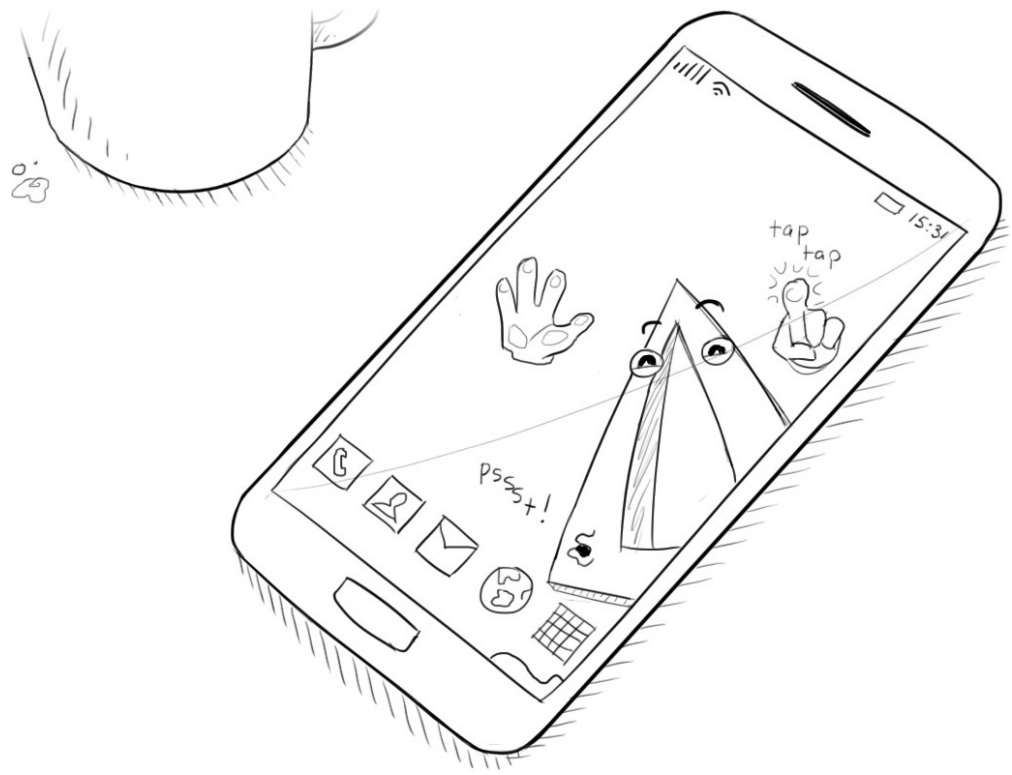
8

Dynamic Product Lines

On the Runtime and Context-aware Application of Deltas



8.1 Introduction

Traditionally, a feature configuration (Section 4.2.2, page 100) is chosen once at build-time. Its corresponding product is then generated and deployed, after which the chosen feature configuration can no longer change. That is sometimes limiting, as it could be advantageous for products to be able to adapt to dynamic conditions at runtime [24]. *Dynamic software product lines* [82] are product lines for which the feature configuration is *not* fixed. It can be changed dynamically in order to meet changing requirements for continuously running systems, upon which the running product can adapt accordingly.

Damiani et al. have discussed delta-based dynamic software product lines before [63, 64]. As a way of offering additional insight, the motivation of this chapter is based on their work. They explore several of the problems encountered in an object-oriented setting. In particular, they introduce a **`reconfigure`** statement to the programming language, which, when reached at runtime, offers the system the opportunity to adapt the running product to the newest feature configuration. A developer places this statement wherever it is deemed safe for the system to do so without creating inconsistencies — a sensible precaution. However, the semantics of **`reconfigure`** is never formalized. From the perspective of structural operational semantics [153], the corresponding inference rule might take the following shape:

$$\frac{\langle \mathit{premises} \rangle}{\langle\, p, H, \texttt{reconfigure}; st, \sigma \,\rangle \;\longrightarrow\; \langle\, p', H', st, \sigma \,\rangle}$$

The next statement st and current state σ are classical in structural operational semantics. The current product (code) p and current heap H are needed to capture the meaning of the **`reconfigure`** statement, because both need to be modified during reconfiguration.

The work of Damiani et al. focuses specifically on the modification of the heap, but does not discuss modification of the product. It is true; if functional correctness is the only concern, the product can simply be generated from scratch each time, using existing techniques (Chapter 4). But if efficiency is a factor, this approach won't suffice. Nor is it feasible to store every possible configuration of the code; this number can be exponential in the number of features, and the approach can't ever scale to the more complex requirement of adaptation to unanticipated change [146]. This chapter explores some strategies that are potentially better:

> ***Goal:*** *Formulate efficient strategies for reconfiguration of the running product in an ADM-based dynamic product line.*

The strategies explored in this chapter are based on the assumption that the difference between two subsequent running products will be small, relative to their individual size. So rather than build them from scratch every time, lightweight deltas can be derived at build-time, which can then be used to perform the proper transformation at runtime.

To reason about the correctness and efficiency of this approach to dynamic reconfiguration, we introduce a new operational semantics. We develop models to represent dynamic product lines in an abstract context and explore different

strategies for 'running' them. These models are defined in terms of *Mealy machines* [131]: finite state machines with an input symbol and an output symbol on each transition. In our case, the input symbol corresponds to a feature (or features) that has been turned on or off by external events and the output symbol corresponds to the delta that has to be applied to bring the current product up to date.

Besides being inherently more efficient than the naive approach, it affords us the opportunity to apply a particular kind of optimization. We assume that monitoring specific features for change has a certain cost —different for each feature— such as powering a sensor or polling a server. We introduce a *cost model* to express these costs, and *optimize* dynamic product lines by disregarding costly features until they become relevant. This is modeled by selectively removing transitions from the Mealy machine.

As this chapter does not offer any contributions regarding the heap or control flow issues of dynamic *software* product lines, it uses an alternative domain so as not to distract from the main contribution. A novel case-study is presented: the development of a mobile application for automated profile management, which is used as a running example throughout the chapter. By monitoring personal data such as time, location and schedule, a smartphone can automatically adjust its internal settings based on user defined rules, essentially operating as a dynamic product line. This allows us to explore strategies for reconfiguring running products without having to consider software-specific issues.

> ***Goal:*** *Develop a profile management app for Android based on the dynamic product line strategies explored in this chapter.*

The rest of the chapter is structured as follows: Section 8.2 introduces the case study that will be used to illustrate the theory of the chapter. We then develop the operational semantics and Mealy machine model in Section 8.3 — the main section of the chapter— and introduce the feature-based cost-model and optimization techniques in Section 8.4. Section 8.5 ties up loose ends by showing how the new operational semantics may be integrated with the classical programming language semantics and, finally, Sections 8.5 and 8.6 offer concluding remarks and discuss related work.

8.2 Automated Profile Management

The running example of this chapter is a mechanism for *automated profile management* on modern mobile devices. Smartphones and tablets, such as those based on Android [11], iOS [20] or Windows Phone [134], have access to a great variety of data concerning the current circumstances of their user: the current time and physical location, their scheduled appointments, which application is currently running, and so on. Privacy issues aside, that sort of information can be used to automatically adjust the devices settings based on user defined rules, such as: "when I'm at the movies, mute all sound" or "when my battery is running low, turn down screen brightness".

This example addresses a real practical need. Smartphones are ubiquitous these days, and a number of applications already provide automated profile management. However, they suffer from various limitations, and are often so complex that one has to be a programmer to use them. I felt I could improve upon this.

The theory in this chapter has lead to the development of a new profile management application for Android [85]. Besides offering a great deal of power with an intuitive user experience, the application also serves to illustrate the versatility of ADM and the theory of its dynamic counterpart.

8.2.1 A Mobile Device

We start by introducing a simplified model of a mobile device. We are interested in two distinct aspects: *quantities* and *settings*. Quantities are what we want a device to monitor, such as 'location', 'schedule', 'weather forecast' and 'battery level'. Settings are all aspects of the device that the user has control over, such as 'volume', 'brightness', 'chat status', 'alarm', and so on.

▷ **8.1. Definition (Device):** First, assume a universal set of identifiers $\mathcal{ID}$ and a universal set of values $\mathcal{V}$. A *device* is a triple $(ID_{\mathrm{q}}, ID_{\mathrm{s}}, \mathrm{type})$ where:

- $ID_{\mathrm{q}} \subseteq \mathcal{ID}$ is a finite set of names for all *quantities* the device can monitor.
- $ID_{\mathrm{s}} \subseteq \mathcal{ID}$ is a finite set of names for all *settings* the device can modify.

The names of quantities and the names of settings are disjoint: $ID_{\mathrm{q}} \cap ID_{\mathrm{s}} = \varnothing$.

- The function $\mathrm{type}: ID_{\mathrm{q}} \cup ID_{\mathrm{s}} \to \mathrm{Pow}(\mathcal{V})$ maps a quantity or setting to its set of possible values. For example, we'd have $\mathrm{type}(\text{battery level}) = \{\, 0\%, ..., 100\% \,\}$ and $\mathrm{type}(\text{chat status}) = \{\, \text{available}, \text{busy}, \text{offline} \,\}$. ⌟

From this point on, for the rest of the chapter, assume that some device $DEV = (ID_{\mathrm{q}}, ID_{\mathrm{s}}, \mathrm{type})$ is given.

We call a complete mapping of quantity values a device's *environment*, and a collection of its current settings its *profile*. The environment is 'read-only'. Through user-defined rules, specific environmental conditions can trigger a modification to the profile. We define the notion of profile explicitly:

▷ **8.2. Definition (Profile):** Define the set of *profiles* $\mathcal{P}_{DEV}$ as a map of all of a device's settings to values of the proper type:

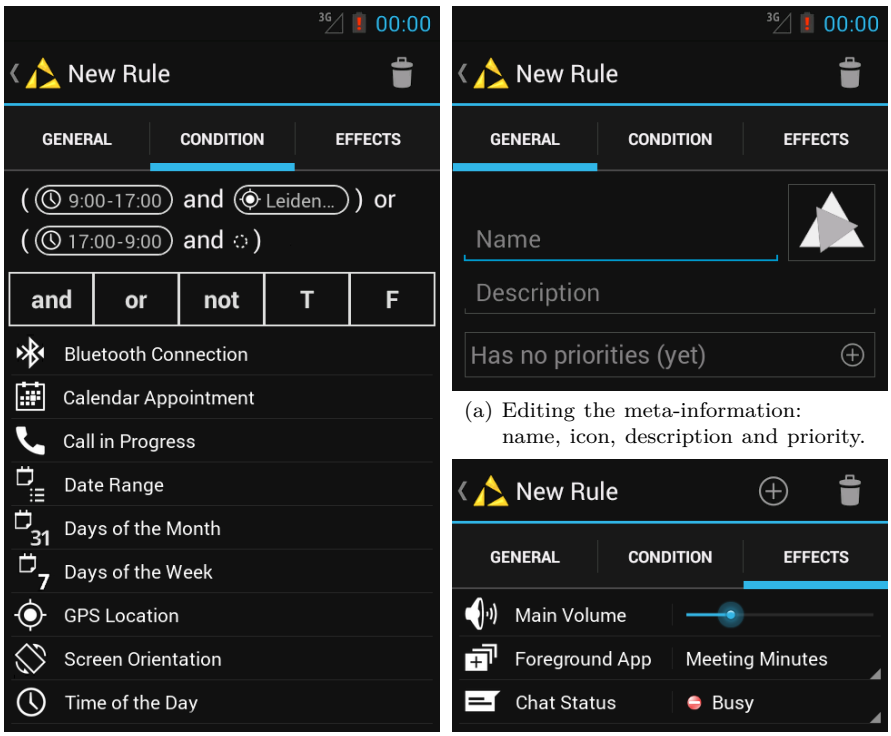
$$\mathcal{P}_{DEV} \quad \stackrel{\text{def}}{=} \quad ID_{\mathrm{s}} \;\to\; \mathcal{V}$$

such that $p(id) \in \mathrm{type}(id)$ for all $p \in \mathcal{P}_{DEV}$ and all $id \in ID_{\mathrm{s}}$. ⌟

▷ **8.3. Example:** The following is a profile $p_{\mathrm{x}} \in \mathcal{P}_{DEV}$:

$$p_{\mathrm{x}} \;=\; \left\{ \begin{array}{rcl} \text{volume} & \mapsto & 10, \\ \text{bluetooth} & \mapsto & \text{on}, \\ \text{brightness} & \mapsto & 3, \\ \text{foreground app} & \mapsto & \text{clock}, \\ & \vdots & \end{array} \right\}$$

Since the number of settings is usually quite large, we show only relevant ones. ⌟



(a) Editing the meta-information: name, icon, description and priority.

(b) Editing the condition: drag and drop constraints from the list.

(c) Editing the effects; add more using the ⊕ button in the action bar.

Figure 8.1: Screenshots of the Android interface. These controls are used for editing profile management rules.

Profiles play the rôle of products (Notation 2.9, page 36) in a device specific deltoid which will be defined shortly.

8.2.2 Rules

The idea behind the profile manager application is that the user manually inputs a set of *rules* using the graphical interface (Figure 8.1). A rule consists of an environmental *condition* and an *effect specification*, which contains new settings. A condition is entered as a formula containing constraints on specific quantities (Figure 8.1b). A constraint is formally defined as follows:

▷ **8.4. Definition (Constraint):** A *constraint* is a dependent pair $\langle id, VAL \rangle$ where $id \in ID_q$ is the name of a quantity and $VAL \subseteq \text{type}(id)$ is the set of values to which id is constrained. The set of all possible constraints is denoted C_{DEV}. ⌟

Constraints play the rôle of features (Notation 4.2), since environmental constraints ultimately decide what the current profile should be.

An *effect specification* is formally similar to a profile (Definition 8.2), as both map settings to values. But profiles are total functions, whereas effect specifications are partial; they represent only the *changes* to a profile:

▷ **8.5. Definition (Effect Specification):** Define the set of *effect specifications* $\mathcal{D}_{DEV}$ as a map of *some* settings to new values:

$$\mathcal{D}_{DEV} \quad \stackrel{\text{def}}{=} \quad ID_{\text{s}} \rightharpoonup \mathcal{V}$$

such that $d(id) \in \text{type}(id)$ for all $d \in \mathcal{D}_{DEV}$ and all $id \in ID_{\text{s}}$. Settings that are not to be modified are not mapped. ⌟

▷ **8.6. Example:** The following is an effect specification d_{x}:

$$d_{\text{x}} = \left\{ \begin{array}{rcl} \text{volume} & \mapsto & 5, \\ \text{foreground app} & \mapsto & \text{calendar} \end{array} \right\}$$

Settings that are not mentioned are not mapped. ⌟

Effect specifications, of course, play the rôle of deltas (Notation 2.10, page 36). They form a monoid:

▷ **8.7. Definition (Profile Delta Monoid):** The *profile delta monoid* $(\mathcal{D}_{DEV}, \cdot, \varepsilon)$ has a composition operator $\cdot \colon \mathcal{D}_{DEV} \times \mathcal{D}_{DEV} \to \mathcal{D}_{DEV}$ defined as follows, for all deltas $x, y \in \mathcal{D}_{DEV}$ and identifiers $id \in \mathcal{ID}$:

$$(y \cdot x)(id) \quad \stackrel{\text{def}}{=} \quad \begin{cases} y(id) & \text{if } \ id \in \text{pre}(y) \\ x(id) & \text{if } \ id \in \text{pre}(x) \\ \bot & \text{otherwise} \end{cases}$$

The neutral profile delta $\varepsilon = \varnothing$ is the "everything undefined" function, mapping no identifiers at all. ⌟

The profile deltoid is functional (Definition 2.66, page 59). As such, we'll simplify the type of the evaluation operator:

▷ **8.8. Definition (Profile Deltoid):** The *profile deltoid* for a device *DEV* is a deltoid $Dt_{DEV} \stackrel{\text{def}}{=} (\mathcal{P}_{DEV}, \mathcal{D}_{DEV}, \cdot, \varepsilon, [\![\text{-}]\!])$, with product set $\mathcal{P}_{DEV}$ from Definition 8.2, delta monoid $(\mathcal{D}_{DEV}, \cdot, \varepsilon)$ from Definitions 8.5 and 8.7 and semantic evaluation operator $[\![\text{-}]\!] \colon \mathcal{D}_{DEV} \to (\mathcal{P}_{DEV} \to \mathcal{P}_{DEV})$ defined as follows, for all profile deltas $d \in \mathcal{D}_{DEV}$, profiles $p \in \mathcal{P}_{DEV}$ and identifiers $id \in ID_{\text{s}}$:

$$[\![d]\!](p)(id) \quad \stackrel{\text{def}}{=} \quad \begin{cases} d(id) & \text{if } \ id \in \text{pre}(d) \\ p(id) & \text{otherwise} \end{cases}$$

⌟

▷ **8.9. Example:** For example, applying delta d_{x} from Example 8.6 to profile p_{x} from Example 8.3 results in the following profile:

$$[\![d_{\text{x}}]\!](p_{\text{x}}) = \left\{ \begin{array}{rcl} \text{volume} & \mapsto & 5, \\ \text{bluetooth} & \mapsto & \text{on}, \\ \text{brightness} & \mapsto & 3, \\ \text{foreground app} & \mapsto & \text{calendar}, \\ & \vdots & \end{array} \right\}$$

⌟

In conclusion, the domain of profile management gives rise to many deltoids — one for every device *DEV*.

Based on such a deltoid, a set of user-defined rules is defined as follows:

▷ **8.10. Definition (Rule-set):** A *rule-set* is a triple $(D, \prec, \gamma)$ where $D \subseteq \mathcal{D}_{DEV}$ is a set of profile deltas representing the effects of the rules, $\prec \subseteq D \times D$ is a strict partial order representing rule-priority and the function $\gamma\colon D \to \mathrm{Pow}(\mathrm{Pow}(C_{DEV}))$ maps effect specifications to the condition under which they should be applied. ⌟

Rule-sets, then, play the rôle of annotated delta models (Definition 4.7, page 103). Intuitively, a rule is an instruction to the profile manager: "Whenever $\gamma(d)$ holds, ensure that the device is set to the values in d." A condition $\gamma(d) \subseteq \mathrm{Pow}(C_{DEV})$ is a set of sets of constraints, but should be thought of as a formula in disjunctive normal form, i.e., a disjunction of conjunctions of constraints (Figure 8.1b).

8.2.3 Defining Rules

I now present a typical scenario of a user entering some new rules, resulting in an example rule-set $(D_{\mathrm{x}}, \prec_{\mathrm{x}}, \gamma_{\mathrm{x}})$. We use these rules as a running example throughout the remainder of the chapter. We first specify each rule in an informal manner and follow up with their formalization.

The user enters the first rule:

▷ **8.11. Rule (At Work):** Whenever I am within 1 km of the Leiden University computer science building between 9:00 and 17: 00, I want volume set to 5:

$$\begin{aligned} \gamma_{\mathrm{x}}(x_{\mathrm{x}}) &\stackrel{\text{def}}{=} \big\langle \text{time, between 9:00 and 17:00} \big\rangle \wedge \\ &\quad\ \big\langle \text{location, } < 1 \text{ km of } +52°\ 10'\ 10'',\ +4°\ 27'\ 24'' \big\rangle \big\} \text{ condition} \\ x_{\mathrm{x}} &\stackrel{\text{def}}{=} \{ \text{volume} \mapsto 5 \} \big\} \text{ effect} \end{aligned}$$ ⌟

The user then proceeds to enter the second rule:

▷ **8.12. Rule (In a Meeting):** During a scheduled meeting, I want the volume set to 0 and the 'meeting minutes' app brought to the foreground:

$$\begin{aligned} \gamma_{\mathrm{x}}(y_{\mathrm{x}}) &\stackrel{\text{def}}{=} \big\langle \text{meeting, } \{\text{true}\} \big\rangle \\ y_{\mathrm{x}} &\stackrel{\text{def}}{=} \{ \text{volume} \mapsto 0, \\ &\qquad \text{foreground app} \mapsto \text{'meeting minutes'} \} \end{aligned}$$ ⌟

Both rules are entered through the interface shown in Figure 8.1. A name, description and icon can be associated with each rule, but those are not relevant to the formalism.

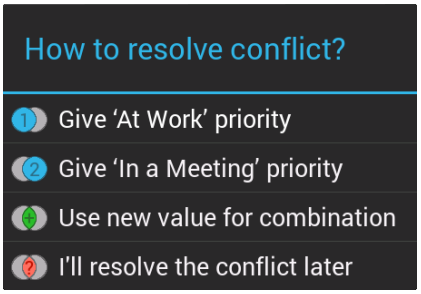


Figure 8.2: I found a conflict.

Upon entering the second rule, the user receives a warning from the application (Figure 8.2). The two rules have overlapping conditions —which means both can be true at the same time— but they disagree about the proper volume setting. So if the user ever attends a scheduled meeting at work during the designated working hours, the profile manager will not know whether the volume should be set to 0 or to 5. To break the tie, the user is given a choice:

1. "always grant priority to the first rule (i.e., set the volume to 5)",
2. "always grant priority to the second rule (i.e., set the volume to 0)",
3. "use a third value, specifically for the combination $\gamma(x_{\mathbf{x}}) \wedge \gamma(y_{\mathbf{x}})$", or
4. "deactivate the rule-set for now; I'll correct the problem later".

In this case, the user chooses option 2 to give the second rule priority:

$$x_{\mathbf{x}} \prec_{\mathbf{x}} y_{\mathbf{x}}$$

In a different situation, an alternative resolution might have been more appropriate. Perhaps a combination of two or more conditions requires specific consideration, and rather than give priority to either rule, a third alternative is required. Option 3 would automatically create a rule $z_{\mathbf{x}}$ such that $x_{\mathbf{x}}, y_{\mathbf{x}} \prec_{\mathbf{x}} z_{\mathbf{x}}$ with a preset default value for the volume setting to override the conflict. Option 4 gives the user the opportunity to manually correct the problem at leisure; perhaps by editing one or both rules. These options correspond roughly to Actions 3.9, 3.11 and 3.8 (pages 77 and 78).

8.2.4 Rule-sets as Product Line Implementations

Next, we create a dynamic product line from a given rule-set. 'Profile features', as noted earlier, are environmental constraints (Definition 8.4). We name the constraints of the running example above as follows:

▷ **8.13. Example (Profile Features):** The set of features $\mathcal{F}_{\mathbf{x}} \subseteq C_{DEV}$ relevant to the example rule-set of Section 8.2.3 is $\{ t, l, m \}$, where:

$$\begin{array}{rcl} t & \stackrel{\text{def}}{=} & \langle \text{time}, \quad \text{between 9:00 and 17:00} \quad \rangle \\ l & \stackrel{\text{def}}{=} & \langle \text{location}, \ < 1 \text{ km of } +52° \text{ 10' 10"}, +4° \text{ 27' 24"} \rangle \\ m & \stackrel{\text{def}}{=} & \langle \text{meeting}, \ \{\text{true}\} \rangle \end{array} \quad ⌟$$

A constraint $\langle id, VAL \rangle$ is essentially a predicate over quantity id, and represents a single feature; features are no longer just symbols. One could argue that reasoning in terms of 'on-or-off' features at all is impractical here, and a more flexible model should be used; perhaps a simple mapping between quantities and values. However, a feature model gives us a discrete and finite state-space, just expressive enough to distinguish the conditions provided by the user. A more realistic representation might well involve continuous and infinite domains, depending on the environmental quantities involved.

Mapping predicates to propositional symbols, as we are doing now, is a trick from SMT (SAT Modulo Theory) [143], allowing us to reason about them propositionally. This technique requires us to impose some restrictions on possible feature configurations, because some combinations of constraints will —by their very nature— exclude or imply others. For instance, $\langle$ time, between 9:00 and 12:00 $\rangle$ and $\langle$ time, between 13:00 and 17:00 $\rangle$ would never appear in the same feature configuration. But the presence of either would also ensure the presence of t (Example 8.13). We can encode such restrictions in a feature model. For a source-code based product line, a feature model is set up manually, based on which features 'make sense' together, and which conceptually exclude or include each other (Section 4.2). A 'profile feature model' is fixed for a given set of constraints, derived from their respective theories (the T in SMT). We give the following definition for interests sake, but we will not require these details further in the chapter:

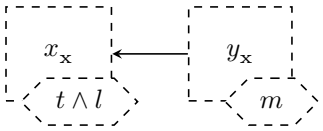


Figure 8.3: The Section 8.2.3 rule-set diagram.

8.14. Definition (Profile Feature Model): Given a set of profile features $\mathcal{F}$, the corresponding feature model Φ contains exactly all sets $F \subseteq \mathcal{F}$ such that

$$\forall id \in ID_{\mathrm{q}}\colon \overbrace{\varnothing \subset \underbrace{\Big(\mathrm{type}(id) \cap \bigcap_{\substack{\langle id, VAL\rangle \\ \in F}} VAL\Big) \not\subseteq \Big(\bigcup_{\substack{\langle id, VAL\rangle \\ \in \mathcal{F}\setminus F}} VAL\Big)}_{\text{and implied constraints are not excluded.}}}^{\text{included constraints are satisfiable}}$$

The only thing missing from our product line implementation now is a core product (Definition 4.9). To determine what a 'core profile' is, we first realize a simple truth: a smartphone application never has full control over the settings. The user can also manipulate them manually. So the core profile can basically be anything. There are a number of ways to capture this in the formalism, but we go for the simplest approach:

▷ **8.15. Definition (Manual Profile):** Introduce the value 'manual' $\in \mathcal{V}$, which is included in all types, i.e., we have 'manual' $\in \mathrm{type}(id)$ for all settings $id \in ID_{\mathrm{s}}$. Then define the *manual profile* $c \in \mathcal{P}_{DEV}$ as the constant that maps all identifiers to 'manual'. ⌟

▷ **8.16. Definition (Rule-set Implementation):** A *rule-set implementation* is a product line implementation $(\Phi, c, D, \prec, \gamma)$, fully defined by Definitions 8.10, 8.14 and 8.15. ⌟

The implementation of the Section 8.2.3 ruleset is depicted in Figure 8.3.

8.2.5 Product Line Specifications

Now what of product line specifications (Section 4.4)? They are a valuable concept in this chapter too. Just as *static* product line implementations were validated against them in Chapter 4 (page 96), so will *dynamic* product line implementations be validated against them in the next section.

However, since users of the profile manager express their requirements directly in the form of delta models, the static notion of product line correctness (Section 4.4.2) loses some meaning. The valuation function is defined directly in terms of the implementation —$\mathrm{V}(F) = [\![PLI]\!](F)$— making rule-set implementations correct by definition. But things become more interesting as we consider the correctness of dynamic product lines instead.

8.3 An Operational Semantics

> For the remainder of the chapter we assume that some product line specification $PLS = (\Phi, \mathrm{V})$ is given.

In this section we work on defining the structure and semantics of ADM-based *dynamic product line implementations*. We start by stating the problem we need to solve. We then proceed step-by-step as we explore possible solutions, using the profile management application as an example.

8.3.1 The Problem

The problem is as follows: Say we are running a dynamic product line. It is currently 'occupying' feature configuration $F_e \in \Phi$, as imposed by the environment, and exhibiting the behavior of a product $p \in \mathrm{V}(F_e)$. In other words, the product we are running is correct with regard to the *environmental feature configuration*. So far so good.

The environment could then impose a new feature configuration $F_e{}'$. This triggers our reconfiguration process, which is then responsible for updating the product p to some product $p' \in \mathrm{V}(F_e{}')$. Our goal is to find the best possible strategy for doing so; preferably one that is (potentially) efficient, since we are now in a runtime setting, where time and space matter.

For the Section 8.2 profile manager, the environmental feature configuration would change whenever the truth value of a constraint is 'flipped' by an environmental quantity receiving a new value. For example, in the Section 8.2.3 rule-set, if we have $F_e = \varnothing$ and it becomes 9:00, we would switch to feature configuration $F_e{}' = \{ t \}$.

8.3.2 An Operational Semantics

In order to reason about different strategies for updating the running product, we develop an operational semantics; one that might be 'inserted' at the point where an imperative program reaches the **`reconfigure`** statement described in the Introduction. However, we'll develop the semantics in the abstract setting of ADM, so we will not track memory state or control flow.

We first need to choose a *configuration space*, as this choice will determine the kind of properties we can express about the dynamic system. If not memory and control flow, what *do* we need to track? It might make sense if our configurations were *feature configurations*. After all, a dynamic product line is all about moving from one feature configuration to another. But then we would not be able to express the property that the running product is continually correct with regard to that feature configuration. The configurations need, at least, to contain that product too. So a minimal configuration space would be as follows:

8.17. Definition (Minimal Configurations): A *minimal configuration* is a pair $\langle F_e, p \rangle \in \Phi \times \mathcal{P}$ representing a dynamic product line state. F_e is the environmental feature configuration and p is the current product. ⌟

Side-note: The lack of a ▶ marker to the left of this definition indicates that it is not part of our final solution. We purposely explore a number of approaches in this section that turn out to be impractical. Doing so allows us to demonstrate useful concepts to build upon, and motivates the work that follows.

Do not let the term 'configuration' confuse you. The product p represents the actual running code (or, as the case may be, the actual running profile).

We make a distinction between *stable* and *unstable* configurations:

8.18. Definition (Configuration Stability): If a given configuration $\langle F_e, p \rangle$ has the property that $p \in \mathrm{V}(F_e)$, we call that configuration *stable*. Otherwise we call it *unstable*. ⌟

Recall that for the profile manager, this is the same as stating that a configuration is stable iff $F_e \ [\![PLI]\!] \ p$, because rule-set specifications are defined directly in terms of rule-set implementations (Section 8.2.5).

▷ **8.19. Example:** An example of a stable configuration would be

$$\langle \{ t, l \}, \{ \text{volume} \mapsto 5, \ldots \} \rangle$$

and an example of an unstable configuration would be

$$\langle \{ t, l, m \}, \{ \text{volume} \mapsto 1, \ldots \} \rangle$$

because all profiles in $\mathrm{V}(\{ t, l, m \})$ need volume set to 0. ⌟

To define a transition relation, we introduce *inference rules* (Notation 1.15, page 21). They are important reference points in the chapter, so we distinguish them typographically by printing their names in SMALL-CAPS and placing them inside a solid box where they are defined.

We distinguish between two different 'kinds' of transitions. There are *environmental transitions* $\xrightarrow{e}$, in which the environment switches to a new feature configuration, and *local transitions* $\xrightarrow{\ell}$, in which the product is updated in an attempt to regain a correct state. The full transition relation $\longrightarrow$ is defined as the smallest relation satisfying both an environmental inference rule and a local inference rule.

After an environmental transition, we will generally end up in an unstable configuration and will need to update the product:

8.20. Definition (Environmental Inference Rule): ENV-MIN

$$\frac{p \ \in \ \mathrm{V}(F_e)}{\langle F_e, p \rangle \ \xrightarrow{e} \ \langle F_e{}', p \rangle}$$

⌟

Note that before we allow any environmental transition, we require that the current configuration is stable. This assumption simplifies the formalism, and is reasonable because we can expect to reach local stability in relatively short amounts of time. It allows us to think about the entire process as an alternation between two distinct phases. In the first phase the environmental feature configuration changes; this always takes one environmental transition.

In the second phase we update the product, using zero or more local transitions to achieve stability. We know that a local phase will not be interrupted by environmental transitions:

$$\overbrace{\langle F_e, p\rangle \xrightarrow{e} \underbrace{\langle F_e', p\rangle \xrightarrow{\ell}{}^{*} \langle F_e', p'\rangle}_{\text{local}}}^{\text{environmental}} \xrightarrow{e} \langle F_e'', p'\rangle \xrightarrow{\ell}{}^{*} \langle F_e'', p''\rangle \xrightarrow{e} \cdots$$

8.3.3 Correctness

So let us now restate our goal more formally: we need to find a local inference rule. We know that we have a good one if it leads to a stable configuration in finite time. We call such a local inference rule *correct* with regard to the product line specification. This is the dynamic counterpart of 'static' product line correctness (Definition 4.20).

We distinguish between two levels of correctness, as we do for static product lines (Section 4.4.2). *Partial correctness* means that *if* the local transition relation ever gets stuck, it will be in a stable configuration (so an environmental transition can take place). *Total correctness* means that a local transition is *guaranteed* to get stuck in a stable configuration within finite steps.

▶ **8.21. Definition (Partial Correctness):** A given local inference rule is *partially correct* iff, for all configurations $\langle cn\rangle$, we have:

$$\langle cn\rangle \xrightarrow{\ell}{\not} \quad \Longrightarrow \quad \langle cn\rangle \text{ is stable}$$ ⌟

We could also state Definition 8.21 as follows:

> "a local inference rule is partially correct iff the transition relation $\xrightarrow{e} \cup \xrightarrow{\ell}$ never gets stuck"

but the current formulation is closer to the traditional meaning of partial correctness, and it allows us to refrain from referring to ENV-MIN.

▶ **8.22. Definition (Total Correctness):** A local inference rule is *totally correct* iff it is partially correct and there is no infinite local transition path (Definition 1.50, page 28):

$$\langle cn\rangle \xrightarrow{\ell}{\not}{}^{\infty}$$ ⌟

8.3.4 A Local Inference Rule: LOC-PRD

We now try to find a correct local inference rule. Let us first get an obvious (but naive) idea out of the way. We could use the same process we used to generate a product statically. We would assume that a static product line implementation *PLI* is given (Definition 4.10), totally correct with respect to *PLS* (Definition 4.20), and define a local inference rule as follows:

8.23. Definition (Local Inference Rule): LOC-PRD

$$\frac{\overbrace{p \notin \mathrm{V}(F_e)}^{\text{a}} \quad \overbrace{F_e \llbracket PLI \rrbracket\, p'}^{\text{b}}}{\langle F_e, p \rangle \xrightarrow{\ell} \langle F_e, p' \rangle}$$

⌟

This local transition can take place (a) from any unstable configuration (b) to a configuration with a product p' built from scratch to correspond with feature configuration F_e. We can prove that this rule is totally correct (Definitions 8.21 and 8.22):

8.24. Theorem: The LOC-PRD rule is totally correct.

Proof: First, assume that a given configuration $\langle F_e, p \rangle$ is locally stuck. This means we have the negation of LOC-PRD's premise: $p \in \mathrm{V}(F_e)$ or $\exists p' \colon F_e \llbracket PLI \rrbracket\, p'$. The latter is untrue by our static correctness assumption. By the former, all configurations that are locally stuck must also be stable. This gives us partial correctness.

If it is not stable, and therefore not stuck, we have $\langle F_e, p \rangle \xrightarrow{\ell} \langle F_e, p' \rangle$ with $F_e \llbracket PLI \rrbracket\, p'$ by LOC-PRD. By the assumed correctness of PLI we can conclude $p' \in \llbracket PLI \rrbracket(F_e) \subseteq \mathrm{V}(F_e)$. Since we use at most one transition to gain this stability, there is clearly no infinite local transition path, which gives us total correctness. □

But generating a new product on the fly this way will turn out to be too inefficient for non-trivial product lines. Recall that we need local transitions to be fast. Storing all possible products in memory beforehand and then dynamically switching to the correct one is also infeasible. In general the number of products will be exponential in the number of features. If we want to model industrial-scale dynamic product lines, we need to do better.

8.3.5 Difference-based Configurations

An alternative approach is to take the current product and transform it into a new one incrementally. This may be a lot more efficient, since we would be reusing the parts of the product that do not need to change. A transformation of a product is, of course, a delta. But how do we decide which delta to apply at every change? We are currently rather limited by the information in our configuration tuples. If we want to do this we need to add some bookkeeping. In particular, if we want to keep track of what changed in the environment, we'll need to store a *local feature configuration*. We can compare it to the environmental feature configuration and use their 'difference' to determine the delta or deltas that need to be applied (Figure 8.4):

▶ **8.25. Definition (Difference-based Configurations):** *Difference-based configurations* are triples $\langle F_e, F_\ell, p \rangle \in \Phi \times \Phi \times \mathcal{P}$ with an environmental feature configuration F_e, a *local feature configuration* F_ℓ and a current product p. ⌟

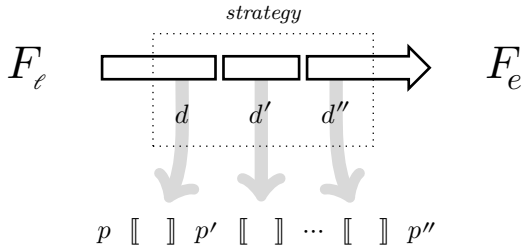


Figure 8.4: Illustrating the rôle of the local feature configuration in $\langle F_e, F_\ell, p \rangle$. If $p \notin \mathrm{V}(F_e)$, one or more deltas are derived from the difference between F_e and F_ℓ. These deltas are then used to transform the running product into valid product p''. This is presumably faster than building p'' from scratch. How to derive the proper sequence of deltas depends on our *strategy*.

Damiani et al. [63, 64] have a similar concept. They call the local feature configuration `CurrentConfiguration` and the environmental feature configuration `NextConfiguration`.

The difference between two feature configurations is their set-theoretic *symmetric difference* (Definition 1.1). We use it to measure their distance and how far we have progressed from one to the other:

▶ **8.26. Example (Symmetric Difference):** Take, as an example, two feature configurations $\{t\}, \{m\} \in \Phi_{\mathrm{x}}$. If we currently occupy $F_\ell = \{t\}$ and intend to reach $F_e = \{m\}$, we need to 'remove' feature t and 'add' feature m in the implementation of the current product. We represent this required work with $F_\ell \ominus F_e = \{t, m\}$. (We preserve the distinction between adding and removing by remembering the context of the operation.)

Conversely, if we start at $F_\ell = \{t\}$ and perform the work described by $F_\Delta = \{t, l\}$, we reach the state $F_\ell \ominus F_\Delta = \{l\}$. The symmetric difference operator has the interesting property that $F \ominus (F \ominus G) = G$, so we can use it for both types of operation. ⌟

Before we try to translate such a difference to a delta, we need to restate Definitions 8.18 and 8.20 to work with our new configurations. They are trivial changes, just adding and disregarding the additional element:

▶ **8.27. Definition (Configuration Stability):** If a given configuration $\langle F_e, F_\ell, p \rangle$ has the property that $p \in \mathrm{V}(F_e)$ then we call that configuration *stable*. Otherwise we call it *unstable*. ⌟

▶ **8.28. Definition (Environmental Inference Rule):** ENV-DIFF

$$\frac{p \in \mathrm{V}(F_e)}{\langle F_e, F_\ell, p \rangle \xrightarrow{e} \langle F_e', F_\ell, p \rangle}$$

⌟

When it comes to stability and environmental transitions, we do not care about our new bookkeeping element; it is left alone. Note that we do not need to redefine Definitions 8.21 and 8.22 because they were presented in a sufficiently general manner.

8.3.6 Dynamic Product Lines as Mealy Machines

We now need to decide, given a feature configuration difference, how to derive the delta or deltas that can transform the current product into a valid target product. We call this a *strategy* (Figure 8.4). We describe different strategies with a new model based on *Mealy machines*. Please have a look at Definition 1.52 (page 28) for the formal definition. This representation will be quite useful for describing as well as visualizing different strategies for running dynamic product lines.

It is worth noting that this type of graph (Figure 1.6, page 29) offers a completely different view of a product line than a delta diagram does. A delta diagram can be said to represent the design space, whereas Mealy machines represent the dynamic state-space.

We define local transitions of the operational semantics in terms of Mealy machine transitions. For this we use the following syntax for the Mealy-machine transition relation, so we need not use T and O directly:

▶ **8.29. Notation (Mealy Machine Transition Relation):** A Mealy machine tuple $(S, \Sigma, \Lambda, \mathrm{T}, \mathrm{O})$ induces a quaternary *Mealy machine transition relation* $\xrightarrow{-/-} \subseteq S \times \Sigma \times \Lambda \times S$ as follows. For all states $s, s' \in S$, input symbols $i \in \Sigma$ and output symbols $o \in \Lambda$:

$$s \xrightarrow{i/o} s' \qquad \overset{\text{def}}{\iff} \qquad \mathrm{T}(s,i) = s' \ \wedge \ \mathrm{O}(s,i) = o \qquad \lrcorner$$

We can now define a local inference rule in terms of the $\xrightarrow{i/o}$ relation of a Mealy machine. But how do we define such a *DPL Mealy machine*? The tuple has five ingredients. The first three are simple enough:

- $S = \Phi$: The states of our machine are feature configurations. The local feature configuration F_ℓ is the *current state*. F_e is the *target state*. In a diagram we annotate these two states as in Figure 8.5.
- $\Sigma = \mathrm{Pow}(\mathcal{F})$: The input we want to process at each transition is the difference —or part of the difference— between feature configurations F_ℓ and F_e. We denote such a difference by $F_\Delta \subseteq \mathcal{F}$.
- $\Lambda = \mathcal{D}$: As we move F_ℓ towards F_e in the machine, we'd like to get, as output, a light-weight delta (or deltas) to update the product.

The output function O requires some thought. Given a current product and feature difference, which delta do we use to update the product?

Since a delta d can be non-deterministic, its application to the current product p could result in more than one possible next product $p' \in [\![d]\!](p)$. We at least need all of those to be be correct: $[\![d]\!](p) \subseteq \mathrm{V}(F_e)$. But at build time we can't be sure what product p is. It could itself be any of the products generated by the previous transition, and so on. In other words, if we want to reason locally, we'll need to choose an invariant on our configurations $\langle F_e, F_\ell, p \rangle$ to give us more information about our current product. The invariant will need to hold in the initial configuration and every transition will need to maintain it. We will use the *local consistency* invariant:

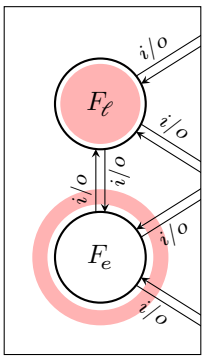


Figure 8.5: Local and environmental marking of states in a DPL Mealy machine diagram. Local feature configuration F_ℓ is shaded and environmental feature configuration F_e has a ring around it. During a local transition, imagine our general strategy as the local state trying to reach the ring by taking any direct path that leads in the right direction.

▶ **8.30. Definition (Local Consistency):** A difference-based configuration $\langle F_e, F_\ell, p \rangle$ is *locally consistent* iff $p \in \mathrm{V}(F_\ell)$. ⌟

For specific systems there may be stronger invariants that are more appropriate, but in general, local consistency is the best we can do. We know now that every configuration has a product consistent with F_ℓ and we can use that to choose the right delta. We need to derive one that transforms *any* product from a locally consistent configuration to a correct target product, i.e., we need an effective procedure for delta derivation (Section 2.4.3):

▶ **8.31. Definition (Derived Delta Operator):** A *derived delta operator* is a binary operator $\mapsto\colon \mathrm{Pow}(\mathcal{P}) \times \mathrm{Pow}(\mathcal{P}) \to \mathcal{D}$ that returns a derived delta for all product sets $P, P' \subseteq \mathcal{P}$:

$$(P \mapsto P') \quad \in \quad (P \Mapsto_{\text{tot}} P')$$

where $\Mapsto_{\text{tot}}$ specifies a set of derived deltas (Definition 2.31). ⌟

From now on we assume that such a procedure is implemented for the given deltoid. We can define one for the profile deltoid as follows:

▷ **8.32. Definition (Derived Profile Delta Operator):** The *derived profile delta operator* $\mapsto_{DEV}\colon \mathrm{Pow}(\mathcal{P}_{DEV}) \times \mathrm{Pow}(\mathcal{P}_{DEV}) \to \mathcal{D}_{DEV}$ takes a finite set of source profiles and a nonempty finite set of target profiles and produces a profile delta that transforms any profile from the source set into an arbitrary profile from the target set. For all product sets $P \subseteq \mathcal{P}_{DEV}$, products $p' \in \mathcal{P}_{DEV}$ and identifiers $id \in ID_{\mathrm{s}}$:

$$(P \mapsto_{DEV} \{p', \ldots\})(id) \quad \stackrel{\text{def}}{=} \quad \begin{cases} p'(id) & \text{if } \exists p \in P\colon p(id) \neq p'(id) \\ \bot & \text{otherwise} \end{cases}$$

The resulting delta remains undefined for the settings on which P and p' agree, and favors p' for the rest. The product p' is chosen arbitrarily from the right-hand operand. ⌟

Generally, given such a procedure, we can define an output function O that produces an appropriately derived delta $\mathrm{O}(F_\ell, F_\Delta) = \mathrm{V}(F_\ell) \mapsto \mathrm{V}(\mathrm{T}(F_\ell, F_\Delta))$. We assume that this is done at build-time for all relevant feature differences, so they can simply be looked up at run-time.

As for our final ingredient: the definition of the transition function T is what determines our further strategy. The output function O will remain fixed, though its preimage will adapt to correspond with T. So we can now define the concept of a *DPL Mealy machine* parametrized on T:

▶ **8.33. Definition (DPL Mealy Machine):** Given a *transition function* $\mathrm{T}: \Phi \times \mathrm{Pow}(\mathcal{F}) \rightharpoonup \Phi$, we define the corresponding *DPL Mealy machine*

$$\mathrm{MM}(\mathrm{T}) \quad \stackrel{\text{def}}{=} \quad (\Phi, \mathrm{Pow}(\mathcal{F}), \mathcal{D}, \mathrm{T}, \mathrm{O})$$

where $\mathrm{O}: \Phi \times \mathrm{Pow}(\mathcal{F}) \rightharpoonup \mathcal{D}$ is defined as

$$\mathrm{O}(F_\ell, F_\Delta) \quad \stackrel{\text{def}}{=} \quad \begin{cases} \mathrm{V}(F_\ell) \mapsto \mathrm{V}(\mathrm{T}(F_\ell, F_\Delta)) & \text{if } (F_\ell, F_\Delta) \in \mathrm{pre}(\mathrm{T}) \\ \bot & \text{otherwise} \end{cases}$$ ⌟

And finally, we define a local inference rule in terms of a DPL Mealy machine, also parametrized on T:

▶ **8.34. Definition (Local Inference Rule):** LOC-DIFF(T)

$$\frac{\overbrace{p \notin \mathrm{V}(F_e)}^{\text{a}} \qquad \overbrace{\varnothing \subset F_\Delta \subseteq F_\ell \ominus F_e}^{\text{b}} \qquad \overbrace{F_\ell \xrightarrow{F_\Delta \,/\, d} F_\ell{}'}^{\text{c}} \qquad \overbrace{p \,[\![\, d \,]\!]\, p'}^{\text{d}}}{\langle F_e, F_\ell, p \rangle \xrightarrow{\ell} \langle F_e, F_\ell{}', p' \rangle}$$

with $\xrightarrow{i/o}$ from Mealy machine MM(T) (Definition 8.33). ⌟

(a) Starting from an unstable configuration $\langle F_e, F_\ell, p \rangle$, (b) some nonempty subset of $F_\ell \ominus F_e$ is chosen as input symbol F_Δ. (c) Given that input symbol from current state F_ℓ, the DPL Mealy machine reaches a state $F_\ell{}' \in \Phi$ and generates a delta $d \in \mathcal{D}$ as output symbol. (d) This delta can transform product p into product p', forming a possible next configuration $\langle F_e, F_\ell{}', p' \rangle$.

At this point a general recapitulation is in order: The behavior of a dynamic product line is modeled by a configuration space and a transition relation on that space (Notations 1.48 and 1.49). We define the transition relation based on environmental inference rule ENV-DIFF and local inference rule LOC-DIFF(T) (Definitions 8.28 and 8.34). The local inference rule is based on a Mealy machine, parametrized on its transition function T (Definition 8.33). The remaining sections will be spent trying to find the optimal function T.

Before we explore the first candidate, we prove a number of useful properties about the local inference rule, which will help us in the upcoming correctness proofs. First, we prove that it maintains local consistency, purely by our fixed choice of output function:

▶ **8.35. Lemma (Local Consistency by LOC-DIFF):** Given any transition function T, the transition relation defined by LOC-DIFF(T) maintains the local consistency invariant (Definition 8.30).

Proof: Assume that $\langle F_e, F_\ell, p \rangle$ is locally consistent, so $p \in \mathrm{V}(F_\ell)$. Take any transition $\langle F_e, F_\ell, p \rangle \xrightarrow{\ell} \langle F_e, F_\ell{}', p' \rangle$. We know from Definition 8.33 that $p \llbracket d \rrbracket p'$ for some delta $d \in \mathrm{V}(F_\ell) \mapsto\!\!\!\!\to_{\mathrm{tot}} \mathrm{V}(F_\ell{}')$. So by our assumption $p \in \mathrm{V}(F_\ell)$ and by Definition 8.31, we have $p' \in \mathrm{V}(F_\ell{}')$. That means $\langle F_e, F_\ell{}', p' \rangle$ is also locally consistent. □

Next, there is a certain property all our choices of transition function *should* exhibit. They will all take a *direct path* from the local to the environmental feature configurations. This means that the difference between the two sets strictly decreases in size from one configuration to the next:

▶ **8.36. Definition (Direct Path):** A transition function $\mathrm{T}\colon \Phi \times \mathrm{Pow}(\mathcal{F}) \rightharpoonup \Phi$ takes a *direct path* iff, for all configurations $\langle F_e, F_\ell, p \rangle$ and a transition relation $\xrightarrow{\ell}$ defined by LOC-DIFF(T), we have:

$$\langle F_e, F_\ell, p \rangle \xrightarrow{\ell} \langle F_e, F_\ell{}', p' \rangle \quad \Longrightarrow \quad F_e \ominus F_\ell \supset F_e \ominus F_\ell{}' \qquad \lrcorner$$

This will help us prove total correctness by use of the following lemma:

▶ **8.37. Lemma (Direct Path Convergence):** Any transition function that takes a direct path (Definition 8.36), allows no infinite local transition path:

$$\nexists \langle F_e, F_\ell, p \rangle \xrightarrow{\ell}{}^{\infty}$$

Proof: By wellfoundedness of $\subset$. The finite set $F_e \ominus F_\ell$ can only shrink until it is empty. □

So basically, for any partially correct local inference rule that takes a direct path, we get total correctness for free.

8.3.7 A Local Inference Rule: LOC-DIFF($\mathrm{T_f}$)

The first obvious strategy is to take the difference between the local and environmental feature configurations $F_\Delta = F_e \ominus F_\ell$ directly as input symbol for the Mealy machine. So, if $F_\ell = \{ t \}$ and $F_e = \{ m \}$ we take a single transition with input symbol $F_\Delta = \{ t, m \}$:

8.38. Definition: A local inference rule LOC-DIFF′(T) is the same as rule LOC-DIFF(T) (Definition 8.34), but with the additional premise that $F_\Delta = F_e \ominus F_\ell$. ⌟

This premise is only temporary, because the strategy will turn out to be impractical. Nonetheless, a brief exploration of it will be instructive. We can define the transition function as follows:

8.39. Definition (Full Difference Transition Function): Define the *full difference transition function* $\mathrm{T_f}\colon \Phi \times \mathrm{Pow}(\mathcal{F}) \rightharpoonup \Phi$ as follows, for all $F_\ell \in \Phi$ and $F_\Delta \subseteq \mathcal{F}$:

$$\begin{aligned} \mathrm{T_f}(F_\ell, F_\Delta) &\stackrel{\text{def}}{=} F_\ell \ominus F_\Delta \\ \mathrm{pre}(\mathrm{T_f}) &\stackrel{\text{def}}{=} \{ (F_\ell, F_\Delta) \mid F_\ell \ominus F_\Delta \in \Phi \} \end{aligned}$$

⌟

There are two almost separate aspects to defining a transition function: defining its output and defining its preimage. The output specified above makes it clear that the new local transition rule $\boxed{\textsc{loc-diff}'(\mathrm{T_f})}$ (Definitions 8.38 and 8.42) moves from F_ℓ to F_e in a single step after every environment change. We chose the preimage so that the output is guaranteed to be a valid feature configuration, rather than some arbitrary feature set.

8.40. Lemma (Direct Path by $\textsc{loc-diff}'(\mathrm{T_f})$): The inference rule $\textsc{loc-diff}'(\mathrm{T_f})$ takes a direct path (Definition 8.36).

Proof: The difference F_Δ provided as an input symbol is a nonempty set (Definition 8.34b). By Definition 8.39 the new local feature configuration is $F_\ell{}' = F_\ell \ominus F_\Delta = F_\ell \ominus (F_\ell \ominus F_e) = (F_\ell \ominus F_\ell) \ominus F_e = F_e$. In short, $F_\ell{}' = F_e$, so the new difference $F_e \ominus F_\ell{}' = \varnothing$ is empty, making it strictly smaller than F_Δ.□

With this results we prove total correctness as defined by Definitions 8.21 and 8.22:

8.41. Theorem: The rule $\textsc{loc-diff}'(\mathrm{T_f})$ is totally correct.

Proof: Assume that a given configuration $\langle F_e, F_\ell, p \rangle$ is stuck. Since $\textsc{loc-diff}'(\mathrm{T_f})$ (Definitions 8.38 and 8.39) is our only local inference rule, we have the negation of one of its premises. So we have one of the following:

a. The configuration is already stable: $p \in \mathrm{V}(F_e)$,
b. there is no F_Δ, because $F_\ell = F_e$,
c. the $\mathrm{T_f}$ function accepts none: $\nexists \varnothing \subset F_\Delta \subseteq F_\ell \ominus F_e$: $(F_\ell, F_\Delta) \in \mathrm{pre}(\mathrm{T_f})$, or
d. the generated delta does not accept the current product: $[\![d]\!](p) = \varnothing$.

By Definition 8.39 it cannot be (c), and by Definition 8.33 it cannot be (d). So by the process of elimination we have $p \in \mathrm{V}(F_e) \vee F_\ell = F_e$. The former would give us our result directly. Given the latter, we have $p \in \mathrm{V}(F_\ell)$ by local consistency (Lemma 8.35), and therefore $p \in \mathrm{V}(F_e)$. So we know that when $\textsc{loc-diff}'(\mathrm{T_f})$ is stuck on a configuration, that configuration must be stable, giving us partial correctness.

We have total correctness by the direct path property (Lemmas 8.37 and 8.40). □

This is a pattern of proof we will use more often:

- Prove that a stuck configuration is stable by invoking local consistency and the negation of one of the premises of Definition 8.34. This gives us partial correctness.
- Prove that the transition function takes a direct path (Definition 8.36), giving us total correctness.

A Mealy machine diagram for $\textsc{loc-diff}(\mathrm{T_f})$ would be quite unreadable. It has an excessive number of transitions: $|\mathrm{pre}(\mathrm{T_f})| = |\Phi|^2 - |\Phi| = 2^{2|\mathcal{F}|} - 2^{|\mathcal{F}|}$, quadratic in the number of feature configurations, so exponential in the number of features. This is not surprising, given that the input alphabet we chose is $\mathrm{Pow}(\mathcal{F})$. This also means that we would have to store a lot of deltas. Too many.

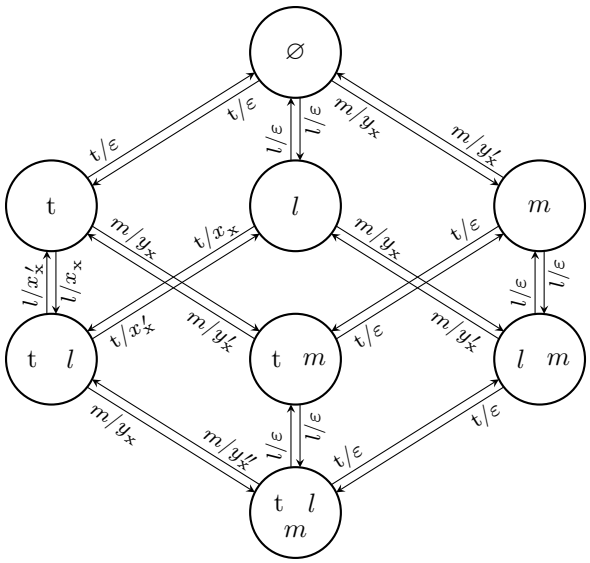


Figure 8.6: The DPL Mealy machine $\mathrm{MM}(\mathrm{T_s})$ for the Section 8.2.3 example.

The key insight here is that we do not need to reach the target product in a single transition. We can use multiple local transitions to get there. And with the concepts introduced up to this point, it will be relatively simple to define a new rule for that.

8.3.8 A Local Inference Rule: LOC-DIFF-ND($\mathrm{T_s}$)

We now reduce the number of Mealy-machine transitions by taking one local transition per *feature* rather than one per *feature-set*. This is at the cost of using multiple local transitions during a single phase if necessary. It will bring the number of transitions in the Mealy machine down from $\left(2^{2^{|\mathcal{F}|}} - 2^{|\mathcal{F}|}\right)$ to $\left(|\mathcal{F}| \times 2^{|\mathcal{F}|}\right)$, a significant difference in practice, even though it is still exponential. So for the profile manager, if $F_\ell = \{t\}$ and $F_e = \{m\}$ we take one local transition for 'it became 17:00' and one for 'a meeting has started', even if both occur simultaneously.

To maintain consistency, we do not change the input alphabet to $\mathcal{F}$, but will simply use singleton sets. The transition function is otherwise similar to $\mathrm{T_f}$:

8.42. Definition (Singleton Transition Function): Define the *singleton transition function* $\mathrm{T_s}$: $\Phi \times \mathrm{Pow}(\mathcal{F}) \rightharpoonup \Phi$ as follows, for all $F_\ell \in \Phi$ and $f_\Delta \in \mathcal{F}$:

$$\begin{aligned} \mathrm{T_s}(F_\ell, \{f_\Delta\}) &\stackrel{\text{def}}{=} F_\ell \ominus \{f_\Delta\} \\ \mathrm{pre}(\mathrm{T_s}) &\stackrel{\text{def}}{=} \{\, (F_\ell, \{f_\Delta\}) \mid F_\ell \ominus \{f_\Delta\} \in \Phi \,\} \end{aligned}$$ ⌟

The DPL Mealy machine $\mathrm{MM}(\mathrm{T_s})$ (Definitions 8.33 and 8.42) for our running example is shown in Figure 8.6. Observe that the features t, l, m each represent one 'dimension' in the diagram. For readability we omited the 'set braces',

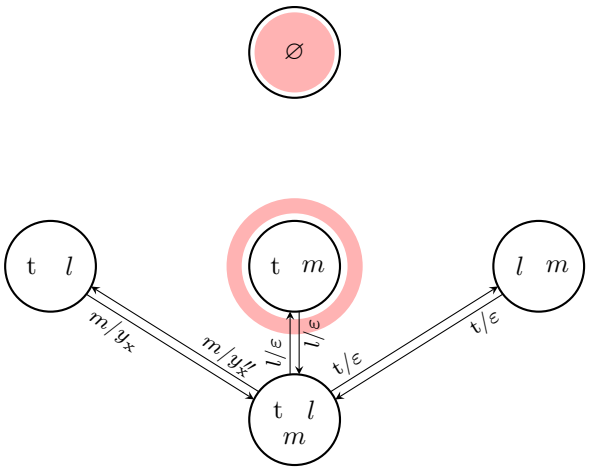


Figure 8.7: A DPL Mealy machine $\mathrm{MM}(\mathrm{T_s})$ based on an some unrestricted feature model. Note that $F_e = \{t, m\}$ cannot be reached from $F_\ell = \varnothing$.

since we only use singleton input sets for $\mathrm{T_s}$. Three known deltas are used in this machine: ε, x_x and y_x (Section 8.2.3). The new deltas x'_x, y'_x and y''_x are:

$$
\begin{aligned}
x'_\mathrm{x} &= \{\ \text{‘volume’} \mapsto \text{‘manual’}\ \}, \\
y'_\mathrm{x} &= \left\{ \begin{array}{r} \text{‘volume’} \mapsto \text{‘manual’}, \\ \text{‘foreground app’} \mapsto \text{‘manual’} \end{array} \right\}, \\
y''_\mathrm{x} &= \left\{ \begin{array}{r} \text{‘volume’} \mapsto 5, \\ \text{‘foreground app’} \mapsto \text{‘manual’} \end{array} \right\}.
\end{aligned}
$$

They reverse the effects of their counterparts. Since semantic profile deltas are not surjective —they just overwrite any value that was there before—, they do not have an inverse without taking their context into account. That’s why we need both y'_x and y''_x.

Let’s investigate local inference rule $\boxed{\textsc{loc-diff}(\mathrm{T_s})}$ (Definitions 8.34 and 8.42). As it turns out, there is a problem with it: it is not correct for arbitrary feature models. Now that we are taking small ‘feature-sized’ steps through the Mealy machine, it is no longer certain all states are reachable. If $\Phi \neq \mathrm{Pow}(\mathcal{F})$ we would be missing some intermediate states we need to land on. We cannot allow Mealy machines such as the one in Figure 8.7, for example. So for this strategy we need to restrict the feature model to $\Phi = \mathrm{Pow}(\mathcal{F})$. The transition function *does* take direct path:

8.43. Lemma (Direct Path by $\textsc{loc-diff}(\mathrm{T_s})$): The inference rule $\textsc{loc-diff}(\mathrm{T_s})$ takes a direct path (Definition 8.36).

Proof: The difference $\{f_\Delta\}$ provided as an input symbol is obviously a nonempty set. By Definition 8.42 we have $F_\ell{}' = F_\ell \ominus \{f_\Delta\}$, so the new difference is $F_e \ominus F_\ell{}' = F_e \ominus (F_\ell \ominus \{f_\Delta\}) = (F_e \ominus F_\ell) \ominus \{f_\Delta\}$. Since $\{f_\Delta\} \subseteq F_e \ominus F_\ell$, we have $(F_e \ominus F_\ell) \ominus \{f_\Delta\} = (F_e \ominus F_\ell) \setminus \{f_\Delta\}$, strictly smaller than $F_e \ominus F_\ell$. □

And this leads to total correctness, much as before, as long as we restrict the feature model as described above:

8.44. Theorem: For feature model $\Phi = \mathrm{Pow}(\mathcal{F})$, the rule LOC-DIFF($\mathrm{T_s}$) is totally correct.

Proof: This proof proceeds much as the one for Theorem 8.41, so we leave some simple steps out. Assume that a given configuration $\langle F_e, F_\ell, p \rangle$ is stuck. By negating the premises, we have either of the following:

a. The configuration is already stable: $p \in \mathrm{V}(F_e)$,
b. there is no F_Δ because $F_\ell = F_e$, or
c. the $\mathrm{T_s}$ function accepts none: $\nexists f_\Delta \in F_\ell \ominus F_e \colon (F_\ell, \{f_\Delta\}) \in \mathrm{pre}(\mathrm{T_s})$.

It cannot be (c), because even though T accepts only singleton sets, we are assuming a complete feature model Φ, so all singleton sets are valid input symbols (Definition 8.42). We therefore have $p \in \mathrm{V}(F_e) \ \vee \ F_\ell = F_e$, giving us partial correctness as before.

And as before, we get total correctness from Lemmas 8.37 and 8.43. □

We also know that any local transition-path starting from $\langle F_e, F_\ell, p \rangle$ will always reach a configuration $\langle F_e, F_e, p' \rangle$ in exactly $|F_e \ominus F_\ell|$ steps.

8.3.9 A Local Inference Rule: LOC-DIFF($\mathrm{T_m}$)

Our next goal is to drop the restriction on the feature model imposed in Section 8.3.8. So we want to go from F_ℓ to F_e, but somewhere on an otherwise direct path we are missing an intermediate state we need to pass through to reach the target state. We are going to add extra transitions to solve this problem — just enough to regain reachability. Those extra transitions will have $|F_e \ominus F_\ell| > 1$.

We are entitled to ask, however: can we still keep using singleton sets as *input symbols*? As long as a single feature unambiguously determines the right direction, we could always define T to jump any additional distance required, i.e., so that $F_\ell \xrightarrow{\{f_\Delta\}/d} F_\ell{}'$ with $\{f_\Delta\} \subset F_\ell \ominus F_\ell{}'$. But the answer is no. There are situations where a single feature cannot unambiguously determine a transition. Take, for example, a feature model $\Phi'_{\mathrm{x}} = \{\varnothing, \{t,l\}, \{t,m\}, \{l,m\}, \{t,l,m\}\}$ with the Mealy machine from Figure 8.7. With $F_\ell = \varnothing$ and $F_e = \{t,m\}$, choosing either of $t, m \in (F_\ell \ominus F_e)$ as the sole input symbol would not be enough to determine the next state. We need the full information $\{t,m\}$ for that.

The following transition function has a preimage that is minimal, unique and preserves reachability:

▶ **8.45. Definition (Minimal Transition Function):** Define the *minimal transition function* $\mathrm{T_m} \colon \Phi \times \mathrm{Pow}(\mathcal{F}) \rightharpoonup \Phi$ as follows, for all $F_\ell \in \Phi$ and $F_\Delta \subseteq \mathcal{F}$:

$$\mathrm{T_m}(F_\ell, F_\Delta) \stackrel{\text{def}}{=} F_\ell \ominus F_\Delta$$

$$\mathrm{pre}(\mathrm{T_m}) \stackrel{\text{def}}{=} \left\{ (F_\ell, F_\Delta) \;\middle|\; \begin{array}{l} F_\ell \ominus F_\Delta \in \Phi \quad \wedge \\ \nexists\, \varnothing \subset F_\Delta{}' \subset F_\Delta \colon \ F_\ell \ominus F_\Delta{}' \ \in \ \Phi \end{array} \right\}$$

⌟

The additional restriction, when compared to Definition 8.39, ensures that the only transitions that are preserved are those that the feature model does not allow taking in smaller steps. If there *are* smaller steps, those will become transitions themselves. So Definition 8.45 covers the 'one feature difference' transitions we had before, as well as new transitions required to bridge larger gaps.

We lift the restrictions from our input symbols as well as our feature model by using local inference ruleset $\boxed{\textsc{loc-diff}(\mathrm{T_m})}$ (Definitions 8.33 and 8.45). Now for the necessary correctness proofs:

▶ **8.46. Lemma (Direct Path by LOC-DIFF($\mathrm{T_m}$)):** The inference rule LOC-DIFF($\mathrm{T_m}$) takes a direct path (Definition 8.36).

Proof: Almost identical to the Lemma 8.43 proof; just replace $\{f_\Delta\}$ with F_Δ.□

▶ **8.47. Theorem:** The LOC-DIFF-ND(MM($\mathrm{T_m}$)) rule is totally correct.

Proof: Again, this proof is quite similar to those for Theorems 8.41 and 8.44. To summarize, if $\mathrm{T_m}$ can always accept some non-empty input symbol $F_\Delta \subseteq F_\ell \ominus F_e$ (Definition 8.34c), then a stuck state is also stable (Definition 8.34a and 8.34b), giving us partial correctness. We then get total correctness from the direct path property as before.

In this case, Definition 8.45 only excludes an input symbol if an equivalent string of smaller ones is also available. So there is always at least one. This gives us our desired result. □

We also know that any local transition-path starting from $\langle F_e, F_\ell, p \rangle$ will always reach a configuration $\langle F_e, F_e, p' \rangle$ in at most $|F_e \ominus F_\ell|$ steps.

The Mealy machine from Figure 8.7 would now be constructed as in Figure 8.8. Note that $\mathrm{T_m}$ gives us the necessary transitions to bridge the distance. If we want to get from $F_\ell = \varnothing$ to $F_e = \{t, l, m\}$, any of the three possible local transition paths will take us there properly:

$$\langle F_e, \varnothing, p \rangle \begin{cases} \xrightarrow{\ell} \langle F_e, \{t,l\}, [\![x_{\mathrm{x}}]\!](p) \rangle \xrightarrow{\ell} \langle F_e, \{t,l,m\}, [\![y_{\mathrm{x}} \cdot x_{\mathrm{x}}]\!](p) \rangle \\ \xrightarrow{\ell} \langle F_e, \{t,m\}, [\![y_{\mathrm{x}}]\!](p) \rangle \xrightarrow{\ell} \langle F_e, \{t,l,m\}, [\![\varepsilon \cdot y_{\mathrm{x}}]\!](p) \rangle \\ \xrightarrow{\ell} \langle F_e, \{l,m\}, [\![y_{\mathrm{x}}]\!](p) \rangle \xrightarrow{\ell} \langle F_e, \{t,l,m\}, [\![\varepsilon \cdot y_{\mathrm{x}}]\!](p) \rangle \end{cases}$$

Note that $y_{\mathrm{x}} \cdot x_{\mathrm{x}} = y_{\mathrm{x}} = \varepsilon \cdot y_{\mathrm{x}}$ (Section 8.2.3), so we reach the same result regardless.

8.4 Cost and Optimization

We now allow an unrestricted feature model and have reduced the number of transitions to a reasonable amount. In this section, we examine a technique for optimizing the Mealy machine even further, in a way that is particularly effective when the activity of 'monitoring' the environment for change carries with it a certain cost that needs to be minimized.

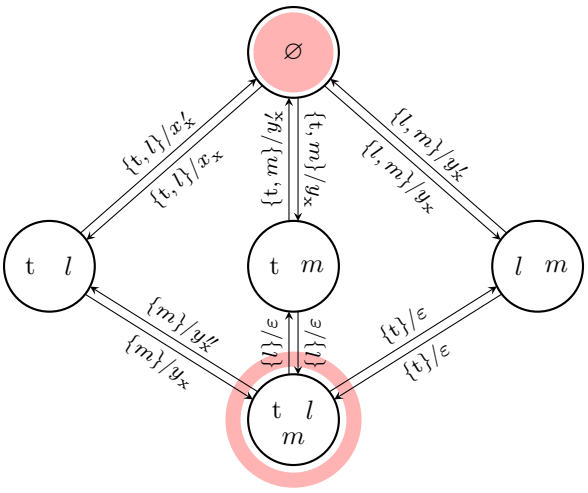


Figure 8.8: The DPL Mealy machine $\mathrm{MM}(\mathrm{T_m})$ based on the same feature model as $\mathrm{MM}(\mathrm{T_s})$ from Figure 8.7. We now have full reachability.

8.4.1 Cost

The cost of occupying a state in a DPL Mealy machine is that of monitoring the features from the accepted input-symbols for change. I posit that monitoring some features can be more expensive than monitoring others.

For example, it is more draining to the battery of a smartphone to constantly monitor GPS location (l) than it is to intermittently check the calendar for meetings (m), since the GPS receiver needs to constantly receive signals and the calendar is internal. But checking the calendar is still more costly keeping track of the time (t). The operating system does that anyway, and can notify our app through an alarm-subscription service.

▶ **8.48. Definition (Cost):** Assume some *cost domain* C measured over time, with an additive neutral element 0. Given DPL Mealy machine $(\Phi, \mathrm{Pow}(\mathcal{F}), \mathcal{D}, \mathrm{T}, \mathrm{O})$ we introduce a function $\mathrm{cost}\colon \Phi \times \mathcal{F} \to C$. The value $\mathrm{cost}(F_\ell, f)$ represents the cost of monitoring feature f from state F_ℓ. A feature is only monitored from a state if that state has an outgoing transition with f in its input symbol. So if the current state does not have such a transition, the cost is 0:

$$\nexists F_\Delta \subseteq \mathcal{F}\colon\ f \in F_\Delta\ \wedge\ (F_\ell, F_\Delta) \in \mathrm{pre(T)} \quad \Longrightarrow \quad \mathrm{cost}(F_\ell, f)\ =\ 0 \qquad \lrcorner$$

We want to maintain generality in the definition, but for the profile manager, the cost-domain is usually *power* in watt, i.e. joules per second. It is also likely that the cost of monitoring a profile feature depends solely on which quantity is being constrained (Definition 8.4), and is independent from the local feature configuration F_ℓ and the set of values of the constraint. When that is the case we can use a shorter notation:

▷ **8.49. Notation (Cost of Monitoring Quantities):** For all device constraints $\langle id_\mathrm{q}, VAL\rangle \in C_{DEV} = \mathcal{F}$ and feature configurations $F \in \Phi$:

$$\mathrm{cost}(id_\mathrm{q}) \quad \stackrel{\text{def}}{=} \quad \mathrm{cost}(F, \langle id_\mathrm{q}, VAL\rangle) \qquad \lrcorner$$

We minimize the cost of running a dynamic product line by removing costly transitions from our Mealy machine through additional restrictions on pre(T), but only so far as we can maintain partial correctness (i.e., so far as we can avoid getting stuck in unstable configurations). Features only need to be monitored when they become relevant.

In our example product line (Figure 8.6) we need to apply delta x_{x} only when we are both in a certain GPS location (l) and at a certain time (t). Either constraint satisfied on its own does not modify the profile. So it makes sense to only start monitoring GPS (the more costly quantity), when it is already the right time. For this to work, we just have to check the GPS immediately when we reach the proper time, since the transition event may have occurred without the device observing it.

8.4.2 Optimization through Refinement

The trick to optimization is to realize that we do not need to reach a configuration where $F_\ell = F_e$, even though that has always been our goal in Section 8.3. There is another way to get a stable configuration. It is sufficient if we occupy a locally consistent configuration $\langle F_e, F_\ell, p \rangle$ with $\mathrm{V}(F_\ell) \subseteq \mathrm{V}(F_e)$. Such an F_ℓ is a state from which we might have $F_\ell \xrightarrow{F_\Delta/\varepsilon} F_e$, i.e. get the neutral delta ε as an output symbol if we actually did make the transition to F_e. Applying ε to a product does not change it, so we can sometimes remove transitions like that to avoid having to monitor the features in F_Δ.

In general, $\mathrm{V}(F_\ell) \subseteq \mathrm{V}(F_e)$ does not imply $\mathrm{V}(F_e) \subseteq \mathrm{V}(F_\ell)$. However, for a device rule-set that yields an unambiguous static product line implementation, it *does*. For every feature configuration there is only one profile that satisfies it, so we can set up an equivalence relation between feature configurations:

▷ **8.50. Definition (Equivalence):** Two feature configurations $F, G \in \Phi$ are *equivalent*, denoted $F \equiv G$, iff $\mathrm{V}(F) = \mathrm{V}(G)$. ⌟

This equivalence can be decided at build-time and is represented in diagrams by a gray background which marks equivalence classes (Figure 8.9). This makes optimization through refinement more intuitive: the system only needs to reach a state in the same equivalence class as F_e.

8.4.3 Optimization through Redundancy

There is another kind of transition we could eliminate. Recall that we explained in Definition 4.10 why we could not go back to using singleton sets as input symbols. Sometimes we simply need all information in $F_\ell \ominus F_\ell{}'$ to unambiguously find the next state in the Mealy machine. So we kept defining T to expect the full difference as an input symbol, i.e., for every $F_\ell \xrightarrow{F_\Delta/d} F_\ell{}'$ we had $F_\Delta = F_\ell \ominus F_\ell{}'$.

But that is not required. It would be enough if the input symbol was *included* in the state-difference: $F_\Delta \subseteq F_\ell \ominus F_\ell{}'$. This is another opportunity to reduce the cost of a Mealy machine, because we may only need to monitor *some* of the features in $F_\ell \ominus F_\ell{}'$ to trigger the full transition.

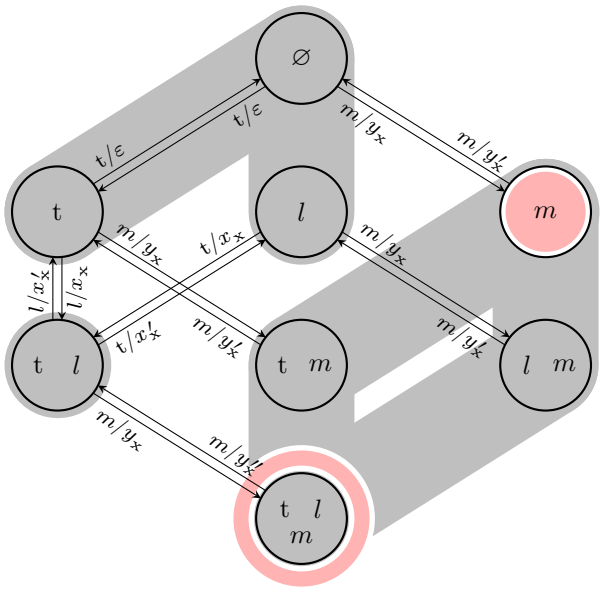


Figure 8.9: A DPL Mealy machine with a transition function $\mathrm{T_o}$. Equivalence classes are marked: $\varnothing \equiv \{t\} \equiv \{l\}$ and $\{m\} \equiv \{t, m\} \equiv \{l, m\} \equiv \{t, l, m\}$.

8.4.4 A Local Inference Rule LOC-DIFF($\mathrm{T_o}$)

But this is where we hit a roadblock, because in an abstract setting there is not just one transition function to find. Finding the best $\mathrm{T_o}$ is an *optimization problem*. The goal is to choose one that minimizes the cost of running the dynamic product line, while preserving the property that $\boxed{\textsc{loc-diff}(\mathrm{T_o})}$ only gets stuck on configurations $\langle F_e, F_\ell, p \rangle$ with $\mathrm{V}(F_\ell) \subseteq \mathrm{V}(F_e)$.

The correctness-proof of such a transition rule will be very similar to those already covered, except that this time, the usual process of elimination will leave us with only the negation of Definition 8.34a: $p \in \mathrm{V}(F_e)$.

For the more specific domain of the profile manager, we can provide an example of $\mathrm{T_o}$. Let us make a couple of assumptions:

- The cost of monitoring f depends solely on the quantity being monitored.
- During the average lifetime of the dynamic product line, all states are occupied for roughly the same amount of time.
- The following holds for our device (Notation 8.49):

$$\text{cost(time)} \quad < \quad \text{cost(meeting)} \quad < \quad \text{cost(gps)}$$

So to define $\mathrm{T_o}$, we start with $\mathrm{T_m}$. We first remove l/ε transitions, then m/ε transitions, then t/ε transitions, so long as the $\mathrm{T_o}$ reachability between equivalence classes is preserved.

As you can see in Figure 8.9, we are able to remove ten transitions as compared to Figure 8.6, significantly reducing the overall monitoring cost. Intuitively, the transitions between $\varnothing$ and l could be removed because the GPS position does not become relevant until it is the right time. The other eight transitions could be removed because y_{x} completely overwrites the effect of x_{x}, so it does not matter what happens with t and l during a meeting.

▷ **8.51. Example:** We show this with an example walk through Figure 8.9:

$$
\begin{array}{llllllll}
\langle \varnothing, & \varnothing, & p & \rangle & \xrightarrow{e,1} & \langle \{l\}, & \varnothing, & p & \rangle & \xrightarrow{e,2} \\
\langle \{l,m\}, & \varnothing, & p & \rangle & \xrightarrow{l,3} & \langle \{l,m\}, & \{m\}, & p_m & \rangle & \xrightarrow{e,4} \\
\langle \{t,l,m\}, & \{m\}, & p_m & \rangle & \xrightarrow{e,5} & \langle \{t,l\}, & \{m\}, & p_m & \rangle & \xrightarrow{l,6} \\
\langle \{t,l\}, & \varnothing, & p & \rangle & \xrightarrow{l,7} & \langle \{t,l\}, & \{t\}, & p & \rangle & \xrightarrow{l,8} \\
\langle \{t,l\}, & \{t,l\}, & p_{t,l} & \rangle & & & & & &
\end{array}
$$

This is the story:

1. We arrive at work before 9:00 for an early meeting. Nothing changes.
2. The meeting starts. We have an unstable configuration.
3. $p_m = [\![y_{\mathtt{x}}]\!](p)$. Our phone is automatically muted and the meeting minutes app is put on the screen.
4. It turns 9:00 during the meeting, but our phone does not have to respond. (This is the configuration marked in Figure 8.9.)
5. The meeting ends. We have another unstable configuration.
6. … updating $(p = [\![y'_{\mathtt{x}}]\!](p_m))$ …
7. … updating …
8. … done. $p_{t,l} = [\![x_{\mathtt{x}}]\!](p)$. The phone is set to volume 5.

Note how the GPS module was not required until transition 8. When we physically arrived at work (F_e), our phone (F_ℓ) was unaware of it. Still, our phone was always operating with a proper profile without requiring a large drain on the battery. ⌟

8.5 Conclusion

This chapter developed, step by step, an operational semantics for the reconfiguration of a running product in a dynamic product line. There has been some previous effort towards keeping objects in the heap up to date with the latest feature configuration. There has been a noticable lack of work, however, in coming up with strategies for keeping the running product itself up to date. By tracking both the local and environmental feature configurations, a set of light-weight deltas can apply just enough changes to the running product to bring it up to date without having to regenerate it from scratch. The number of necessary deltas has been reduced to a minimal level, and we have proof that the system maintains correctness of the product before and after every reconfiguration. The software deltoid defined for the main example of this thesis was intended for structural modification, not to reason about running programs. It has no syntax defined below the statement level, let alone a memory model. Instead, this chapter introduced a new deltoid based on profile management. The main case study is a mobile application for managing the settings of a smartphone based on user-defined rules and input from its sensors.

That being said, we can now get a clearer picture of what the concrete operational semantics may look like around the **`reconfigure`** inference rule of page 167. Recall, this statement was introduced by by Damiani et al. [63,

64] so that developers can indicate when it is safe to reconfigure the product. That means the entire process described in this chapter will have to occur while the control flow waits on that statement.

We'll describe a new operational semantics; a hybrid of the classical imperative program semantics and the semantics developed in this chapter, embodied by a new transition relation $\longrightarrow\!\!\!\!\!\rightarrow$. We'll also include reconfiguration of the heap, as explored by Damiani et al. They describe *reconfiguration translations*, modeled by an automaton much like our Mealy machines. We'll abstract from the details, and encapsulate their technique into a function $\mathrm{rh}\colon \Phi \times \Phi \times \mathcal{H} \to \mathcal{H}$, representing an effective procedure that performs those translations. It takes a local feature configuration, an environmental feature configuration and a heap, and returns a reconfigured heap. The hybrid configurations $\langle\, F_e, F_\ell, p, H, st, \sigma \,\rangle$ contain the environmental feature configuration F_e, the local feature configuration F_ℓ, the current product p, the current heap H, the next statement st and the current state σ.

The inference rules of the hybrid system would be as follows. First, environmental transitions can happen at any time during normal execution, but not while reconfiguration is taking place:

$$\frac{p \in \mathrm{V}(F_e) \qquad \langle\, F_e, F_\ell, p \,\rangle \xrightarrow{e} \langle\, F_e{}', F_\ell, p \,\rangle}{\langle\, F_e, F_\ell, p, H, st, \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle\, F_e{}', F_\ell, p, H, st, \sigma \,\rangle}$$

Until a **`reconfigure`** statement is encountered, the program just runs like it normally would, following the imperative semantics:

$$\frac{st \neq \texttt{reconfigure;}st'' \qquad \langle\, st, H, \sigma \,\rangle \longrightarrow \langle\, st', H', \sigma' \,\rangle}{\langle\, F_e, F_\ell, p, H, st, \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle\, F_e, F_\ell, p, H', st', \sigma' \,\rangle}$$

When control flow reaches a **`reconfigure`** statement, what happens next depends on whether the running product is out-of-date. If so, control is released to the dynamic product line semantics:

$$\frac{p \notin \mathrm{V}(F_e) \qquad \langle\, F_e, F_\ell, p \,\rangle \xrightarrow{\ell} \langle\, F_e, F_\ell{}', p' \,\rangle \qquad \mathrm{rh}(F_\ell, F_\ell{}', H) = H'}{\langle\, F_e, F_\ell, p, H, \texttt{reconfigure;}st, \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle\, F_e, F_\ell{}', p', H', \texttt{reconfigure;}st, \sigma \,\rangle}$$

Note that the heap is updated along with the product at every step.

If/when the product is up to date, a **`reconfigure`** statement can be discarded and control released to the (now modified) program:

$$\frac{p \in \mathrm{V}(F_e)}{\langle\, F_e, F_\ell, p, H, \texttt{reconfigure;}st, \sigma \,\rangle \longrightarrow\!\!\!\!\!\rightarrow \langle\, F_e, F_\ell, p, H', st, \sigma \,\rangle}$$

Further formalization —and implementation— of this idea would be a fascinating topic for future research.

8.6 Related Work

Hallsteinsen et al. [82] describe several properties that constitute a dynamic software product line. The approach presented in this chapter, and the papers on DDM [6, ¤1], allow 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 approach of this chapter, even though the environmental feature configuration can change during runtime, the set of available feature configurations is still fixed at build time.

Though ADM was designed from a software product line engineering perspective, the profile management case study of Section 8.2 is, of course, not a software product line, as it does not model the variability of software. It does, however, bear resemblance to a *Context-aware Program* [38, 103] or a *Self Adaptive System* [50, 146, 180]. Self-adaptive systems in particular have been linked with software product lines in recent literature. A number of papers aim to implement self-adaptive systems with dynamic software product line techniques [73, 170].

In self-adaptivity terms, DDM is *closed-adaptive*, as it is not able to cope with unexpected changes, in contrast to *open-adaptive* systems [146]. According to a recent survey by Weyns et al. [180], the vast majority of papers on these topics do focus specifically on development of flexible and reliable self-adaptive *software*. In comparison, the profile management model is relatively simplistic. As such, to call the profile management app a self-adaptive system would do the field (which has been concerned with self-driving cars and unmanned air vehicles behind enemy lines) a disservice. It is safe to say that DDM has not yet proved itself in those terms.

A number of recent publications, though, *have* explored dynamic software product lines in terms of delta modeling. Damiani et al. [63, 64] apply delta oriented programming to the problem, and focus on control flow, heap reconfiguration and type safety, as explained in the introduction to this chapter. Additionally, Muschevici et al. [141] recently extended the ABS language (Section 7.5) for the implementation of dynamic systems. They do mention the issue of dynamic product reconfiguration, and propose that certain deltas not already present in the original delta model (e.g., deltas x'_{x}, y'_{x} and y''_{x} in Figure 8.6) should be manually developed. One of the messages of Section 8.3.6 is that this may in fact be automated, given a correctly implemented delta derivation procedure, something Muschevici et al. have proposed as future work.

Interestingly, both groups mention the unanticipated runtime evolution necessary for open-adaptive systems. The idea is that adding, removing and modifying deltas while the product is still running is valid, so long as those changes have no impact on the deltas used in generating the currently running product. In particular, Muschevici et al. discuss MetaABS, an API based on reflection used for this very purpose.