

Dynamic Delta Modeling

Michiel Helvensteijn (michiel.helvensteijn@cwi.nl)

CWI, Amsterdam, the Netherlands
LIACS, Leiden University, the Netherlands

Abstract. Abstract Delta Modeling offers an algebraic description of how a (software) product line may be built so that every product can be automatically derived by structured reuse of code. 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. We present a new technique for generating efficient dynamic product lines from their static counterparts. We use Mealy machines for their dynamic reconfiguration. Furthermore, we posit that monitoring some features will be more expensive than monitoring others, and present techniques for optimizing a Mealy machine to minimize cost. We illustrate these techniques through a mobile application for Android, which dynamically changes a devices profile based on environmental factors.

1 Introduction

Delta Modeling [13, 11, 12] is designed as a technique for implementing *software product lines* [10]: a way to optimally reuse code between software products which differ only by which features they support. The code is divided into units called *deltas*, which can incrementally transform a core product in order to mechanically generate a product in the product line. Each delta has an application condition, which indicates for which combinations of features the delta should be applied. The legal combinations of features are expressed through a feature model [2]. Such legal combinations of features are commonly referred to as *feature configurations*.

Clarke et al. [3, 4] described delta modeling in an abstract algebraic manner called the *Abstract Delta Modeling* (ADM) approach. In that work, delta modeling is not restricted to software product lines, but rather product lines of any domain. It gives a formal description of deltas, how they can be combined, how they can be linked to features from the feature model, as well as how to avoid and resolve implementation conflicts.

Traditionally, a feature configuration is chosen once at build-time, its corresponding product is generated, and it does not change at runtime. Dynamic (software) product lines [7] are product lines for which the feature configuration is not fixed at runtime. It can change dynamically, in order to meet changing requirements for continuously running systems. Dynamic product lines have already been discussed in the context of delta modeling [5], but only in a concrete object oriented setting (called delta oriented programming).

We now formalize dynamic product lines in the context of ADM. We show how to transform a static product line, as described in [3, 4], into a dynamic product line. It takes the form of a Mealy Machine, a finite automaton with an input symbol and an output symbol on every transition. In our case, the input symbol corresponds to a feature that has been turned on or off and the output symbol corresponds to the delta that has to be applied to the current product in order to bring it up to date.

Based on this representation of dynamic product lines, we introduce a cost model. We assume that monitoring a specific feature for change has a certain cost, and that some features are more costly than others. We then describe how to optimize dynamic product lines by selectively removing transitions from them, effectively disregarding costly features until they become relevant.

We also introduce a novel case study to demonstrate this approach for a concrete domain. To illustrate both dynamic product lines and the versatility of ADM, we do not use a traditional software product line. Instead, we use a profile manager for modern mobile devices, such as smartphones; a much more common use case these days. By monitoring personal data such as time, location and schedule, a smartphone can automatically adjust its internal settings based on user defined rules. We show that delta modeling and, by extension, dynamic delta modeling, are a natural fit for this use-case.

The rest of the paper is structured as follows. Section 2 summarizes the relevant theory of ADM. Section 3 introduces our case study, both as motivation and illustration. Dynamic product lines are formally described in Section 4 as an extension to ADM. Section 5 describes the cost model as well as our optimization strategy. Finally, Section 6 concludes the paper and discusses related and future work.

2 Preliminaries

To make this paper self-contained, we now repeat the relevant theory from ADM. For more detailed information, we refer the reader to [3, 4]. Readers familiar with the theory can skip this section.

2.1 Products and Deltas

Firstly, we assume a set of *products*, \mathcal{P} . The set of possible modifications to products forms a delta monoid, as follows.

Definition 1 (Delta Monoid). *A delta monoid is a monoid $(\mathcal{D}, \cdot, \epsilon)$, where \mathcal{D} is a set of product modifications (referred to as deltas), and the operation $\cdot : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ corresponds to their sequential composition. $y \cdot x$ denotes the modification applying first x and then y . The neutral element ϵ of the monoid corresponds to modifying nothing.*

Applying a delta to a product results in another product. This is captured by the notion of delta application.

Definition 2 (Delta Action). Delta action is an operation $-(-) : \mathcal{D} \times \mathcal{P} \rightarrow \mathcal{P}$. If $d \in \mathcal{D}$ and $p \in \mathcal{P}$, then $d(p) \in \mathcal{P}$ is the product resulting from applying delta d to product p . It satisfies the conditions $(y \cdot x)(p) = y(x(p))$ and $\epsilon(p) = p$.

This all leads to the notion of a *deltoid*, which describes all building blocks necessary to create a product line in a concrete domain.

Definition 3 (Deltoid). A deltoid is a 5-tuple $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$, where \mathcal{P} is a product set, $(\mathcal{D}, \cdot, \epsilon)$ is a delta monoid and $-(-)$ is a delta application operator.

Finally, we describe a useful notion of expressiveness.

Definition 4 (Maximal Expressiveness). A deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$ is said to be maximally expressive iff $\forall p, p' \in \mathcal{P} : \exists x \in \mathcal{D} : x(p) = p'$.

Having this property means that any product can be transformed into any other product by applying the right delta. From this point on, we assume that every deltoid is maximally expressive. This is often true in practice.

2.2 Product Lines

We will now formally define the representation of a product line. The following concepts are built upon some deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$, which we assume as given in the definitions below.

A product line consists of several ingredients. There is a finite set of relevant *features* \mathcal{F} . From a delta modeling perspective, these features have no inherent semantic meaning and are treated as labels.

Definition 5 (Feature Model). A feature model $\Phi \subseteq \mathcal{P}(\mathcal{F})$ is a set of sets of features from \mathcal{F} . Each $F \in \Phi$ is a set of features corresponding to a valid feature configuration, i.e. a set of features that may be active together.

A product line contains a core product $c \in \mathcal{P}$ and an underlying delta model, the selective application of which to c should be able to generate any product in the product line. A delta model describes the set of deltas required to build a specific product, along with a strict partial order on those deltas, restricting the order in which they may be applied.

Definition 6 (Delta Model). A delta model is a tuple (D, \prec) , where $D \subseteq \mathcal{D}$ is a finite set of deltas and $\prec \subseteq D \times D$ is a strict partial order on D . $x \prec y$ states that x must be applied before y , though not necessarily directly before.

The partial order represents the intuition that a delta applied later has full access to earlier deltas and more authority over modifications to the product.

To connect features and deltas, we use *application conditions*.

Definition 7 (Application Function and Condition). Given a set of deltas $D \subseteq \mathcal{D}$ and feature model Φ , an application function $\gamma : D \rightarrow \mathcal{P}(\Phi)$ is used to map each delta $x \in D$ to its application condition. An application condition $\gamma(x) \subseteq \Phi$ determines for which feature configurations delta x needs to be applied.

So a product line describes all possible products, and how to generate them.

Definition 8 (Product Line). *A product line is a tuple $(\mathcal{F}, \Phi, c, D, \prec, \gamma)$ where \mathcal{F} is a feature set, $\Phi \subseteq \mathcal{P}(\mathcal{F})$ is a feature model, $c \in \mathcal{P}$ is a core product, (D, \prec) is an underlying delta model and $\gamma : D \rightarrow \mathcal{P}(\Phi)$ is an application function.*

If some feature configuration F is valid according to Φ , its corresponding product can be generated by $prod(PL, F)$, which uses all elements of the product line. For a detailed description of this process, we refer the reader to [3, 4].

3 Case Study

Traditionally a product line only exists in the development stage [10]. It is a way to efficiently reuse code between otherwise separate products. However, the delta modeling formalism described in Section 2 is a surprisingly good fit for an entirely different kind of use-case. That of a dynamic product line, which reconfigures itself at runtime.

3.1 Profile Management

The first thing that may come to mind when thinking about dynamic delta modeling is a software product line with features that can be dynamically turned on and off, based, for example, on the personal preferences of who is using the software at the time, or because of changing requirements. [7, 5]

While that would be a valid use-case, we choose to go a different direction in motivating and explaining the research in this paper: profile management on modern smartphones and other mobile devices. This may prove more interesting, as it departs from what a product line is traditionally supposed to be.

Modern smartphones and tablets, such as those based on Android [6], iOS [1] or Windows Phone [9], have access to a great variety of data with regard to the current circumstances of their user: They know the current time and their current physical location. They know which application the user is currently running, what their scheduled appointments are, and much more. This sort of information can be used to automatically adjust the devices settings based on user defined rules, such as: “when my headphones are plugged in, play music” or “when my battery is running low, turn down screen brightness”. This is known as automated profile management. We show that delta modeling is a natural way to model such rules.

The basic idea is this: a profile, consisting of a specific configuration of the devices settings, is a product in the product line. Deltas can be applied to modify those settings. Features are now specific conditions on (environmental) factors monitored by the device. Application conditions are sets of feature configurations, as before, and now represent the exact conditions under which certain settings should be modified.

A profile management (Delta Profiles [8]) application for Android [6] is currently in the final stages of development. It uses the theory in this section.

3.2 Profile Deltoid

We now formalize these notions by defining a concrete deltoid (Definition 3). Some formerly abstract constructs are refined in this section, such as what a product, a feature and a delta is. Note, however, that the remainder of this paper is still in an abstract setting, and these refinements are only used when referring to Section 3 for illustration.

We start by narrowing down, for a specific device, what it is capable of monitoring and modifying for us:

Definition 9 (Device). *A device is a 4-tuple (FC, S, V, vl) where FC contains the names of the factors that the device is capable of monitoring (e.g. ‘time’, ‘gps location’, ‘battery status’), S contains the names of the settings the device controls (e.g. ‘volume’, ‘brightness’, ‘im status’), V is the encompassing set of possible values for these factors and settings and $vl : (FC \cup S) \rightarrow \mathcal{P}(V)$ maps each factor and setting to the set of values it is allowed to take on.*

To simplify matters, we assume that FC and S are disjoint, although in practice this will not be the case. i.e., we might want to modify the devices settings based on monitoring other settings. This may introduce a host of new challenges that we plan to address in future work.

Each possible device defines a concrete deltoid upon which (dynamic) product lines can be created. From this point on, we assume that some specific device (FC, S, V, vl) is given, and define a deltoid based in that device.

Definition 10 (Profiles). *We refine the set of possible products to the set of possible profiles, mapping settings to values:*

$$\mathcal{P} \stackrel{\text{def}}{=} S \rightarrow V$$

such that for each $s \in S$ we have $\mathcal{P}(s) \in vl(s)$. (see Definition 9)

Example 1 (Profile) *As an example, consider the following profile:*

$$p_e = \left\{ \begin{array}{l} \text{‘volume’} \mapsto 10, \\ \text{‘bluetooth’} \mapsto \text{on}, \\ \text{‘brightness’} \mapsto 3, \\ \text{‘foreground app’} \mapsto \text{‘clock’}, \\ \vdots \end{array} \right\}.$$

Since S is usually quite large, we show only the relevant parts of profiles.

Definition 11 (Profile Deltas). *We refine the set of deltas to the set of profile deltas, which can modify profiles:*

$$\mathcal{D} \stackrel{\text{def}}{=} S \rightarrow V.$$

such that for each $s \in \text{dom}(\mathcal{D})$ we have $\mathcal{D}(s) \in vl(s)$ (see Definition 9). \mathcal{D} is similar to \mathcal{P} , as both map settings to values, but \mathcal{D} is a partial function. Settings that are not mapped are not modified by the delta. Neutral delta ϵ doesn’t map any settings, so it modifies nothing.

Example 2 (Profile Delta) *As an example, take the following profile delta:*

$$d_e = \left\{ \begin{array}{l} \text{'volume'} \mapsto 5, \\ \text{'foreground app'} \mapsto \text{'calendar'} \end{array} \right\}.$$

We assume that settings that are not mentioned, are not mapped.

When a delta is applied to a profile, it overwrites that profile's values:

Definition 12 (Profile Delta Action). Profile delta action $-(-) : \mathcal{D} \times \mathcal{P} \rightarrow \mathcal{P}$ is defined as follows, for all $d \in \mathcal{D}$, $p \in \mathcal{P}$ and $s \in S$:

$$(d(p))(s) \stackrel{\text{def}}{=} \begin{cases} d(s) & \text{if } s \in \text{dom}(d) \\ p(s) & \text{otherwise} \end{cases}$$

Example 3 (Profile Delta Action) *For example, delta d_e (from Example 2) applied to product p_e (from Example 1) results in the following product:*

$$d_e(p_e) = \left\{ \begin{array}{l} \text{'volume'} \mapsto 5, \\ \text{'bluetooth'} \mapsto \text{on}, \\ \text{'brightness'} \mapsto 3, \\ \text{'foreground app'} \mapsto \text{'calendar'}, \\ \vdots \end{array} \right\}.$$

Profile delta composition $\cdot : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}$ is implicitly defined by Definition 12, and can be seen as function composition. Deltas that are applied later can overwrite settings from deltas that are applied earlier.

This defines a concrete deltoid $(\mathcal{P}, \mathcal{D}, \cdot, \epsilon, -(-))$ for each device (FC, S, V, vl) .

3.3 Rule Sets as Product Lines

The idea behind the profile manager application is that the user manually creates a set of rules using the app's graphical interface, after which it is put into effect as a dynamic product line, regulating the devices profiles.

Let us examine the parts of a profile manager product line more closely. Recall that an abstract product line consists of $(\mathcal{F}, \Phi, c, D, \prec, \gamma)$. From these, D , \prec and γ are based directly on user-defined rules. The rest is implicitly derived.

Before we describe the ingredients of such a product line, we refine the notion of 'feature' that has to be used:

Definition 13 (Conditions). *We restrict the set of features for any product line for a device (FC, S, V, vl) to the set of possible conditions over factors:*

$$\mathcal{F} \subseteq FC \times \mathcal{P}(V)$$

such that for each $(fc, V') \in \mathcal{F}$ we have $V' \subseteq vl(fc)$ (see Definition 9).

Example 4 (Condition) *As an example, consider the following conditions:*

$$\begin{aligned} f_{e1} &= (\text{'time', between 9:00 and 17:00}), \\ f_{e2} &= (\text{'gps location', within 1 km of } [+52^\circ 21' 23'', +4^\circ 57' 8'']). \end{aligned}$$

Note that these names are mapped to a set of values. A condition is satisfied (a feature is 'on') if the actual value is contained within that set.

The user can input rules which include conditions and effect. These form the application function γ (Definition 7) and the delta set D respectively.

Example 5 (Rule) *For example, the rule: "if I am within 1 km of } [+52^\circ 21' 23'', +4^\circ 57' 8''] and the time is between 9:00 and 17:00, set the volume to 5 and set the calendar application to the foreground" is encoded as follows (using Examples 2 and 4):*

$$\begin{aligned} D &\ni d_e \\ \gamma(d_e) &= \{ F \in \Phi \mid \{f_{e1}, f_{e2}\} \subseteq F \} \end{aligned}$$

A partial order \prec can be defined between the deltas to encode rule priority, in order to avoid and resolve conflicts [3, 4]. Basically, a delta can overwrite values of other deltas if it is greater in the partial order.

The other elements of the product line are implicitly derived, as follows. \mathcal{F} contains all conditions that appear in application function γ :

$$\mathcal{F} = \bigcup_{d \in D} \gamma(d)$$

The feature model Φ (Definition 5) is implicitly defined by the conditions in \mathcal{F} . Basically, Φ consists of all feature configurations that do not contain any contradictory conditions but do contain all implied conditions. Finally, for our profile manager, we assume the core product c to be the devices profile as the user has manually set it before any deltas are applied. In effect, in c , every value is set to 'manual' $\in V$.

3.4 Example Rule Set

We now give an example set of rules and the corresponding product line:

- If I am within 1 km of } [+52^\circ 21' 23'', +4^\circ 57' 8''] and the current time is between 9:00 and 17:00 then
 - set 'volume' to 5.
- If I currently have a meeting scheduled then
 - set 'volume' to 0 and
 - set 'foreground app' to 'meeting minutes'
 and this rule has priority over the previous rule.

Note that we need to establish a priority between the two rules, as they might otherwise conflict with each other on the volume setting. These are the deltas:

$$\begin{aligned} D \ni d_1 &= \{ \text{'volume'} \mapsto 5 \}, \\ D \ni d_2 &= \left\{ \begin{array}{l} \text{'volume'} \mapsto 0, \\ \text{'foreground app'} \mapsto \text{'meeting minutes'} \end{array} \right\}. \end{aligned}$$

with $d_1 \prec d_2$. And these are the application conditions:

$$\begin{aligned} \gamma(d_1) &= \{ F \in \Phi \mid \{t, l\} \subseteq F \}, \\ \gamma(d_2) &= \{ F \in \Phi \mid \{m\} \subseteq F \} \end{aligned}$$

where

$$\begin{aligned} t &= (\text{'time', between 9:00 and 17:00}), \\ l &= (\text{'gps location', within 1 km of } [+52^\circ 21' 23'', +4^\circ 57' 8'']), \\ m &= (\text{'ongoing meeting', yes}). \end{aligned}$$

Based on this, $\mathcal{F} = \{t, l, m\}$ and the feature model $\Phi = \mathcal{P}(\mathcal{F})$ contains all combinations of conditions, as none of them exclude or imply each other. So there are 8 possible profiles. We will use this product line for illustration later.

4 Dynamic Product Lines

A naive way to turn a static product line PL into a *dynamic product line* (DPL), i.e. to dynamically switch from one feature configuration (F) to another (F') and keep the current product consistent with that feature configuration, is to generate the product in the traditional way, namely $prod(PL, F')$, each time the feature configuration changes. However, this can be rather costly, and hurt performance. The other extreme is to pre-generate all products, and to continually switch between them. However, the number of possible products can be exponential in the number of features, so this is infeasible for non-trivial product lines.

4.1 Dynamic Product Lines as Mealy Machines

Instead, we represent a DPL as a Mealy Machine. A Mealy Machine is a finite automaton with an input symbol and an output symbol on each transition. We assume that in a DPL the feature configuration changes dynamically by individual features being sequentially turned on and off with regard to the current feature configuration. These features are used as input symbols for our Mealy Machine. The output symbols are deltas, which, when applied to the current product, yield a new product consistent with the new feature configuration.

Definition 14 (Mealy Machine). *A Mealy Machine is represented as a 5-tuple $(S, \Sigma, \Lambda, T, G)$ where S is a set of states, Σ is an input alphabet, Λ is an output alphabet, $T : S \times \Sigma \rightarrow S$ maps a state and input symbol onto a next state*

and $G : S \times \Sigma \rightarrow \Lambda$ maps a state and input symbol onto an output symbol. T and G are partial functions but are defined for the same inputs, i.e. $\text{dom}(T) = \text{dom}(G)$. Sometimes the definition of Mealy Machine includes an initial state, but in our case the initial state is not fixed, so we do not include one.

Notation 1 (Symmetric Difference) Because input symbols in our Mealy Machines will turn a feature from a feature configuration either on or off, we often use symmetric difference between sets, which is denoted:

$$F_1 \ominus F_2 \stackrel{\text{def}}{=} (F_1 \cup F_2) \setminus (F_1 \cap F_2).$$

Notation 2 (Product Difference) Note that since we assumed our deltoid to be maximally expressive (Definition 4), any product can be transformed into any other by application of the right delta. The delta that transforms p into p' is denoted $p \mapsto p'$:

$$(p \mapsto p')(p) = p'$$

Definition 15 (Dynamic Product Line). Given a product line $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$, we define its dynamic product line as a Mealy Machine $(\Phi, \mathcal{F}, \mathcal{D}, T, G)$ where:

- Every state is a valid feature configuration in Φ .
- Every input symbol is a feature in \mathcal{F} .
- Every output symbol is a delta in \mathcal{D} .
- $T(F, f) = F \ominus \{f\}$. A feature input is an event that triggers that feature to be added to or removed from the originating state, resulting in a target state.
- $G(F, f) = \text{prod}(PL, F) \mapsto \text{prod}(PL, T(F, f))$. The output generated by a transition is the delta required to transform the product corresponding to the originating state to the product corresponding to the target state.

$T(F, f)$ and $G(F, f)$ are only defined if $F \ominus \{f\} \in \Phi$.

A dynamic product line has to be generated only once, and can then be run indefinitely. Figure 1 shows a graphical representation of the dynamic product line generated from our Section 3.4 example. Besides the conditions t , l , m and deltas d_1 , d_2 given there, the three extra deltas that appear in the diagram are:

$$\begin{aligned} d'_1 &= \{ \text{'volume'} \mapsto \text{'manual'} \}, \\ d'_2 &= \left\{ \begin{array}{l} \text{'volume'} \mapsto \text{'manual'}, \\ \text{'foreground app'} \mapsto \text{'manual'} \end{array} \right\}, \\ d''_2 &= \left\{ \begin{array}{l} \text{'volume'} \mapsto 5, \\ \text{'foreground app'} \mapsto \text{'manual'} \end{array} \right\}. \end{aligned}$$

Note from Figure 1 that we have entered a very different view of our product line than the one formed in Section 2.2. Whereas there deltas and their partial order were the main focus, we assume now that those have been processed, and we are looking at the different products of the product line, and how they relate to each other.

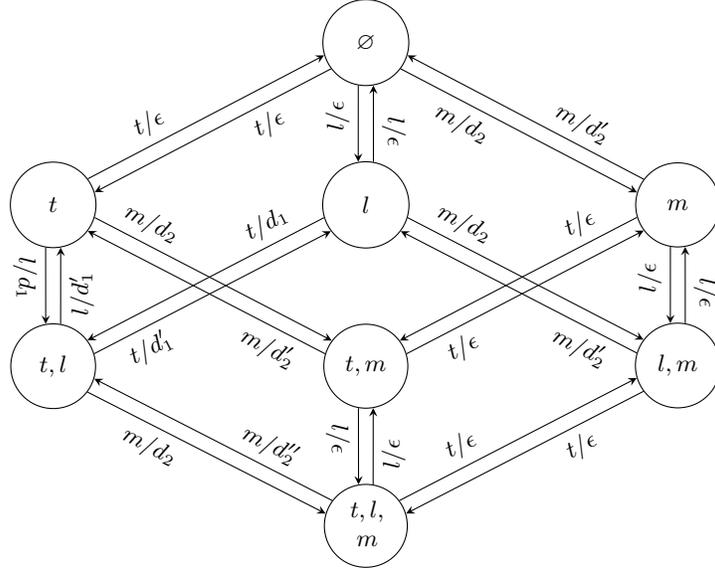


Fig. 1. The dynamic product line of the Section 3.4 example. Note that when looking at it as a cube, each dimension represents one feature.

Notation 3 (Outgoing Transitions) We write the set of outgoing transitions from state $F \in \Phi$ as:

$$out(F) \stackrel{\text{def}}{=} \{ f \mid (F, f) \in dom(T) \}$$

We now prove that if you start with a product corresponding to some state, walking any path from that state in the DPL and applying the output deltas to that product results in a product consistent with the new state.

Theorem 1 (Product Consistency). Given a dynamic product line $DPL = (\Phi, \mathcal{F}, \mathcal{D}, T, G)$ generated from PL . For any two states $F, G \in \Phi$ and any path $f_1, \dots, f_n \in \mathcal{F}^*$ such that $T(\dots T(F, f_1) \dots, f_n) = G$, deltas $d_1, \dots, d_n \in \mathcal{D}^*$ are encountered as output such that $d_n(\dots d_1(prod(PL, F)) \dots) = prod(PL, G)$.

Proof. Induction on n :

- $n = 0$. Immediate.
- $n > 0$. Take $F' = T(F, f_1)$:

$$\begin{aligned}
 d_n(\dots d_2(d_1(prod(PL, F)) & \dots)) &= \text{(Definition 15)} \\
 d_n(\dots d_2(G(F, f_1)(prod(PL, F)) & \dots)) &= \text{(Definition 15)} \\
 d_n(\dots d_2((prod(PL, F) \mapsto prod(PL, F'))(prod(PL, F)) & \dots)) &= \text{(Notation 2)} \\
 d_n(\dots d_2(prod(PL, F') & \dots)) &= \text{(induction)} \\
 prod(PL, G) & \quad \square
 \end{aligned}$$

4.2 Event Based Strategy

We first describe the most straight-forward strategy for ‘running’ a DPL.

We assume that there is some *target state* $tfc \in \Phi$, indicating the feature configuration we ‘want’ to be in. Before the Mealy Machine starts, we set the *current state* $cfc \in \Phi$ equal to the target feature configuration. The current product cp is then generated in the traditional static way, consistent with tfc :

$$\begin{aligned} cfc &\leftarrow tfc \\ cp &\leftarrow \text{prod}(PL, cfc). \end{aligned}$$

When the machine is running we assume that, when in the current state cfc , every feature f that is accepted as input from there is monitored by the system somehow. When that feature is turned on or off, giving $tfc = cfc \ominus \{f\}$, an input symbol f is generated to make the current state equal to the target state again. And in that way a stream of input symbols is generated for our Mealy Machine, which returns the appropriate deltas to keep our product up to date. So, for every input symbol f , do the following:

1. update the product $cp \leftarrow G(cfc, f)(cp)$
2. set the next state $cfc \leftarrow T(cfc, f)$

After that, the DPL has *stabilized*, until tfc changes again.

We need to make sure that any strategy we use results in a new current product consistent with the target state after the Mealy Machine has stabilized.

Definition 16 (Correct Target Product). *We say that a certain strategy for running a DPL reaches a correct target product iff whenever $cfc \neq tfc$, the Mealy Machine will stabilize after which $cp = \text{prod}(PL, tfc)$.*

Theorem 2. *The event based strategy always reaches a correct target product.*

Proof. By initialization, cp is consistent with tfc in the beginning. Whenever tfc changes, there is an $f \in \text{out}(cfc)$ (Notation 3) such that $tfc = cfc \ominus \{f\}$ (Definition 15). Feature event f is fired, giving $cp \leftarrow G(cfc, f)(cp)$ and $cfc \leftarrow T(cfc, f) = tfc$. So by Theorem 1, $cp = \text{prod}(PL, cfc) = \text{prod}(PL, tfc)$. \square

5 Optimization

In this section, we discuss the cost of dynamic product lines, and the minimization of that cost.

5.1 Cost Model

We assume that occupying a state in a DPL has a cost: the cost of monitoring the features of the outgoing transitions for change. We posit that monitoring some features will be more expensive than monitoring others. For example, in Section 3.4 it will cost more power to continually monitor the current ‘gps location’ (f_l) than it will to monitor the current ‘time’ (f_t).

First we introduce some cost domain C over time (such as *power* in *watt*).

Definition 17 (Cost). *Given dynamic product line $(\Phi, \mathcal{F}, \mathcal{D}, T, G)$ we introduce a function $cost : \Phi \times \mathcal{F} \rightarrow C$. The function $cost(F, f)$ expresses the cost of monitoring feature f from state F . A feature is only monitored from a state if it has an outgoing transition there, so if $(F, f) \notin dom(T)$ then $cost(F, f) = 0$.*

In the Section 3 profile manager, we assume that the cost of monitoring a condition depends only on the factor being checked, so we can use a short notation for that: $cost(fc) = cost(F, (fc, V'))$ for all $F \in \Phi$, $fc \in FC$ and $V' \subseteq V$.

Basically, we minimize the cost of a DPL by removing costly transitions, but only where this would not ‘break’ the machine. This means we only need to monitor features when they become relevant. For example, in the Section 3 profile manager, we want to modify the profile when we are in a certain ‘gps location’ (f_l) at a certain ‘time’ (f_t). Either condition satisfied on its own does not modify the profile. So it makes sense to only start monitoring ‘gps location’ (the more costly factor), when it is already the right ‘time’.

We need to find conditions under which transitions may be removed, as well as an accompanying strategy for walking through a reduced DPL. We give possible solutions in Sections 5.2 and 5.3.

5.2 Reduced Dynamic Product Lines

We first establish some useful notions. We set up an equivalence relation between different feature configurations that correspond to the same generated product.

Definition 18 (Feature Configuration Equivalence). *Given a product line $PL = (\mathcal{F}, \Phi, c, D, \prec, \gamma)$, two feature configurations $F, G \in \Phi$ are equivalent, denoted $F \equiv G$, iff $prod(PL, F) = prod(PL, G)$.*

Realize that while running a DPL, by Definition 16, we don’t care whether we end up in the exact target state. It is sufficient that we always reach an equivalent state. That is why this will be a useful notion once we start removing transitions.

When walking a DPL from which transitions are removed, it is possible that our path will be blocked, preventing current state cfc to become equal to target state tfc . This means that even though tfc will only change by one feature at a time, a gap greater than one feature may appear between the two. In Section 5.3, we will explain how to walk through such a machine by nondeterministically firing a transition whenever some are available, to bring cfc closer to tfc .

Definition 19 (Available Transitions). *For current state $cfc \in \Phi$ and target state $tfc \in \Phi$, the set of available transitions is defined as:*

$$at(cfc, tfc) \stackrel{\text{def}}{=} (cfc \ominus tfc) \cap out(cfc)$$

We give a new transition function which takes a target state, rather than a single feature, and returns the set of possible resulting states. It is defined recursively.

Definition 20 (Target Based Transition Function). Given dynamic product line $(\Phi, \mathcal{F}, \mathcal{D}, T, G)$, the target based transition function $\bar{T} : \Phi \times \Phi \rightarrow \mathcal{P}(\Phi)$ is defined as:

$$\bar{T}(cfc, tfc) \stackrel{\text{def}}{=} \bigcup_{f \in at(cfc, tfc)} \bar{T}(T(cfc, f), tfc)$$

It always terminates, as the set $at(cfc, tfc)$ becomes smaller when a step is taken.

We now define what a reduced dynamic product line is:

Definition 21 (Reduced Dynamic Product Line). The Mealy Machine $DPL' = (\Phi, \mathcal{F}, \mathcal{D}, T', G')$ is a reduced dynamic product line iff there is a dynamic product line $DPL = (\Phi, \mathcal{F}, \mathcal{D}, T, G)$ such that:

- $\forall (F, f) \in dom(T') : T'(F, f) = T(F, f)$
- $\forall (F, f) \in dom(G') : G'(F, f) = G(F, f)$
- $\forall cfc, tfc \in \Phi : \forall F \in \bar{T}'(cfc, tfc) : F \equiv tfc$

The first two conditions indicate that the set of transitions in DPL' should be a subset of the transitions in DPL . The third condition makes sure that despite the ‘missing’ transitions, we still always reach a correct target product. Note that by that third condition, we may only remove transitions with ϵ output.

Figure 2 shows a reduced version of the dynamic product line of Figure 1 with equivalence classes marked (basically, wherever an ϵ transition was).

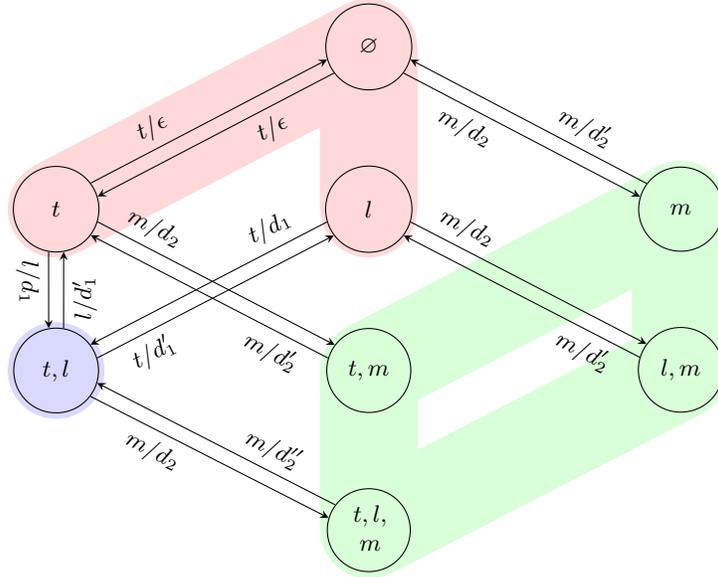


Fig. 2. The reduced version of Figure 1 with equivalence classes marked.

Let us assume that for the device we are considering:

$$\text{cost}(\text{'time'}) < \text{cost}(\text{'ongoing meeting'}) < \text{cost}(\text{'gps location'}).$$

So we would like to get rid of l/ϵ transitions first, then m/ϵ transitions, then t/ϵ transitions, so long as the third condition from Definition 21 continues to hold. As you can see in Figure 2, we were able to remove 10 transitions, significantly reducing the overall cost of the DPL. Intuitively, the transitions between \emptyset and l could be removed, because the ‘gps location’ does not become relevant until it is the right ‘time’. The other 8 transitions could be removed because d_2 completely overwrites the effect of d_1 , so it does not matter what happens with t and l while we are in a scheduled meeting. Profile manager rules often look like this in practice, so the DPL can often be significantly optimized by this process.

5.3 Target Based Strategy

The strategy of Section 4.2 is most straight-forward, but it will no longer reach a correct target product (Definition 16) after we have removed transitions, because we might disregard a feature event when we are not monitoring it, and then neglect to apply it when it does become relevant. The solution is to modify our strategy to take this into account, to safely walk a reduced DPL (Definition 21).

The target state tfc will gradually change by single features being turned on and off, as before. But now the current state cfc will always try to ‘mimic’ the target feature configuration by generating feature events based on the current difference between the two (Definition 19) and walking the Mealy Machine accordingly. For example, if the current state is $\{f, g\}$ and the target state is $\{g, h\}$, the f and h events may be triggered in a nondeterministic order until $cfc = tfc$ or until we are blocked from going any further by missing transitions.

The target based strategy works as follows. Given the reduced dynamic product line $DPL = (\Phi, \mathcal{F}, \mathcal{D}, T, G)$ created originally from product line PL , we first set the initial state cfc to equal tfc , and its corresponding product cp is generated in the familiar static way as before:

$$\begin{aligned} cfc &\leftarrow tfc \\ cp &\leftarrow \text{prod}(PL, cfc) \end{aligned}$$

When we start running the DPL, tfc may change and the set of available transitions $at(cfc, tfc)$ will become non-empty. We then initialize an empty delta d :

$$d \leftarrow \epsilon$$

and repeat the following until $at(cfc, tfc)$ is empty again:

1. nondeterministically choose a feature $f \in at(cfc, tfc)$,
2. compose the resulting delta $d \leftarrow G(F, f) \cdot d$ and
3. set the next state $cfc \leftarrow T(cfc, f)$.

This is consistent with the target based transition function from Definition 20.

When $at(cfc, tfc)$ is once again empty, and the dynamic delta model has stabilized, we apply the accumulated deltas to the current product:

$$cp \leftarrow d(cp)$$

We now prove that the target based strategy always results in a correct target product (Definition 16) for reduced dynamic product lines.

Theorem 3. *The target based strategy applied on a reduced dynamic product line always reaches a correct target product.*

The proof of this theorem relies heavily on the third condition of Definition 21, which is admittedly a strong condition. Finding an efficient algorithm for deciding whether or not a transition may be removed is future work.

Proof. By initialization, cp is consistent with tfc in the beginning. Then, whenever tfc changes, cfc will become one of the states in $\overline{T}(cfc, tfc)$ after stabilization. By Theorem 1, we have $cp = prod(PL, cfc)$. By Definition 21, we will have $cfc \equiv tfc$ and thus $cp = prod(PL, tfc)$.

With an example walk in Figure 2, we show how we can always reach a state equivalent to the one targeted. We start in state $cfc = tfc = \emptyset$. Say we reach specified gps coordinates before 9:00. Transition \bar{l} would be fired, if it were available. As it is, $cfc = \emptyset$ and $tfc = \{l\}$. If we then start a scheduled meeting, m is fired, and $cfc = \{m\}$ and $tfc = \{l, m\}$. If the meeting ends (bringing us back to the previous situation), and it becomes 9:00, $cfc = tfc = \{t, l\}$, because t and l both fire. In all cases, we have $cfc \equiv tfc$, so we always reach the correct product.

6 Conclusion

Dynamic Delta Modeling is an extension of Abstract Delta Modeling [3, 4] which includes a formal framework for modeling dynamic product lines. Using Mealy Machines, we accurately describe the behavior of product lines with dynamic feature configurations, while remaining on an abstract level. We have defined a cost-model, and shown an optimization opportunity for certain kinds of dynamic product lines. We have described the practical case-study of profile management on modern mobile devices directly in our formal framework, illustrating the versatility of ADM and the applicability of its dynamic extension.

6.1 Related Work

Hallsteinsen et al. [7] introduce several properties they believe constitute a dynamic software product line. Dynamic Delta Modeling allows several of these, such as ‘dynamic variability’, ‘changes binding several times over lifetime’ and ‘context awareness’, but does not yet model others, such as ‘variation point change during runtime’ and ‘deals with unexpected changes during runtime’. In

our approach, even though the current feature configuration can change during runtime, the set of available feature configurations is still fixed at build time.

The work of Damiani and Schaefer [5] complements our work. A Mealy Machine generated by the technique in this paper, in a concrete object oriented setting, may be enriched by their *reconfiguration translations*, basically embedding a *reconfiguration automaton* into our DPL, and in that way reconfiguring existing objects in the heap to be consistent with the change in code. To be fully compatible with their technique, stabilization of the DPL should wait until the `reconfigure` statement is encountered by the program.

6.2 Future Work

We have made a beginning, but there is plenty of opportunity for future work. Optimization of the dynamic product line is still a complex endeavor, so there is need for an efficient algorithm. Also, we now assume that the target feature configuration can only change by single features being turned on and off and we would like to loosen that restriction, a goal that introduces several complications. On a more concrete level, it would be interesting to enrich the profile manager to allow monitoring – as well as modifying – settings. As delta application would be able to trigger a new feature configuration change, some sort of termination analysis would be required.

References

1. Apple. iOS. <http://www.apple.com/ios>.
2. D.S. Batory. Feature Models, Grammars, and Propositional Formulas. In *Proc. Int'l Software Product Line Conference (SPLC)*, 2005.
3. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In *Proc. of GPCE*, pages 13–22. ACM, 2010.
4. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. *Accepted to MSCS special issue*, 2012.
5. F. Damiani and I. Schaefer. Dynamic delta-oriented programming. In *Proceedings of the 15th International Software Product Line Conference*, page 34. ACM, 2011.
6. Google. Android. <http://www.android.com>.
7. S. Hallsteinsen, M. Hinchey, S. Park, and K. Schmid. Dynamic software product lines. *Computer*, 41(4):93–95, 2008.
8. M. Helvensteijn. Delta Profiles. <http://code.google.com/p/delta-profiles>.
9. Microsoft. Windows Phone. <http://www.microsoft.com/windowsphone>.
10. K. Pohl, G. Böckle, and F. Van Der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, Heidelberg, 2005.
11. 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.
12. 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.
13. 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.