Delta-oriented Monitor Specification

Eric Bodden[1], Kevin Falzon[1], Ka I Pun[2], and Volker Stolz[2,3]

[1] Secure Software Engineering Group, European Center for Security and Privacy by Design (EC SPRIDE), Technische Universität Darmstadt, Germany
[2] Dept. of Informatics, University of Oslo, Norway
[3] UNU-IIST, Macau S.A.R.

Abstract. Delta-oriented programming allows software developers to define software product lines as variations of a common code base, where variations are expressed as so-called program deltas. Monitor-oriented programming (MOP) provides a mechanism to execute functionality based on the execution history of the program; this is useful, e.g., for the purpose of runtime verification and for enforcing security policies.
In this work we discuss how delta-oriented programming and MOP can benefit from each other in the Abstract Behavior Specification Language (ABS) through a new approach we call Delta-oriented Monitor Specification (DMS). We use *deltas over monitor definitions* to concisely capture protocol changes induced by feature combinations, and propose a notation to denote these deltas. In addition, we explore the design space for expressing runtime monitors as program deltas in ABS.
A small case study shows that our approach successfully avoids code duplication in monitor specifications and that those specifications can evolve hand in hand with feature definitions.

Keywords: Runtime Verification, Monitor-oriented Programming, Interface Protocols, Software Product Lines

1 Introduction

Delta-oriented programming (DOP) allows software developers to define software product lines as variations of a common code base. Variations are expressed as *program deltas*, which can add, remove, and re-define units of code such as classes or methods [5]. Delta-oriented programming has been proposed as a way to structure software product lines (SPL) [6] and as a more structured alternative to other conditional-compilation constructs such as `#ifdef` [12].

The application interfaces (APIs) of software products frequently come with implicit or explicit usage contracts that describe how the individual methods of the API are to be called, e.g., in which order or with what parameters. Runtime monitoring is commonly used to verify the adherence to such usage contracts at runtime [3]. In runtime monitoring, the program under test is instrumented with (often stateful) runtime checks that signal an error if clients of the API violate the usage rules at runtime. In practice, the runtime monitoring machinery can be used for other purposes. More specifically, runtime monitors can be seen as a declarative programming paradigm in which code is executed based on events observed in the execution history of the program — a programming style coined Monitor-oriented Programming (MOP) [3].

Subjecting a product line's code to program deltas complicates its monitoring, as the introduction of the deltas may modify or extend usage contracts, or in the more general case of MOP, may expose new or altered execution histories. Thus, it follows that any runtime monitors present in the system may need to be updated as well.

In this work, we describe an initial design of Delta-oriented Monitor Specification (DMS), our approach to updating finite-state-machine based monitor specifications in line with delta definitions for regular program code. DMS allows programmers to deploy monitors as deltas, and to define deltas over monitor definitions comprising additions, removal or replacements of individual transitions, the introduction of new initial states and variable bindings or the additions, removal or replacement of transition guards.

We situate our approach in the context of the Abstract Behaviour Specification language (ABS), a modelling language for active objects [4] that has built-in support for DOP. Concretely, we provide an example based on ABS and propose a tool approach for integration with the ABS platform. We also report on the suitability of ABS for Monitor-oriented Programming.

To assess the viability of our approach, we apply Delta-oriented Monitor Specification to a small case study of a cashier system from the component-based development community. The Common Component Modelling Example (CoCoME) [15] is given as a use case with optional variabilities, which we treat as features in a software product line. Firstly, we use the example to introduce the ABS language, and then use the same language mechanism to instrument the example program, to monitor and enforce consistent API use through DMS. We give an implementation strategy that generates the necessary deltas which augment every method with monitoring code. As DMS are rather explicit since they describe changes with respect to a base automaton, we also introduce a more accessible, graphical high-level notation, from which one can automatically calculate the delta automaton.

To summarize, this paper contains the following original contributions:

- The idea of and a design for Delta-oriented Monitor Specifications, including a formalization and an implementation strategy.
- A discussion of the suitability of ABS for Monitor-oriented Programming.
- An assessment of the viability of the approach using a small case study.

The remainder of this paper is organized as follows: we present the salient features of the ABS language and its support for SPLs in the context of a running example in Sec. 2. A short motivation for runtime verification and protocols is provided in Sec. 3. We formalize the base automata with variable bindings and delta-automata to capture protocol changes in Sec. 4. Sec. 5 outlines how deltas can be used to enforce protocols as an optional feature in SPL products. Sec. 6

concludes with related work and a few suggested features to improve ABS's support for runtime verification.

2 Overview

ABS is very much in the style of traditional programming languages like Java or C++, but also models asynchronous behaviour, similar to Actors [9]. Every object can be understood as a process receiving and sending messages, with explicit release points in method bodies over boolean guards on the object state. On the static level, ABS uses subtyping through interfaces, but not code inheritance, making formal reasoning in ABS simpler than in other languages that support code inheritance. However, the language supports another important mechanism for reuse, since it directly includes a notion of software product lines (SPL), a feature language, and a low-level assembly mechanism for so-called "deltas".

In an SPL, features are mapped to sets of deltas, each of which may modify the program by removing/adding fields or methods, and overriding method-bodies with new code that can call back into the original code, allowing a construct similar to the `around()`-advice with `super`-calls of aspect-oriented programming. ABS is thus closer to an aspect-oriented programming language, although it lacks the flexibility of, e.g., wildcard matches on method invocation.

In previous work [2], we have used techniques from aspect-oriented programming [10] to instrument applications with runtime checks that enforce a particular *protocol* between objects. In runtime verification, one is generally interested in detecting patterns in the execution history, usually described by linear temporal logic formulas or regular expressions. Additionally, one may specify an action which must be taken when a monitor is triggered. When a monitor matches, the behaviour of the program is overridden with the behaviour annotated to the monitor. Enforcing protocols can be useful to add security aspects to an API, or guard against the misuse of an interface. Monitor specifications are often domain-specific, and can often be derived from the (informal) documentation.

In this paper, instead of using the full power of aspect-oriented programming techniques, we show that the more restricted subset of ABS programs is—in general, save some minor elements which have not yet been implemented in the prototype of the ABS language—sufficient to implement runtime verification.

Most importantly, we lift the notion of deltas to the level of monitoring. This allows us to customize protocols for features and products using a similar mechanism that customizes the code. Deltas are thus used *twice* in our approach: as part of the input, they define the products, and our approach contributes an additional delta which implements the monitoring *per product*.

Next, we give an overview of the ABS language and its support for SPLs.

ABS in the CoCoME case study We illustrate our approach with an example derived from the CoCoME case study [15]. It specifies a simple supermarket system on various levels (single cashdesk, single shop, enterprise) using components (the cashdesk, a store-component providing back-end services, a bank for credit card payments, etc.).

Based on an informal description of the principal use case, we focus here on the design of the cashdesk and how the cashier interacts with it. The scenario also conveniently specifies variabilities which we can express using features.

For each customer, the cashier initiates a new sale, and processes the purchases by scanning them with a barcode scanner. The backend provides necessary information such as price and description. All purchases are aggregated into a sale, and after indicating that the processing has finished, the system calculates the total. The cashier retrieves the money from the customer and enters the amount into the system. The system displays the amount of change to return. After receiving the change, the customer leaves, and the cashier starts over.

We obtain a self-explanatory program for the cashdesk with interface functions `startSale, enterItem, finishSale` and `pay`. In addition to the business logic in the form of methods, data types and (pure) functions over those data types are defined in the functional subset of ABS, e.g., key/value maps.

```
module CoCoME;
class Cashdesk(Store s) implements Cashdesk {
  Store store = s;
  Int total = -1;
  Bool finished = False;
  List<Item> items = Nil;

  Unit startSale() { total = 0; finished = False; items = Nil; }
  Unit enterItem(Int code, Int qty) {
    assert store != null;
    Item item = store.lookup(code);
    total = total + qty*price(item);
    items = Cons(item,items);
  }
  Unit finishSale() { finished = True; }
  Int pay(Int given) {
    assert given >= total;
    return given-total;
  }
}
```

Features of CoCoME On top of this base program, we define the following optional features: the system should permit credit-card payment as an alternative, and support an express-checkout lane for customers with only a few items. When a cash-desk is in express checkout mode, customers may only purchase a bounded number of products, and only cash payments are allowed.

Instead of changing the program to support those features directly using object-oriented design, we use ABS's software product lines to specify the different products. Delta *Credit* introduces a new method `Bool cardPay(CCData cc)`. Likewise, we ignore the details of refusing a customer should she try to buy too many items when the desk is in express mode—note that the number k of items is configurable by the feature through the delta. We also trigger an assertion when she attempts a credit-card payment while in this mode. The other requirement involving the interaction between both features is specified in delta *ExpressCC*. In express mode, no card payments are allowed:

```
delta Credit {                         delta ExpressCC {
 modifies class Cashdesk {               modifies class Cashdesk {
  adds Bool cardPay(CCData cc)            modifies Bool cardPay(CCData cc) {
    { return store.authorize(cc);}          // Not allowed in express mode
  adds Int cashPay(Int given)               assert ~mode;
    { return this.pay(given); }             return original(cc);
}}                                     }}}
```

It is evident that the sequence in which deltas are applied is relevant, such as when overwriting the `cardPay()` method following its introduction by a previous delta, or when accessing the `mode` attribute. Here, we need ABS's mechanism of explicitly ordering deltas for a particular feature. This is recorded through the `after`-clause in the product-line specification, which assembles the features shown below. We will later show that from our protocol deltas, we can derive a delta which is almost identical to this, since the functionality expressed in the requirement is *exactly* a protocol issue (enabledness of a method based on the execution history).

As a last ingredient, we need to define the valid products in this product line. We have the base product without any features, and both optional features, yielding four possible products in total.[4]

```
productline CoCoME
  features Express, Credit;           product Base();
  delta Credit when Credit;           product Credit(Credit);
  delta Express(10) when Express;     product Ex(Express);
  delta ExpressCC after Credit        product CCEx(Express,Credit);
          when Express && Credit;
```

3 Enforcing correct behaviour

The intended use of a programming API, such as our Cashdesk system, is usually not directly inferable from the code. This is problematic, and frequently leads to usage errors. It is therefore desirable to support programmers by documenting and checking usage restrictions.

In [2], we have formalized usage protocols to make their intended use explicit within the code, and to make it automatically checkable. The protocol is specified in a machine-readable notation as annotations in the Java code. *Method invocations*, including constructors, are specified via *atomic propositions* (or equivalently, as transition labels). Any violation of the contract, i.e., a method invocation that is not allowed by the protocol, will terminate the execution. While this is generally undesired for production code (there should not be any runtime errors), this approach is useful for defining testing oracles.

Extension to deltas. As the protocol is clearly application specific, if the application is the product of an SPL, there must be support for various protocols in different products. This gives us two possible options: specifying the full protocol *per product*, or *incrementally changing* the protocol, similarly to how deltas change code. We argue that the latter approach is preferable.

[4] ABS supports a product-selection language from, e.g., mutually-exclusive features, or dependencies, which is more than we can make use of in our example. See [6].

To correctly assemble a product from features, which map to sets of deltas, a designer needs intricate knowledge of the internal structure of the program. Features, and consequently their deltas, manipulate a potentially large base application. Clearly, a major focus on the protocol design will be on the base system. Modification of existing methods *may* make it necessary to update the protocol, and new methods *must* be incorporated (unless they require no special interaction protocol). Deletion of methods is straightforwardly handled by removing any occurrence of the method call in the protocol. Thus, we expect that specifying the changes in the protocol per feature is cheaper in terms of syntax and effort than re-specifying the complete protocol for each product.

Base protocol The intended API use of our component can be specified through a labelled transition system, where the labels are (guarded) method calls, as shown in Fig. 1. The intended usage, as indicated by the system use case, is that the cashier starts a sale for a new customer, records all items, indicates that all items have been processed, and handles the payment. Correspondingly, the state labels s, b, f are mnemonics for "starting", "buying", "finished". In Sec. 4, we will formalize the automaton construction.

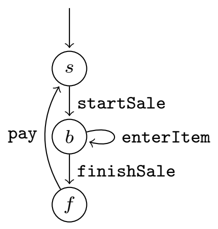

Fig. 1. Base protocol

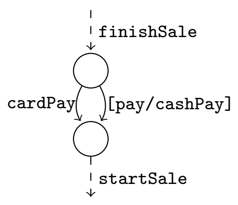

Fig. 2. Credit card payment

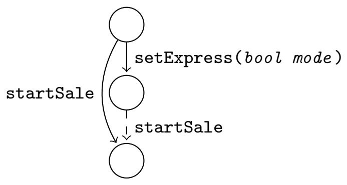

Fig. 3. Mode switch

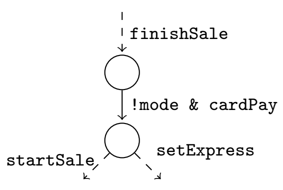

Fig. 4. ExCC

For the behaviour of the different products, we informally give the *relative* change in the protocol. Fig. 2 shows that after `finishSale()`, there are now two payment methods available. We have renamed the existing method from `pay` to `cashPay` for clarity, and added the `cardPay` method. The diagram shows wildcard states that the changed transitions attach to; the intention here is to add the new option as an alternative to the existing edge. Any existing edges that are not referred to in the protocol delta are left unchanged. The dashed transitions are used to determine which states in the original protocol to attach to. Since state names should only be used implicitly, one of our design goals is to *avoid referring to states*, matching, instead, on *existing transitions.* We elaborate on the necessary pointcut expressions in Sec. 4.

Fig. 3 introduces the mode-switch method, which can optionally be called *before* the `startSale` method. We make use of a *binding occurrence* with formal parameter `mode` that must match the signature of the operation. Note that the state s before the `startSale` invocation in the original protocol is an initial state. Therefore, the semantics of "before" should include relocation of the initial state.

Fig. 4 illustrates the interaction between the two available features of credit-card payment and express mode, where the previous mode switch pattern *binds* data (the current mode), and the new, additional part uses the data in the guard. Here, the intention is that the (existing) `cardPay` transition is only enabled when the `mode`-flag is not set to express mode. It is obvious that applying the second protocol constraint can only be valid in the presence of the former with the binding occurrence. This corresponds to the delta `ExpressCC` in our product line from the previous section.

4 Formalization

We model our protocols as finite automata with an extension to *bind* formal parameters of method calls to their instantiated values upon taking a transition. A transition in the automaton refers to variables used as placeholders in its binding function.

4.1 Defining Base Automata

Given that $\Theta := \texttt{VAR} \to \texttt{VAL}$ is a set of functions that resolve the name of a variable to its bound value, a *base automaton* $\mathcal{M}$ is a tuple $\langle Q, \Sigma \times \overrightarrow{\texttt{VAR}}, q_0, \theta_0, \Gamma\rangle$, with Q states, an alphabet Σ with a list of formal parameters, an initial state $q_0 \in Q$, an initial variable binding $\theta_0 \in \texttt{VAR} \to \texttt{VAL}$ and a set of transitions Γ, where:

$$\Gamma \subseteq \underbrace{Q}_{\substack{\text{current}\\\text{state}}} \times \underbrace{(\Sigma \times \overrightarrow{\texttt{VAR}})}_{\substack{\text{method}\\\text{signature}}} \times \underbrace{(\Theta \to \mathbb{B})}_{\text{guard}} \times \underbrace{((\Theta \times \overrightarrow{\texttt{VAL}}) \to \Theta)}_{\substack{\text{variable-binding}\\\text{transformation}}} \times \underbrace{Q}_{\substack{\text{next}\\\text{state}}}$$

Each transition relates a pair of states via a symbol with its parameters, a guard function and a state-binding function. The guard function is evaluated during traversal, with an outgoing transition only being chosen when its guard evaluates to true. The state-binding function will return a new binding function derived from the current bindings s and the input parameters $\vec{c}$. For the sake of brevity, one may forego specifying a guard or a state-binding function, in which case the functions are replaced by an always-true guard and an identity function, respectively. Thus, $(q, e, q') := (q, e, \lambda s.true, \lambda(s, \vec{c}).s, q')$. We also assume correct arity of formal parameters and binding functions.

Configurations A base automaton *configuration* is a pair consisting of a state and a variable binding. The initial configuration Φ_0 is thus defined as (q_0, θ_0).

Configurations Over Single Transitions An automaton $\mathcal{M}$ *accepts* an input $a := e(c_0, \ldots, c_n), e \in \Sigma, c_i \in \texttt{VAL}$ if, given its current configuration, there is an outgoing transition for the input symbol e whose guard evaluates to true. The evolution from a configuration Φ to Φ' within automaton $\mathcal{M}$ on receiving input a is denoted by $\Phi \xrightarrow{a}_{\mathcal{M}} \Phi'$ and is defined as follows:

$$\begin{aligned}(q, \theta) \xrightarrow{e(c_0,\ldots,c_n)}_{\mathcal{M}} (q', \theta') \;:=\; & (q, e(x_0, \ldots, x_n), \mathit{guard}, \mathit{binding}, q') \in \Gamma \\ & \wedge\ \mathit{guard}(\theta) \wedge\ \mathit{binding}(\theta, (c_0, \ldots, c_n)) = \theta'\end{aligned}$$

Trivially, $\Phi \xrightarrow{\epsilon}_{\mathcal{M}} \Phi' \;:=\; \Phi = \Phi'$. All states in the automaton are implicitly accepting, and the system is in a correct state as long as a next state is defined for the given input. Conversely, if the automaton cannot progress, then the input is invalid, signalling a *failure*. One can think of such an automaton as being implicitly total, with the complement of the defined transitions leading to a failure state. We define *single step rejection* from configuration Φ on input a:

$$a \notin \mathcal{L}_{\mathcal{M}}(\Phi) \;:=\; \neg(\exists q' \in Q,\ \theta' \in\ \texttt{VAR} \rightarrow \texttt{VAL} \cdot\ \Phi \xrightarrow{a}_{\mathcal{M}} (q', \theta'))$$

Accepting and Rejecting Runs The notion of accepted and rejected elements can be lifted onto sequences of inputs (or *runs*). Given a run as, with $a \in \Sigma \times \overrightarrow{\texttt{VAL}}$ and $s \in (\Sigma \times \overrightarrow{\texttt{VAL}})^*$, one can define the acceptance of a sequence of elements as:

$$\begin{aligned}(q, \theta) \overset{as}{\Rightarrow}_{\mathcal{M}} (q', \theta') \;:=\; & \exists q'' \in Q,\ \theta'' \in \texttt{VAR} \rightarrow \texttt{VAL} \cdot (q, \theta) \xrightarrow{a}_{\mathcal{M}} (q'', \theta'') \\ & \wedge\ (q'', \theta'') \overset{s}{\Rightarrow}_{\mathcal{M}} (q', \theta')\end{aligned}$$

Trivially, $\Phi \overset{\epsilon}{\Rightarrow}_{\mathcal{M}} \Phi' \;:=\; \Phi = \Phi'$.

A rejected sequence w starting from a configuration Φ is denoted as follows:

$$w \notin \mathcal{L}_{\mathcal{M}}(\Phi) \;:=\; \neg(\exists q' \in Q, \theta' \in \texttt{VAR} \rightarrow \texttt{VAL} \cdot \Phi \overset{w}{\Rightarrow}_{\mathcal{M}} (q', \theta'))$$

Thus, a run w is within the base automaton language if $\exists \Phi \cdot \Phi_0 \overset{w}{\Rightarrow}_{\mathcal{M}} \Phi$. Similarly, a run is not in the language (invalid) if $w \notin \mathcal{L}_{\mathcal{M}}(\Phi_0)$.

4.2 Well-formedness of Automata

Our use of variables in guards necessitates a notion of well-formedness that ensures that every variable occurring in a guard on a transition has been assigned a value *on all paths* leading to this transition.

Assuming a function $\mathit{vars} : \Gamma \rightarrow \overrightarrow{\texttt{VAR}}$ which yields the used variables in a guard, a transition $\langle S, a, g, b, T \rangle$ is *well-formed*, iff $\mathit{vars}(g) \subseteq \mathit{defs}_{\mathcal{M}}(S)$ where $\mathit{defs}_{\mathcal{M}}(S) : Q \rightarrow \overrightarrow{\texttt{VAR}}$:

$$\mathit{defs}_{\mathcal{M}}(S) \;:=\; \begin{cases} \mathit{dom}(\theta_0) & \text{iff } s = q_0; \\ \displaystyle\bigcap_{(S^p, e(x_0,\ldots,x_n), g, \theta, S) \in \Gamma} (\mathit{defs}_{\mathcal{M}}(S^p) \cup \{x_0, \ldots, x_n\}) & \text{otherwise} \end{cases}$$

where the S^p are the predecessors of the state S. An automaton is well-formed if all its transitions are well-formed.

4.3 Deltas

Deltas are structures that augment a base automaton by adding, modifying or removing transitions. It can also redefine the *initial state* and *variable bindings* of the base automaton.

Defining Deltas A *delta automaton* is a tuple $\langle Q^{\Delta}, \Sigma^{\Delta} \times \overrightarrow{\mathtt{VAR}}, q_0^{\Delta}, \theta_0^{\Delta}, \Gamma_+^{\Delta}, \Gamma_-^{\Delta} \rangle$, where Q^{Δ} is the set of (possibly new) introduced states, $\Sigma^{\Delta} \times \overrightarrow{\mathtt{VAR}}$ is a set of symbols, q_0^{Δ} is an optional redefined start state, θ_0^{Δ} is a binding function to be composed with any existing initial binding function, and Γ_+^{Δ} and Γ_-^{Δ} are the transitions to be added and removed, respectively. It is assumed that $\Gamma_+^{\Delta} \cap \Gamma_-^{\Delta} = \emptyset$.

Applying Deltas Given a base automaton $\mathcal{M} = \langle Q^{\mathcal{M}}, \Sigma^{\mathcal{M}} \times \overrightarrow{\mathtt{VAR}}, q_0^{\mathcal{M}}, \theta_0^{\mathcal{M}}, \Gamma^{\mathcal{M}} \rangle$ and a delta automaton $\Delta = \langle Q^{\Delta}, \Sigma^{\Delta} \times \overrightarrow{\mathtt{VAR}}, q_0^{\Delta}, \theta_0^{\Delta}, \Gamma_+^{\Delta}, \Gamma_-^{\Delta} \rangle$, the application of Δ to $\mathcal{M}$ yields a base automaton $\mathcal{M}' := \mathcal{M} \downarrow \Delta$, and is defined as follows:

$$\begin{aligned}
Q' &:= Q^{\mathcal{M}} \cup Q^{\Delta} \\
\Sigma' \times \overrightarrow{\mathtt{VAR}} &:= \Sigma^{\mathcal{M}} \times \overrightarrow{\mathtt{VAR}} \cup \Sigma^{\Delta} \times \overrightarrow{\mathtt{VAR}} \\
q_0' &:= q_0^{\mathcal{M}} \text{ if } q_0^{\Delta} = \bot, q_0^{\Delta} \text{ otherwise} \\
\theta_0' &:= \theta_0^{\mathcal{M}} \text{ if } \theta_0^{\Delta} = \bot, \\
&\qquad \lambda c.(\mathtt{case}\ \theta_0^{\Delta}(c) = \bot \Rightarrow \theta_0^{\mathcal{M}}(c); \mathtt{otherwise},\ \theta_0^{\Delta}(c)) \text{ otherwise} \\
\Gamma' &:= (\Gamma^{\mathcal{M}} \cup \Gamma_+^{\Delta}) - \Gamma_-^{\Delta}
\end{aligned}$$

where $\bot$ is an undefined element. Deltas can introduce or redefine bindings stated within the initial binding. In the case of redefinitions, the latest updated binding will be used. If the empty base automaton is $\mathcal{M}_\emptyset := \langle \emptyset, \emptyset, \bot, \lambda c.\bot, \emptyset \rangle$, one can redefine a base automaton $\mathcal{M}$ as a delta operation on $\mathcal{M}_\emptyset$. Formally, $\mathcal{M} := \mathcal{M}_\emptyset \downarrow \Delta_{\mathcal{M}}$, where $\Delta_{\mathcal{M}} := \langle Q^{\mathcal{M}}, \Sigma^{\mathcal{M}} \times \overrightarrow{\mathtt{VAR}}, q_0^{\mathcal{M}}, \theta_0^{\mathcal{M}}, \Gamma^{\mathcal{M}}, \emptyset \rangle$. Unreachable states after applying a delta automaton can be pruned implicitly as they can no longer influence the behaviour of the monitor.

Example 1. The delta automaton for the credit card payment (Fig. 2) is

$$\begin{aligned}
\Delta_{CC} := \langle\ &\emptyset, \{\mathtt{cashPay}, \mathtt{cardPay}\} && \text{no new state/new symbols} \\
&\bot, \bot, && \text{no initial state/no new initial binding} \\
&\{(f, \mathtt{cashPay}, s), (f, \mathtt{cardPay}, s)\}, && \text{transitions added} \\
&\{(f, \mathtt{pay}, s)\}\rangle && \text{transition removed}
\end{aligned}$$

As the transitions within the delta do not make use of guards or alter variable bindings, the shorthand transition notation is used.

Example 2. The delta automaton for switching Express mode (Fig. 3) is

$$
\begin{array}{lll}
\Delta_M := & \langle \{m\}, \{\texttt{setExpress}\}, m, & \text{new state/symbol/initial state} \\
& \lambda c.(\texttt{case}\ c = \text{“}\mathit{mode}\text{”} \Rightarrow \mathit{true}), & \text{new initial binding} \\
& \{(m, \texttt{setExpress}, \lambda s.\mathit{true}, \lambda(s, x). & \text{transitions added} \\
& \quad (\lambda y.(\texttt{case}\ y = \text{“}\mathit{mode}\text{”} \Rightarrow x_0; \texttt{otherwise},\ s(y))), s), & \\
& (m, \texttt{startSale}, b)\}, & \\
& \emptyset\rangle & \text{no transitions removed}
\end{array}
$$

The newly-added transition redefines the variable binding function, adding a binding for “mode”. Its value, x_0, is the first element of the list of values x passed on to the `setExpress` function.

Example 3. The delta automaton for Fig. 4 is

$$
\begin{array}{lll}
\Delta_{ExCC} := & \langle \emptyset, \emptyset, \perp, \perp, & \text{no new state/symbols/initial state/initial binding} \\
& \{(f, \texttt{cardPay}, \lambda s.\ !s(\text{“}\mathit{mode}\text{”}), \lambda(s, \vec{c}).s, m)\}, & \text{transition added} \\
& \{(f, \texttt{cardPay}, m)\}\rangle & \text{transition removed}
\end{array}
$$

The delta effectively modifies a transition in the original automaton, adding a guard on the value of “mode”.

Applying all three delta automata to our initial protocol, we obtain the resulting automaton $\mathcal{M}' := \mathcal{M} \downarrow \Delta_{CC} \downarrow \Delta_M \downarrow \Delta_{ExCC}$, as shown in Fig. 5.

4.4 Further design decisions

Conceptually, and based on the examples shown, it is clear that explicitly specifying the source- and target states for a transition does not scale very well: in general, a method may be used at various times, and accordingly occur in multiple places in the protocol (our example here is a degenerate case, as every method only occurs once). Ideally, graph-matching, as intended in Sec. 3, will take care of this. Matching the transitions in the base protocol and binding the wild-card states s_i allows us to calculate the set of transitions to add or remove.

The second important feature, that of binding of values during a run, requires a suitable representation of terms and a useful collection of function symbols over primitive types and their interpretation. In [2], we included functions to test object-identities, and allow invoking arbitrary methods over bound values in guards. We refer to the aforementioned paper for a detailed discussion.

The third and last important feature is *quantification*: in our running example, the protocol pertains to exactly one interface (or its implementing classes). A monitor is instantiated *per-object.* Conceivably, a protocol can cover coordinated interaction with several objects. Then, the aforementioned object-identities become a mandatory feature. Labels are then of, e.g., the form `o.m(x)`, and guards could use a more flexible form which allows reference to the variables *just* bound in the current call, e.g. `o` $\neq$ `p & o.m(x)`, denoting that the invoked object must not correspond to the previously bound p (which could come from either the callee-, or an argument position in a preceding transition).

Instantiation of such a truly crosscutting monitor would then occur on the initial matching transitions, and care must be taken when assigning meaning to

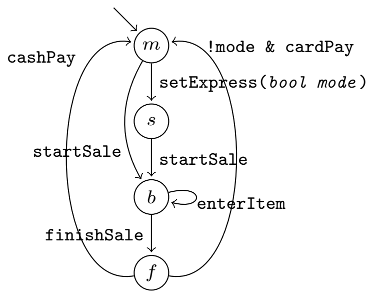

Fig. 5. The resulting automaton

```
delta ExpressMon {
  modifies class Cashdesk {
    adds Maybe<Bool> monMode = Just(True);
    modifies Unit setExpress(Bool m) {
      monMode = Just(m); // record mode
      // Only allowed between sales
      if (state == M) {
        original(m);
        state = S;
      } else {assert False; }
    }
}}
```

Listing 6. Binding of argument value

a fragment such as $\xrightarrow{\texttt{o.m()}} \cdot \xrightarrow{\texttt{p.n()}}$. The “hidden” reference in the second transition to the monitor instantiated by the first one requires *static* access to the monitor, which alas is currently not feasible in the ABS language (see our evaluation of suitability of ABS in Sec. 6).

5 Implementation

In the following, we outline how delta-protocols can be enforced for an interface by keeping track of progress through the state machine and generating assertions, which we naturally deploy using deltas. We will also comment on the use of annotations to make the protocol formally part of the model.

We have two different options for deploying monitors into an existing product line: either we first deploy the base monitor, and then the delta-protocols on top, or we first “flatten” the base protocol automaton and its deltas, and then generate code based on the resulting automaton. The former approach would require subsequent overwriting of previous enforcement code: we can see this clearly in the two different protocols that `cardPay` is involved in, depending on whether express mode is enabled or not. Although in principle the ABS language supports *targeted original calls*, which would aid the implementation, we would like to avoid redundant manipulation, and settle for the latter option.

For a monitor, we first need to introduce a datatype over all states, and a corresponding state variable per class which needs monitoring. Next, we collect all (reachable) transitions from the automaton that a method is involved in. We modify each method to assert that the transition is enabled, execute the original code, and update the state variable before returning from the method, similarly to `around`-advices in aspect-oriented programming (AOP). For ϵ-transitions, we collect the subsequent transitions. For the bindings, we need to introduce a state variable of the corresponding type, and read (write) the value in guards (binding) events. Listing 6 shows the generated code for binding the `mode`-switch.

As ABS lacks the means to apply deltas to classes *implementing a particular interface*, we also have to designate or compile the list of classes to be instrumented in a preprocessing step as well.

Storing the base protocol and the protocol deltas as part of the model is another problem. ABS has built-in support for annotations, which could be a suitable way of storing the protocol data as part of the file. Annotations attach values over user-defined datatypes to methods or statements. We can then define a datatype to specify the transitions of a protocol as annotations. These are then available during compilation when using the ABS toolchain.

A prototypical implementation of monitoring for the ABS compiler frontend is available from `http://www.mn.uio.no/ifi/english/research/projects/rvabs/`.

6 Related Work and Conclusion

We contrast our work with other works from the areas of aspect-oriented programming, model-driven development, monitor-oriented programming, runtime-verification for software product lines and typestate checking.

Aspect-oriented programming. Both AOP and DOP have in common that they use programming-language elements that allow programmers to insert code into some existing "base code" systems. However, both approaches fundamentally differ in their intent and methodology. The goal of AOP is to modularize concerns that are inherently crosscutting. Most AOP languages therefore support quantification constructs that allow programmers, for instance, to insert code before all method calls or after all field accesses. In addition, most AOP languages have a purely dynamic semantics. While AOP tools typically modify code through static weaving, their semantics are defined through dynamic entities, e.g., the interception of runtime events. Aspects are often intended to be re-used among several software systems. DOP, on the other hand, aims to allow structured compile-time variations of a given piece of software. This is a purely static view; after compilation, deltas are "flattened away", there is no notion of intercepted runtime events. There is also no quantification: in DOP, programmers need to explicitly specify the code elements that need to be modified, and there is no way to specify a whole range of such elements in a declarative style. This lack of quantification makes it less convenient to implement highly crosscutting features such as runtime monitoring. On the other hand, DOP makes it simpler to define delta-oriented monitor specifications, because the code-level effects of applying a delta are immediately obvious. The monitors can thus be defined in terms of the unmodified and modified interface. In AOP, such definitions would be more complex, as the weaving process in AOP is typically hidden from the user, and thus the modified interface is not as easy to deduce.

Model-driven development. The lifting of aspect-oriented techniques to UML models has been done for activity diagrams in [13]. As activity diagrams are syntactically richer than state machines, correspondingly we expect a concrete aspect to be equally verbose. The article does not give a detailed example, but this is confirmed in earlier work, where matching is clearly not based on the diagrammatic representation [14]. In the same paper, the authors also indicate their own and other existing approaches to weave state machines. The manipulations are purely structural, independent of state machine semantics, whereas in contrast, we have well-formedness requirements on the resulting model due to the specific nature of our automata. Similar checks could of course be employed on their resulting models as well.

In the field of SPLs, the Common Variability Language CVL [7] uses an approach to match fragments in the base model which could be useful to implement user-friendly matching on the graphical notation and calculate the delta automata. CVL uses matching on boundary elements (which would be states or transitions in our setting) to define anchor points for substitutions; these anchors are defined in terms of concrete elements of the base model, which indicates that only exactly one substitution can be carried out (a suitable matching mechanism for our purposes should find *all* instances of a pattern in the base automaton). Again, defining delta automata through substitutions will result in a very verbose notation, whereas we envision a more convenient, dedicated notation for adding and removing edges in fragment automata.

Monitor-oriented programming. MOP, prominently advocated by Chen and Roşu [3], is a programming model in which program features can be implemented in a declarative style, as responses to sequences of events in the program's execution history. One natural application of MOP is runtime verification, in which one uses MOP to define testing oracles, notifying the user of a failed test run after having observed a property-violating sequence of program events. However, there are other uses of MOP. For example, one can envision using MOP to implement an auto-save feature that saves a file after every 1000 key strokes. Our delta-oriented monitor specifications allow the delta-oriented adaptation of monitors for the general case of MOP.

Runtime verification for software product lines. Our work on delta-oriented monitor specifications allows monitor specifications to evolve together with delta-oriented code. As explained above, this can be particularly useful in the area of runtime verification. However, there are other ways to combine runtime verification with software product lines. Kim et al. exploit the constraints imposed by a feature model, paired with a static program analysis to restrict runtime verifications only to products that actually have the potential of violating the property in question [11]. This approach could be extended for delta-oriented monitor specifications, and we consider such an extension for future work.

Typestate checking. The stateful patterns that runtime monitors match against can also be checked statically through a mechanism called typestate checking, if appropriate annotations are present in the code. Plaid, for example, is a programming language for implementing software in a typestate-checkable way [1]. In Plaid, programmers annotate methods with the effects that they have on the internal state of a (virtual) state-based monitor. A static type-checker then verifies whether the usage of those methods complies with the given finite-state patterns. The annotations necessary for Plaid bear some similarities to the annotations that we propose in this paper, but in Plaid are much more verbose. In particular, the programmer must add non-trivial aliasing annotations.

General Runtime Verification. In earlier work [2], we have used more general finite alternating automata with variable bindings to support verification of linear-time logics (LTL) properties at runtime. So-called *tracecheck* are defined per Java-interface, and a program is instrumented using AOP. We did not envision variable protocols, and—given their difficult readability—are of the opinion that LTL-specifications are unsuitable candidates to relative modifications.

An interface behaviour specification language for the actor-language Creol was proposed in [8]. It is a regular language over constructor- and method invocations with variable bindings, which *only* supports matching of bound object-identities. In combination with a model checker, a Creol object can then be checked against an interface specification through synchronous parallel composition, with the usual limitations on state space explosion when model checking OO systems. It does not address the runtime of a system, and does not support guards, although this could probably be added.

Conclusion We have presented a definition and implementation strategy for DMS. Interface protocols, such as [2], for the different products in an SPL can be specified as relative changes to the protocol of the base product, just as relative changes describe a software product in the ABS language. Instrumentation of methods to enforce protocols is done through deltas, as well. Protocol deltas can be generated based on our notion of flattening a base- and delta automaton.

As future work, we will follow up on using annotations to store protocol deltas and develop a preprocessor for the ABS toolchain. Also, we would like to formalize calculation of the delta automata from the (graphical) specification of relative changes as indicated in Sec. 3. This can most likely be discharged by referring to existing graph-matching approaches. Naturally, we are also interested in applying our approach to a non-trivial example.

Currently, one of the limitations of the ABS language is that it neither has constructors, nor can a class-initializer be modified by a delta. This makes it difficult to inject, e.g., a factory for monitor-instances, where many objects communicate with a single monitor. As an immediate workaround, all call-sites of object instantiations would need to be instrumented, which in general cannot be done with simple advice and an `original`-call, but would require code-duplication.

In this paper, we have not made use of ABS as an actor language. Its release points, where execution is suspended until a boolean condition on the object state holds, could be used to alternatively model the protocol: in an actor-based, or even distributed system, in our opinion it would feel much more natural to ignore a "babbling" participant which sends messages out of turn, instead of terminating execution (since, e.g., the assertions which we have used would terminate the *callee*, not the actually misbehaving caller). We could envision `await` statements on the state variable tracking progress through the protocol. Also, the implicit identity of the caller could be incorporated into protocols (avoiding its explicit occurrence in an argument position). In addition, an actor-based setting would encourage the study of the use of protocols in an asynchronous environment.

Acknowledgements This work was supported by the German Federal Ministry of Education and Research (BMBF) within EC SPRIDE, by the Hessian LOEWE excellence initiative within CASED, the project *Runtime Verification for ABS Product Lines* funded by DAAD and Forskningsrådet, and by the project ARV funded by Macau Science and Technology Development Fund. We thank Franziska Kühn from U. Lübeck for her work on prototyping the implementation.

References

1. J. Aldrich, J. Sunshine, D. Saini, and Z. Sparks. Typestate-oriented programming. In *Proc. 24th ACM SIGPLAN conf. companion on Object oriented programming systems languages and applications*, OOPSLA '09, pages 1015–1022. ACM, 2009.

2. E. Bodden and V. Stolz. Tracechecks: Defining semantic interfaces with temporal logic. In W. Löwe and M. Südholt, editors, *Software Composition*, volume 4089 of *Lecture Notes in Computer Science*, pages 147–162. Springer, 2006.
3. F. Chen and G. Roşu. MOP: an efficient and generic runtime verification framework. In *OOPSLA '07*, pages 569–588. ACM, 2007.
4. D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, I. Schaefer, J. Schäfer, R. Schlatte, and P. Y. H. Wong. Modeling spatial and temporal variability with the HATS abstract behavioral modeling language. In M. Bernardo and V. Issarny, editors, *11th Intl. School on Formal Methods for Eternal Networked Software Systems (SFM)*, volume 6659 of *Lecture Notes in Computer Science*, pages 417–457. Springer, 2011.
5. D. Clarke, M. Helvensteijn, and I. Schaefer. Abstract delta modeling. In E. Visser and J. Järvi, editors, *GPCE*, pages 13–22. ACM, 2010.
6. D. Clarke, R. Muschevici, J. Proença, I. Schaefer, and R. Schlatte. Variability modelling in the ABS language. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *Intl. Symp. Formal Methods for Components and Objects (FMCO)*, volume 6957 of *Lecture Notes in Computer Science*, pages 204–224. Springer, 2010.
7. F. Fleurey, Ø. Haugen, B. Møller-Pedersen, G. K. Olsen, A. Svendsen, and X. Zhang. A generic language and tool for variability modeling. Technical Report A13505, SINTEF, Oslo, Norway, 2009.
8. I. Grabe, M. Kyas, M. Steffen, and A. B. Torjusen. Executable interface specifications for testing asynchronous Creol components. In F. Arbab and M. Sirjani, editors, *Fundamentals of Software Engineering (FSEN). Revised Selected Papers*, volume 5961 of *Lecture Notes in Computer Science*, pages 324–339. Springer, 2009.
9. P. Haller and M. Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2–3):202–220, 2009.
10. G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.
11. C. H. P. Kim, E. Bodden, D. Batory, and S. Khurshid. Reducing configurations to monitor in a software product line. In *1st Intl. Conf. on Runtime Verification*, volume 6418 of *Lecture Notes in Computer Science*, pages 285–299. Springer, 2010.
12. J. Liebig, S. Apel, C. Lengauer, C. Kästner, and M. Schulze. An analysis of the variability in forty preprocessor-based software product lines. In *ICSE 2010 (1)*, pages 105–114. IEEE, 2010.
13. D. Mouheb, D. Alhadidi, M. Nouh, M. Debbabi, L. Wang, and M. Pourzandi. Aspect weaving in UML activity diagrams: A semantic and algorithmic framework. In L. S. Barbosa and M. Lumpe, editors, *7th Intl. Workshop on Formal Aspects of Component Software (FACS). Revised Selected Papers*, volume 6921 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 2010.
14. D. Mouheb, C. Talhi, M. Nouh, V. Lima, M. Debbabi, L. Wang, and M. Pourzandi. Aspect-oriented modeling for representing and integrating security concerns in UML. In R. Y. Lee, O. Ormandjieva, A. Abran, and C. Constantinides, editors, *8th Intl. Conf. on Software Engineering Research, Management and Applications*, volume 296 of *Studies in Computational Intelligence*, pages 197–213. Springer, 2010.
15. A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors. *The Common Component Modeling Example*, volume 5153 of *Lecture Notes in Computer Science*. Springer, 2008.