions and the definition of class System; their complete definition can be found in the HATS project report [8]. Specifically, the class Main provides a run() method, which is executed immediately after object creation. This method performs the following tasks: It first gathers the number of updates allowed per execution; this is recorded by the Map variable is. It then defines the schedules, or the number of locations in the file system at which changes to different replication item types are monitored; this is recorded by the List variable ss. It then defines the set of synchronisation clients to replicate data to; this is recorded by the Set variable cs. Finally, it instantiates another typeless, nameless object of class System that takes variables is, ss and cs as constructor arguments. This object essentially implements the replication protocol.

After implementing base feature RS, we have the product line $(c, \{RD\}, \varnothing, \{(RD, \Phi')\})$. Since there is only one delta and one feature, there are neither feature interactions to be implemented nor delta conflicts to be resolved. Since RS is a mandatory feature, RD is applied for all feature configurations in Φ'.

Next in line is the feature $Load$. Recall that it has no semantics. If we want to follow the DMW closely, we could create an empty delta to 'implement' this feature. But we choose not to do so. From here on, we ignore both $Load$ and $Schedule$, and treat their subfeatures as subfeatures of RS.

The next feature to implement can be any one of the four left unimplemented, as none of those is a subfeature of the others. We first consider the optional feature $Client$. It is implemented by the following delta CD:

```
delta CD(Int c) {
  modifies class Main {
    modifies Set<CId> getCids() {
      return takeSet(c, RD.original());
    }
  }
}
```

This delta modifies method `getCids()` of class `Main` to change the number of synchronisation clients for the replication system. The function `takeSet` is a built-in function over the `Set` data type such that `takeSet(c,s)` returns `c` number of elements from set `s`.

We now obtain the resulting product line, where we write $S^\star$ for the transitive closure of binary relation S:

$$RS = (c, D, \prec, \gamma) \text{ where}$$
$$D = \{\text{RD}, \text{CD}\},$$
$$\prec = \{(\text{RD}, \text{CD})\}^\star,$$
$$\gamma = \{(\text{RD}, \Phi'), (\text{CD}, \{F \in \Phi' \mid Client \in F\})\}.$$

Delta `CD` is to be applied after `RD`, as it implements a subfeature of RS. It is applied whenever the *Client* feature is selected.

The next feature we consider is *DSched*. This feature records the number of locations for directory replication. We implement this feature with the following delta `DSD`:

```
delta DSD(Int s) {
  modifies class Main {
    modifies List<Schedule> getSchedules() {
      List<Schedule> ss = RD.original();
      return take(lookup(ss,Dir),s);
    }
  }
}
```

DSched does not interact with *Client*, nor is its delta in conflict with `CD`. So, with no feature interaction or conflict resolution to implement, we obtain the following product line $RS = (c, D, \prec, \gamma)$ where

$$D = \{\text{RD}, \text{CD}, \text{DSD}\},$$
$$\prec = \{(\text{RD}, \text{CD}), (\text{RD}, \text{DSD})\}^\star,$$
$$\gamma = \{(\text{RD}, \Phi'), (\text{CD}, \{F \in \Phi' \mid Client \in F\}),$$
$$(\text{DSD}, \{F \in \Phi' \mid DSched \in F\})\}.$$

Next we consider feature *FSched* for recording the number of locations for file set replication. This feature is implemented by the following delta `FSD`:

```
delta FSD(Int f) {
  modifies class Main {
    modifies List<Schedule> getSchedules() {
      return take(lookup(ss,File),f);
    }
  }
}
```

This feature modifies `getSchedules()` to return `f` file set replication schedules.

We notice delta `FSD` causes a conflict with delta `DSD`. We resolve this conflict by providing the following delta `DFD`:

```
delta DFD {
  modifies class Main {
    modifies List<Schedule> getSchedules() {
      List<Schedule> ss = DSD.original();
      return appendRight(ss,FSD.original());
    }
  }
}
```

The delta `DFD` assumes the built-in function `appendRight` over two `List` values. Specifically, the delta `DFD` resolves the conflict between `FSD` and `DSD` by insisting that the returned

list of schedules must contain a (possibly empty) list of directory replication schedules *followed by* a (possibly empty) list of file set replication schedules. With no further feature interaction and conflict resolution, we obtain the following product line $RS = (c, D, \prec, \gamma)$ where

$$D = \{\text{RD}, \text{CD}, \text{DSD}, \text{FSD}, \text{DFD}\},$$
$$\prec = \{(\text{RD}, \text{CD}), (\text{RD}, \text{DSD}), (\text{RD}, \text{FSD}),$$
$$(\text{DSD}, \text{DFD}), (\text{FSD}, \text{DFD})\}^\star,$$
$$\gamma = \{(\text{RD}, \Phi'), (\text{CD}, \{F \in \Phi' \mid Client \in F\}),$$
$$(\text{DSD}, \{F \in \Phi' \mid DSched \in F\}),$$
$$(\text{FSD}, \{F \in \Phi' \mid FSched \in F\}),$$
$$(\text{DFD}, \{F \in \Phi' \mid \{DSched, FSched\} \subseteq F\})\}.$$

The final feature that must be considered is *JSched*, for recording the number of locations for journal replications. This feature is implemented by the following delta `JSD`:

```
delta JSD(Int l) {
  modifies class Main {
    modifies List<Schedule> getSchedules() {
      return take(lookup(ss,Journal),l);
    }
  }
}
```

This feature modifies `getSchedules()` to return `l` file set replication schedules.

When replicating journals, it is important to maintain stability at the client side. To this end, we need to make sure there exist at least two `SyncClient` instances in the replication system as a fail-safe mechanism. This means we need to implement feature interaction between features *Client* and *JSched*. We implement this interaction using delta `JCD`:

```
delta JCD {
  modifies class Main {
    modifies Set<CId> getCids() {
      Set<CId> cs = CD.original();
      if (size(cs) == 1) { cs = Insert(failSafe(),c); }
      return cs;
    }
  }
}
```

We assume the built-in functions `size()` and `failSafe()` to return the size of a set and the id of the default fail-safe synchronization client respectively. This delta has to be applied after the deltas implementing *Client* and *JSched*, and only when those features are selected.

We also note that delta `JSD` causes a conflict with both deltas `DSD` and `FSD`. The DMW would dictate that we construct a conflict resolving delta for both pairs of deltas left in an unresolved conflict (the conflict `FSD` ⨍ `DSD` was already resolved). Then, a final conflict resolving delta would be created to handle the case where all three features are selected together. This scenario is shown in Fig. 4. However, since all four of these conflicts are resolved quite consistently in each case, we can save a lot of effort if we create one conflict resolving delta `DFJD` for all four of these cases, parametrized with the selection status of the relevant features. This delta is greater in the partial order than all three original conflicting deltas, and it is to be applied when at least one of the three relevant features is selected. Note that this disjunction differs from the usual conjunctive application conditions. This is usual when working with parametrized deltas,
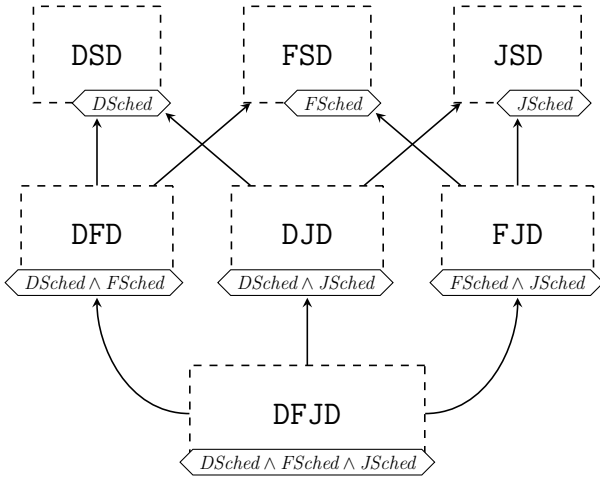
Figure 4: Example of the three-way conflict resolution between the scheduling features. The dashed boxes are deltas. The partial order $\prec$ is represented by the arrows and each delta $x \in D$ is decorated with a propositional logic formula representing $\gamma(x)$.

in which the distinction between different feature configurations is made in the code rather than on the level of delta modeling.

```
delta DFJD(Bool DSched, Bool FSched, Bool JSched) {
  modifies class Main {
    modifies List<Schedule> getSchedules() {
      List<Scdedule> ss = Nil;
      if (DSched) { ss = appendRight(ss,DSD.original()); }
      if (FSched) { ss = appendRight(ss,FSD.original()); }
      if (JSched) { ss = appendRight(ss,JSD.original()); }
      return ss;
    }
  }
}
```

This delta can now *replace* conflict resolver DFD, as it encompasses that specific case. This scenario is shown in Fig. 5.

With no further feature interaction or conflict resolution to implement in this iteration, and no further features to implement at all, we obtain the following final product line $RS = (c, D, \prec, \gamma)$ where

$$D = \{\text{RD}, \text{CD}, \text{DSD}, \text{FSD}, \text{JSD}, \text{JCD}, \text{DFJD}\},$$
$$\prec = \{(\text{RD}, \text{CD}), (\text{RD}, \text{DSD}), (\text{RD}, \text{FSD}), (\text{RD}, \text{JSD}),$$
$$(\text{DSD}, \text{DFJD}), (\text{FSD}, \text{DFJD}), (\text{JSD}, \text{DFJD}),$$
$$(\text{CD}, \text{JCD}), (\text{JSD}, \text{JCD})\}^\star,$$
$$\gamma = \{(\text{RD}, \Phi'), (\text{CD}, \{F \in \Phi' \mid Client \in F\}),$$
$$(\text{DSD}, \{F \in \Phi' \mid DSched \in F\}),$$
$$(\text{FSD}, \{F \in \Phi' \mid FSched \in F\}),$$
$$(\text{JSD}, \{F \in \Phi' \mid JSched \in F\}),$$
$$(\text{JCD}, \{F \in \Phi' \mid \{Client, JSched\} \subseteq F\}),$$
$$(\text{DFJD}, \{F \in \Phi' \mid$$
$$\{DSched, FSched, JSched\} \cap F \neq \varnothing\})\}.$$

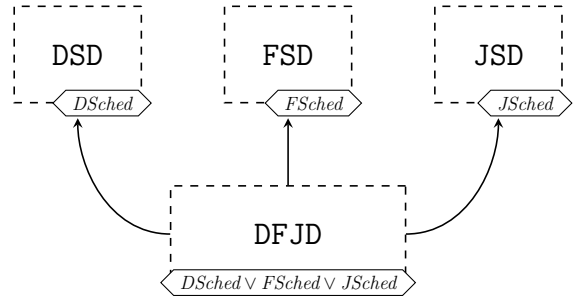The corresponding product line configuration is encoded in ABS as follows.

Figure 5: Example of the three-way conflict resolution between the scheduling features by one parametrized delta.

```
productline RS {
  features RS, Client, DSched, FSched, JSched;
  delta RD when RS;
  delta CD (Client.c) when Client;
  delta DSD(DSched.s) when DSched;
  delta FSD(FSched.f) when FSched;
  delta JSD(JSched.l) when JSched;
  delta JCD when Client and JSched after CD, JSD;
  delta DFJD(DSched, FSched, JSched)
          when DSched or FSched or JSched
          after DSD, FSD, JSD;
}
```

Using the product line RS we confirm that the application of DMW terminates for this case study and that all products generated by RS are unique and implement the required features. Table 1 lists all feature configurations $F \in \Phi'$ against its unique generated product $\text{prod}(RS, F)$ and its corresponding product selection in ABS.

6. DISCUSSION

The case study described in this paper considered the replication system, which is part of the Fredhopper Access Server (FAS) product line. The existing FAS product line is implemented in Java, and has over 150,000 lines of code. As part of future work we aim to extend this case study beyond the replication system. Table 2 shows some metrics about the existing implementation and the ABS model of the replication system. In particular, the number of deltas reduces from 10 to 7 if three-way conflict resolution between scheduling features is applied, while the number of products reduces from 12108 to 96 if feature attributes are ignored.

Metrics	Java	ABS
Nr. of lines of code	6400	5000
Nr. of classes	44	40
Nr. of interfaces	2	43
Nr. of user-defined functions	N/A	80
Nr. of user-defined data types	N/A	17
Nr. of features	N/A	15
Nr. of deltas	N/A	10 (7)
Nr. of products	N/A	12108 (96)

Table 2: Metrics about Case Study

DMW provides a step-by-step guide for developing a complete software product line from a product line specification

Feature Configuration	Generated Product	ABS Product Selection
$\{RS\}$	RD(c)	**product** P1(RS)
$\{RS,DSched\}$	DFJD(DSD(RD(c)))	**product** P2(RS,DSched{s=4})
$\{RS,FSched\}$	DFJD(FSD(RD(c)))	**product** P3(RS,FSched{f=3})
$\{RS,JSched\}$	DFJD(JSD(RD(c)))	**product** P4(RS,JSched{l=4})
$\{RS,DSched,FSched\}$	DFJD(FSD(DSD(RD(c))))	**product** P5(RS,DSched{s=4},FSched{f=3})
$\{RS,DSched,JSched\}$	DFJD(JSD(DSD(RD(c))))	**product** P6(RS,DSched{s=4},JSched{l=4})
$\{RS,FSched,JSched\}$	DFJD(JSD(FSD(RD(c))))	**product** P7(RS,FSched{f=3},JSched{l=4})
$\{RS,DSched,FSched,JSched\}$	DFJD(JSD(FSD(DSD(RD(c)))))	**product** P8(RS,DSched{s=4},FSched{f=3},JSched{l=4})
$\{RS,Client\}$	CD(RD(c))	**product** P9(RS,Client{c=5})
$\{RS,DSched,Client\}$	DFJD(DSD(CD(RD(c))))	**product** P10(RS,DSched{s=4},Client{c=5})
$\{RS,FSched,Client\}$	DFJD(FSD(CD(RD(c))))	**product** P11(RS,FSched{f=3},Client{c=5})
$\{RS,JSched,Client\}$	JCD(DFJD(JSD(CD(RD(c)))))	**product** P12(RS,JSched{l=4},Client{c=5})
$\{RS,DSched,FSched,Client\}$	DFJD(FSD(CD(DSD(RD(c)))))	**product** P13(RS,DSched{s=4},FSched{f=3},Client{c=5})
$\{RS,DSched,JSched,Client\}$	JCD(DFJD(JSD(CD(DSD(RD(c))))))	**product** P14(RS,DSched{s=4},JSched{l=4},Client{c=5})
$\{RS,FSched,JSched,Client\}$	JCD(DFJD(JSD(CD(FSD(RD(c))))))	**product** P15(RS,FSched{f=3},JSched{l=4},Client{c=5})
$\{RS,DSched,FSched,JSched,Client\}$	JCD(DFJD(JSD(CD(FSD(DSD(RD(c)))))))	**product** P16(RS,DSched{s=4},FSched{f=3},JSched{l=4},Client{c=5})

Table 1: Delta Derivations

that consists of a feature model and satisfaction relation. In this section we discuss our experiences while applying the DMW to the implementation of the FAS case study. This case study did not only raise discussion points about the pros and cons of DMW, but also guided the development of DMW while its practical applicability was put to the test.

Completeness Following DMW we are able to, in a top-down fashion, systematically implement all features in the feature model to obtain an SPL for the replication system. We are also able to systematically implement all necessary feature interaction and resolve implementation conflicts between deltas, since we are directed to consider every situation by the workflow. So we avoid accidentally forgetting to implement some functionality from a complex feature model.

Flexibility Following DMW we are able to reduce the number of deltas, in the product line, while still ensuring the final product line is complete and globally unambiguous. Specifically, we introduce the delta DFD originally to resolve the conflict between deltas DSD and FSD. However, after introducing the delta JSD for feature *JSched*, we are able to provide the single delta DFJD to resolve all conflicts between all combinations of DSD, FSD and JSD. Since DFJD is greater in the partial order than all of DSD, FSD and JSD, and can semantically resolve the conflict between DSD and FSD, it replaces the delta DFD. Reducing the number of deltas in this case also reduces redundancy and enhances the reusability of the code base.

Evolution DMW assumes the initial core product to be the empty product. We relaxed this assumption to facilitate product line evolution. In practice it is often the case that a product line will not be implemented from scratch, but will be built on legacy code, which lends itself to be incorporated as the core product.

Collaboration During the case study, we were unsure how to apply DMW in a collaborative development environment. Feedback from this case study has lead to a better formalization of concurrent development in the Delta Modeling Workflow.

Tool support In the FAS product line, we use μTVL to specify the feature model of the replication system. As a result of the case study, we feel that it would be beneficial to provide a mechanical translation from μTVL to the corresponding DMW's feature model Φ and structural feature model Ψ. While this might be outside of the scope of the development workflow, in terms of applicability, such a translation should be automated as we have found that manual translation can be error-prone. Similarly, after obtaining the final product line for the replication system, we manually extract necessary information into the ABS product line configuration. While this translation may be beyond the scope of DMW, it would help to provide software support for the translation.

Overall we have found DMW offers a useful guideline for systematically traversing the feature model and implementing its features to arrive at a software product line which is globally unambiguous (that is, for each feature configuration there is only one product) and complete (that is, each product is guaranteed to match its specifications).

7. RELATED WORK

The Abstract Behavioral Specification (ABS) language was designed within the HATS project [10]. The core ABS language [12] is a general purpose specification language for distributed object-oriented systems. The delta modeling extension of ABS [5] adds support for developing software product lines. Based on the Fredhopper case study, we have extended the language further, to support parametrized deltas and targeted original calls.

Abstract Delta Modeling [4], the formalism on which the Delta Modeling Workflow (DMW) [11] is based, is not the only way to model variability of product lines [2, 3, 9, 14, 18] but it is the first that inherently lends itself to a systematic workflow for developing product lines from scratch that support automated generation of all member products with minimal code duplication and explicit handling of interaction and conflicts.

8. CONCLUSION

Delta modeling is a relatively new paradigm for developing software product lines and, as such, has not yet been thoroughly evaluated in a "real life" development scenario. This paper is the first account of using delta modeling to implement a system of industrial scale and of practical use. We used the Delta Modeling Workflow to guide the modeling

and implementation of the Fredhopper FAS replication system in the ABS language. This case study served as a test bed for DMW, providing feedback that was used to refine the workflow. It also served as evaluation of the practical applicability of ABS and led to the addition of a more flexible mechanism for method invocation.

ABS and its implementation of delta modeling offers a lot of flexibility in designing the code base of an SPL as a core and a set of deltas, and in associating these with the feature model. Meanwhile, best practices and patterns of good delta design have yet to be established. DMW provides some much needed guidance for applying delta modeling in practice. Using DMW comes with the benefit of obtaining, through concurrent development, an SPL implementation that is both globally unambiguous and complete.

Acknowledgements

9. REFERENCES

[1] *The ABS Language Specification*, 2011. `http://tools.hats-project.eu/download/absrefmanual.pdf`.

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

[3] 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.

[4] D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract Delta Modeling. In *Proceedings of the ninth international conference on Generative programming and component engineering*, GPCE '10, pages 13–22, New York, NY, USA, Oct. 2010. ACM.

[5] D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In *Formal Methods for Components and Objects*, volume 6957 of *LNCS*. Springer, 2011.

[6] A. Classen, Q. Boucher, and P. Heymans. A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming*, 76(12):1130–1143, 2011.

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

[8] Evaluation of Core Framework, Aug. 2010. Deliverable 5.2 of project FP7-231620 (HATS), available at `http://www.hats-project.eu`.

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

[10] Highly Adaptable and Trustworthy Software using Formal Models, Mar. 2009. `http://www.hats-project.eu`.

[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] E. B. Johnsen, R. Hähnle, J. Schäfer, R. Schlatte, and M. Steffen. ABS: A core language for abstract behavioral specification. In B. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Proc. 9th International Symposium on Formal Methods for Components and Objects (FMCO 2010)*, Lecture Notes in Computer Science. Springer-Verlag, 2011. To appear.

[13] 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.

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

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

[16] I. Schaefer. Variability modelling for model-driven development of software product lines. In D. Benavides, D. S. Batory, and P. Grünbacher, editors, *International Workshop on Variability Modelling of Software-Intensive Systems (VaMoS)*, volume 37, pages 85–92. Universität Duisburg-Essen, 2010.

[17] I. Schaefer, L. Bettini, F. Damiani, and N. Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the 14th international conference on Software product lines: going beyond*, SPLC'10, pages 77–91. Springer, 2010.

[18] 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.

[19] J. Schäfer and A. Poetzsch-Heffter. JCoBox: Generalizing active objects to concurrent components. In *European Conference on Object-Oriented Programming (ECOOP'10)*, volume 6183 of *Lecture Notes in Computer Science*, pages 275–299. Springer-Verlag, June 2010.

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