Delta Modeling and Model Checking of Product Families

Hamideh Sabouri, Ramtin Khosravi

Delta Modeling and Model Checking of Product Families *

Hamideh Sabouri[1] and Ramtin Khosravi[1,2]

[1] School of Computer science and Electrical Engineering, College of Engineering, University of Tehran, Tehran, Iran
[2] School of Computer Science, Institute for Research in Fundamental Sciences (IPM),Tehran, Iran

Abstract. Software product line engineering focuses on proactive reuse to reduce the cost of developing families of related systems. A recently proposed method to develop software product lines is delta modeling where a set of deltas specify modifications that should be applied to a core product to achieve other products. The main advantage of this technique is its modularity and flexibility. In this paper, we propose an approach to model check delta-oriented product lines. To this end, we transform a delta model to a corresponding annotated model where an application condition is associated to each statement. An application condition specifies the set of products that a statement is included in them. We present the semantics of the resulting model in form of a featured transition system where each transition is annotated with an application condition. Featured transition systems are supported by a variability-aware model checking technique that can be used to verify the annotated model.

1 Introduction

Software product line (SPL) engineering enables proactive reuse by developing a family of related products instead of developing individual products separately. To this end, the commonalities and differences between products should be modeled explicitly [1]. Feature models are widely used to model the variability in SPLs. A feature model is a tree of features containing mandatory and optional features as well as other constraints among them, e.g., mutual exclusion. A product is defined as a valid combination of features, and a family is the set of all possible products [2] (section 3).

Delta modeling [3] is a modular, flexible, and expressive modeling approach that is recently proposed to develop SPLs. In this approach, an SPL is represented by a core product and a set of deltas. Deltas represent modifications that must be applied to the core product to derive other products of the product line. Each delta has an application condition that specifies the feature configurations on which the modifications are applicable.

* This research was in part supported by a grant form IPM. (No.CS1390-4-02).

As SPL engineering is increasingly used in the development of mission-critical and safety-critical systems such as embedded systems [4], formal verification of software product lines is essential. Recently, a number of approaches has been proposed to deductively verify delta-oriented SPLs using theorem proving [5–8]. However, to the best of our knowledge, there is no approach applying model checking technique [9] to verify delta-oriented models of software product lines. On the other hand, in [10] model checking algorithm is adapted to be applicable to SPLs (which we refer to it as variability-aware model checking). The drawback of this approach is that it does not support modular modeling of SPLs as the underlying formal model of product family is annotative. In annotative modeling approaches, each transition/statement of the model is annotated by an application condition to indicate the feature combinations that enable the transition/statement. The model checker uses the annotations to determine the set of products that satisfy/violate a property.

In this paper, we propose an approach to model check delta-oriented models of product families. Due to the compositional nature of delta models, developing a technique to analyze the entire family is not reasonable. A simple strategy is to generate all products of an SPL and model check each individually. However, this strategy may involve redundant computations due to similarities among the products. An alternative is to take benefit from the variability-aware model checking techniques. To this end, we transform a delta model to a corresponding annotative model. The annotated model can then be verified using variability-aware model checking technique. We select Rebeca [11] for formal modeling which is an actor-based language with a formal foundation to model and verify concurrent and distributed systems (section 4.1). Recently, the ABS language [12] is developed to model the behavior of configurable and distributed systems using delta modeling. However it is not supported by a model checking tool yet. Therefore, we choose Rebeca which is supported by an accessible model checker Afra [13]. Due to modularity, object-based nature, and Java-like syntax of Rebeca, its adaptation to support delta modeling is straightforward (considering the work on delta-oriented modeling of object-oriented SPLs like Java [3]).

To provide an approach to model check delta-oriented product families, we extend Rebeca to support delta modeling (section 4.2). We also introduce annotations in Rebeca and define the semantics of annotated Rebeca models using featured transition systems (FTS) [10] to which the variability-aware model checking algorithm is applicable (section 4.3). In FTS, an application condition determines the feature combinations that enable a transition. Then, we propose a method to transform a delta model to a corresponding annotated model (section 5.1). We also justify the correctness of our proposed approach intuitively (section 5.2). There are two possible approaches to model check the resulting annotated Rebeca model: using its underlying FTS or generating a plain Rebeca model from it to use the existing model checker of Rebeca for verification (section 5.3).

The main contribution of our work is developing SPLs in a high-level and modular manner by employing delta-oriented modeling concept while taking

advantage of variability-aware model checking which is currently only applicable to annotative models of product families. The contributions of our paper can be summarized as follows:

- We extend Rebeca to support delta modeling which enables us to model families of actor systems.
- We extend Rebeca with annotations along with its semantics based on FTS to apply the existing variability-aware model checking techniques.
- We propose a method to transform delta models to annotated models which enables us to use existing techniques for annotated models of SPLs.

The Coffee Machine Example. We use Coffee Machine family as the running example in the paper. A coffee machine may serve coffee or tea or both. Adding extra milk and extra sugar may be supported optionally. The payment method of a coffee machine is either by coin or by card. □

2 Related Work

Several approaches has been proposed for formal modeling of SPLs using SMV [14], automata and transition based systems [15, 10, 16, 17], process algebra [18], Petri nets [19], and Promela [20]. These approaches capture the behavior of the entire product family in a single model by including the variability information in it using annotations. Annotating a transition/statement with an application condition indicates the configurations that enable the transition/statement. In [18], an operator is added to CCS to specify alternative processes.

To model check annotated models, model checking technique has been adapted in [10] to verify featured transition systems. In FTS, an application condition determines the feature combinations that enable a transition. To explore the state space of an FTS, the track of products should be kept. Thus, a reachability relation is constructed while exploring the state space which is a set of pairs (s, px). Such couple indicates that state s is reachable by products in px.

Recently, a number of methods has been proposed to deductively verify delta-oriented SPLs. In [5], all derivable products of a delta-oriented SPL are verified incrementally using interactive theorem proving. In the first step, the core product is verified completely. Then, for each other product, the invalidated proofs and some new obligation proofs are proven. In [6], a family-based technique is proposed to reduce the cost of deductive verification for SPLs. For modular verification of software families, a Liskov principle is developed for delta-oriented programming in [7]. In [8], a transformational proof system is developed for delta-oriented programs which supports modular verification.

3 Background: Software Product Lines

Software product line engineering is a paradigm to develop software applications using platforms and mass customization. To this end, the commonalities and

differences between products should be modeled explicitly. Feature models [2] are widely used for this purpose. A feature model represents all possible products of a software product line in terms of features and relationships among them. A feature model is a tree of features that allows the *mandatory*, *optional*, *or*, and *xor* relationships among features. It also includes *requires* and *excludes* constraints between features. A product is derived from a feature set by making a decision to include/exclude each feature. A valid product conforms to the constraints that are specified in the feature model.

A configuration keeps track of including/excluding features. The root feature can be omitted in a configuration as it is included in all products. Having feature set $\mathcal{F}$ with n features, a configuration is defined as $c \in \{true, false, ?\}^n$ where $c_i = true/false$ represents inclusion/exclusion of the i^{th} feature. The value '?' indicates that a feature is not included nor excluded yet. A configuration is *decided* if it does not contain any '?' values. In other words, a decision is made about inclusion/exclusion of all features. Otherwise, the configuration is *partial*.

Sets of products can be described using application conditions. An *application condition* φ is a propositional logic formula over a set of features $\mathcal{F}$, defined by $\varphi ::= true \mid f \mid \varphi_1 \wedge \varphi_2 \mid \neg\varphi$ where $f \in \mathcal{F}$.

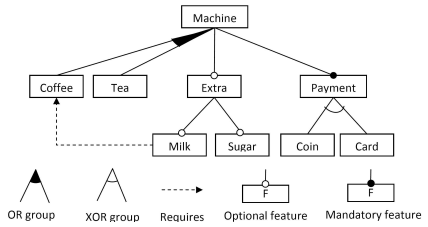

Fig. 1. The feature model of the coffee machine example

The Coffee Machine Example: Feature Model. The corresponding feature model of the coffee machine is depicted in Figure 1. Coffee and tea features have an *or* relationship implying that the machine serves one of these drinks at least. Adding extra milk and extra sugar is optional. However, when adding extra milk feature is available, the machine should be able to serve coffee because of the *requires* constraint between the milk and coffee features. Finally, the coin and card features have an *xor* relationship meaning that each product supports one and only one of them. A configuration that includes milk and excludes coffee is not a valid configuration. An example of a valid configuration is one that includes coffee, payment, and coin features and excludes the rest. □

4 Modeling Product Families in Rebeca

In this section, we describe two approaches to model product families in Rebeca: delta-oriented approach and annotative approach.

```
reactiveclass Controller() {
  knownrebecs {
    CoffeeMaker cm;
  }

  statevars {
    int cash, change, cost;
  }

  msgsrv initial() {
    cash, change = 0;
    cost = 1;
    self.receivePayment();
  }

  msgsrv nextOrder() {
    cm.makeCoffee(cash);
  }

  msgsrv receivePayment() {
    cash = ?(1,2,3);
    self.nextOrder();
  }

  msgsrv returnChange(int c) {
    change = cash - c;
    cash = 0;
    self.receivePayment();
  }
}
```

```
reactiveclass CoffeeMaker() {
  knownrebecs {
    Controller ctrl;
  }

  statevars {
    boolean addingCoffee;
    int cost;
  }

  msgsrv initial() {
    addingCoffee = false;
    cost = 1;
  }

  msgsrv makeCoffee(int cash) {
    addingCoffee = true;
    self.complete();
  }

  msgsrv complete() {
    ctrl.returnChange(cost);
    addingCoffee = false;
    cost = 1;
  }
}
```

Fig. 2. Rebeca model of a coffee machine

4.1 Rebeca

Rebeca is an actor-based language for modeling concurrent and distributed systems as a set of reactive objects which communicate via asynchronous message passing. A Rebeca model consists of a set of *reactive classes*. Each reactive class contains a set of *state variables* and a set of *message servers*. Message servers execute atomically, and process the receiving messages. The *initial* message server is used for initialization of state variables. A Rebeca model has a *main* part, where a fixed number of objects are instantiated from the reactive classes and execute concurrently. We refer to these objects as *rebecs*. The rebecs have no shared variable. Each rebec has a single thread of execution that is triggered by reading messages from an unbounded message queue. When a message is taken from the queue, its corresponding message server is invoked.

The Coffee Machine Example: Rebeca Model. Figure 2 shows the Rebeca model of a product from the coffee machine family that only includes coffee, payment, and coin features. In this model, the controller manages the payment as well as incoming orders and sends message to coffee maker accordingly. In the first step, the payment is received which can be at most three coins. Then, the drink may be ordered (the only option is coffee in this product). The controller is informed of the completion of serving the requested drink as well as the final

cost so it can return the change and prepare for another order. Note that the cost of a drink may increase by adding extra milk or sugar in those products that support the corresponding features. □

4.2 Delta-oriented Modeling in Rebeca

In delta modeling, an SPL is represented by a core product and a set of deltas. The core product is a valid product and deltas represent the modifications that should be applied to the core product to derive other products. In [21], the delta modeling technique is applied to object-oriented implementations of software product lines. Due to object-based nature of Rebeca language, we take a similar approach to [21] to introduce delta modeling in Rebeca.

Core Product A core product is a Rebeca model that captures the behavior of a product for a valid and decided feature configuration. Therefore, it contains a set of reactive classes and the main part where rebecs are instantiated.

The Coffee Machine Example: Core Product. We consider the product depicted in Figure 2 as the core product. As we mentioned earlier, it includes coffee, payment, and coin features and excludes other features. □

Deltas A delta represents the modification that must be applied to the core product to derive another product from the family. Deltas may add, remove, or modify reactive classes. Modifying a reactive class may add or remove known rebecs, state variables, and message servers. It may also change the behavior of an existing message server. Furthermore, an application condition is associated with each delta to specify the configurations for which the delta is applicable to the core product:

delta ⟨`name`⟩ [**after** ⟨`delta names`⟩] **when** ⟨`application condition`⟩{
  removes ⟨`reactive class name`⟩
  adds ⟨`reactive class definition`⟩
  modifies ⟨`reactive class name`⟩ {
    removes ⟨`state variable/known rebec/message server name`⟩
    adds ⟨`state variable/known rebec/message server definition`⟩
    modifies ⟨`message server name`⟩ ⟨`message server definition`⟩
  }
}

In the above description of a delta, the `when` clause represents the application condition and the `after` clause is used to specify the order of applying deltas.

The Coffee Machine Example: Deltas. A number of deltas that can be defined to describe a family of coffee machines are:

```
delta δ1 after δ3, δ4 when ¬coffee{
   removes CoffeeMaker
   modifies Controller {
      modifies nextOrder {I1}
   }
}
```

```
delta δ2 when tea {
   adds reactiveclass TeaMaker {...}
   modifies Controller {
      adds TeaMaker tm
      modifies nextOrder {I2}
   }
}
```

```
delta δ3 when milk {
   modifies CoffeeMaker {
      modifies makeCoffee {I3}
   }
}
```

```
delta δ4 when sugar {
   modifies CoffeeMaker {
      modifies makeCoffee {I4}
   }
}
```

Delta δ_1 defines the products without the coffee feature. Delta δ_2 adds the facility to serve tea. Deltas δ_3 and δ_4 are applied when adding extra milk and sugar are supported. □

Resolving Conflicts Conflicts among deltas may happen when they manipulate the same program entity in different ways. This leads to different implementations for one product when deltas are applied in different order. To achieve a unique product for a certain configuration, an order must be defined to apply deltas. A conflict-resolving delta δ_{ij} should be introduced to avoid conflict between two unordered deltas δ_i and δ_j. The application condition of δ_{ij} is the conjunction of the application conditions of δ_i and δ_j and δ_{ij} is applied later than both conflicting deltas (δ_{ij} has a higher priority than δ_i and δ_j).

The Coffee Machine Example: Conflicts. Deltas δ_1 and δ_2 are unordered and they are both modifying the implementation of the *nextOrder* message server. Another conflict exists between δ_3 and δ_4 which both modify the *makeCoffee* message server. Assume that in I_3 we add milk to coffee and increase the cost of the drink. In I_4 sugar is added and the cost is increased. When both features are available, the ultimate behavior of *makeCoffee* is determined by the delta that is applied later. To handle these issues, two conflict resolving deltas should be defined as δ_{12} and δ_{34}:

```
delta δ34 after δ3, δ4
when milk ∧ sugar {
   modifies CoffeeMaker {
      modifies makeCoffee {I5}
   }
}
```

```
delta δ12 after δ1, δ2
when ¬coffee ∧ tea {
   modifies Controller {
      modifies nextOrder {I6}
   }
}
```

In I_5 we add sugar and milk to coffee and determine the cost accordingly. □

Delta Model We define a delta model as a set of deltas along with their application conditions and priorities. A delta model is a triple $(\Delta, \Gamma, \prec)$ where

- Δ is a finite set of deltas,
- $\Gamma : \Delta \rightarrow \Phi_{\mathcal{F}}$ is function that associates an application condition with each delta,
- $\prec \subseteq \Delta \times \Delta$ is a partial order on Δ. $\delta_i \prec \delta_j$ states that δ_i should be applied before (not necessarily directly before) δ_j, when both deltas are applicable.

In the above definition, $\Phi_{\mathcal{F}}$ is the set of all possible application conditions over feature set $\mathcal{F}$. In our approach, we assume that we have an unambiguous delta model where all the conflicts among unordered deltas are resolved by defining appropriate conflict resolving deltas.

4.3 Annotated Rebeca Models

Syntax A fine-grained approach to represent variability in Rebeca model is to annotate the model with application conditions. We denote an annotation by $@\varphi$ where φ represents an application condition. In a Rebeca model, we may annotate reactive classes, known rebecs, state variables, message servers, and statements (collectively referred to as *model entities*). Annotating an entity with an application condition specifies the set of products that include the entity. However, annotations on statements are enough to model SPLs. We can express annotations on other types of entities just by using annotations on statements, as shown below.

- A reactive class R annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement sending a message to a rebec r where r is an instance of R.
- A known rebec r annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement that sends message to r.
- A state variable v annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement that assigns to v or uses the value of v.
- A message server m annotated by $@\varphi$ is modeled by associating $@\varphi$ to every statement that sends the message m.

Hence, in the rest of this paper, we assume that only statements of message servers are annotated with application conditions when presenting the semantics of annotated Rebeca models.

The Coffee Machine Example: Annotations. We can annotate the Coffee maker reactive class as `@coffee reactiveclass CoffeeMaker`, to indicate that the reactive class is only available in the products supporting the coffee feature. An alternative way is to annotate the statement `cm.makeCoffee(cash)` with application condition `@coffee`. As a result, no message is sent to the coffee maker, thus it would be excluded from the model implicitly. □

Semantics The semantics of annotated Rebeca models can be described using a featured transition system (FTS). In [22], Rebeca semantics is described in form of labeled transition systems (LTS). In this work, we extend such LTS to an FTS to capture the notion of variability within an annotated Rebeca model. An FTS [10] is a transition system where the transitions are annotated using application conditions. Assuming that $\Phi_{\mathcal{F}}$ is the set of all possible application conditions over feature set $\mathcal{F}$, we define the semantics of an annotated Rebeca model as a featured transition system (S, I, A, T, γ) where

- S is a set of global states
- I is the initial state
- A is a set of actions (message servers)

– $T \subseteq S \times A \times S$ is a set of transitions
– $\gamma : T \rightarrow \Phi_{\mathcal{F}}$ associates an application condition to each transition

Each rebec has a local state composed of the values of its variables and the state of its queue: $\varsigma = \langle \mathcal{V}, q \rangle$. A Rebeca model consists of a number of rebecs executing concurrently. Thus, the global state is defined as the combination of the local states of all rebecs: $s = \prod \varsigma_i$. In the initial state, all of the queues only contain the *initial* message and all of the state variables have their default values. Message servers in Rebeca are executed in one atomic step, therefore an action corresponds to the execution of a message server. Variability in the behavior of a model is realized through annotating the statements of message servers. The concept of variability is reflected in the semantics of Rebeca as follows.

We define message server m as a sequence of statements $\langle st_1; ...; st_n \rangle$ where φ_i is the application condition of st_i. To describe the semantics of variability in message servers easier, we consider that a sub-transition labeled by a sub-action from a sub-state to another, represents the execution of a single statement. In state s, execution of statement st_i has two possible outcomes. If φ_i does not hold, st_i is skipped without changing the local state of any rebecs. Otherwise when φ_i holds, execution of st_i may affect the local state of the currently executing rebec by changing the value of its state variables or putting a message in its queue (when the rebec sends a message to itself). Moreover, it may change the local state of other rebecs by putting messages in their queue. A possible path that denotes execution of m is: $s \xrightarrow{st_1,\varphi_1} \alpha_1 \xrightarrow{st_2,\neg\varphi_2} \alpha'_2 \xrightarrow{st_3,\varphi_3} ... \xrightarrow{st_n,\varphi_n} s'$.

Note that from each sub-state α_{i-1}, two sub-transitions with application conditions φ_i and $\neg\varphi_i$ are possible: $\alpha_i \xrightarrow{st_i,\varphi_i} \alpha_{i+1}$ and $\alpha_i \xrightarrow{st_i,\neg\varphi_i} \alpha'_{i+1}$. Due to atomic execution of message servers, we can compactly represent each execution path of m as a transition $t : s \xrightarrow{m,\gamma(t)} s'$ to denote removing message m from the queue of a rebec and executing it. Consequently, execution of message server m with n annotated statements leads to 2^n potential transitions from the current state s. The application condition of t is the conjunction of the application conditions of sub-transitions that constitute the path that t represents. For example, four possible transitions that represent execution of $m : \langle st_1, st_2 \rangle$ are: $s \xrightarrow{m,\neg\varphi_1\wedge\neg\varphi_2} s'$, $s \xrightarrow{m,\neg\varphi_1\wedge\varphi_2} s''$, $s \xrightarrow{m,\varphi_1\wedge\neg\varphi_2} s'''$, and $s \xrightarrow{m,\varphi_1\wedge\varphi_2} s''''$.

4.4 Product Generation

Given a decided configuration c, a product can be derived from the model of the product family automatically. For this purpose, every application condition φ is evaluated by substituting all of its variables (each corresponds to a feature) by *true/false* based on c. By $c \vDash \varphi$ we denote that configuration c makes application condition φ *true*, otherwise $c \nvDash \varphi$.

Delta Model Having a core Rebeca model M_0 along with a delta model $D = (\Delta, \Gamma, \prec)$, a product with configuration c is obtained by applying every delta $\delta \in \Delta$ such that $c \vDash \Gamma(\delta)$ to M_0 considering the application order of deltas specified by $\prec$. The result is a plain Rebeca model for configuration c.

We define $\Delta|_c \subseteq \Delta$ to contain all deltas that are applicable in c: $\Delta|_c = \{\delta_i \mid \delta_i \in \Delta \ \wedge \ c \vDash \delta_i\}$. Moreover, we assume that if $i < j$, either $\delta_i \prec \delta_j$ or δ_i and δ_j are unordered. Accordingly, we denote the model of the derived product corresponding to configuration c by $M_{\Delta|_c} = \delta_{c_k}(...(\delta_{c_1}(M_0))...)$ where $\Delta|_c = \{\delta_{c_1}, ..., \delta_{c_k}\}$.

Annotated Model The projection of an annotated Rebeca model R over a decided configuration c, denoted by $R|_c$, is a plain Rebeca model where the application conditions of every annotated reactive class, known rebec, state variable, message server, and statement are evaluated and those entities that their application condition does not hold for c are removed.

Note that the result of product generation (for delta-oriented or annotated models) may be a model that is not syntactically correct. For example, a rebec may send a message to another rebec that does not exists in the current configuration because it is removed by a delta (in case of delta modeling) or it is annotated with an application condition that does not hold (in case of annotative modeling). These inconsistencies can be detected by analyzing the model of product family statically. In this paper, we assume that every product that is derivable from the model of the product family is syntactically correct.

5 Model Checking Delta-oriented Rebeca Models

A naive approach to model check SPLs is to generate the Rebeca model of every possible valid product (as we described in 4.4), then model check each product individually. This way, we lose the benefit of having commonalities among the products in the family. In this section, we propose an approach to transform a delta model to an annotated Rebeca model. Given the underlying FTS of an annotated Rebeca model, we can take benefit from variability-aware model checking technique proposed in [20] to model check delta-oriented actor systems. We can also use the late feature binding approach, proposed in [17] to handle variability in the model itself and use the existing model checker of Rebeca.

5.1 Transforming Deltas to Annotations

To transform delta model $D = (\Delta, \Gamma, \prec)$ to the corresponding annotated model, we modify the core model M_0 (which is a plain Rebeca model), according to the deltas defined in D. We assume that deltas with smaller identifiers has lower priority than deltas with greater identifiers. We start by changing the core model M_0 based on the modifications specified by delta δ_1 which results in the model M_1. Likewise, the model M_i is obtained by applying δ_i to M_{i-1}. Due to our earlier assumption on unambiguity and correctness of delta models, the transformation results in a unique annotated model. Moreover, it is not required to deal with cases such as removing an entity that does not exits.

We define a model to be the set of all entities that exist in it. An entity is a reactive class, message server, state variable, known rebec, or statement. Each entity e is represented by a pair $e = (n, d)$ where n is the name of the entity and d

```
reactiveclass Controller() {
  knownrebecs {
    CoffeeMaker cm;
    @tea TeaMaker tm;
  }

  statevars {
    int req, cash, change, cost;
  }

  msgsrv initial() {
    cash, change = 0;
    cost = 1;
    self.receivePayment();
  }

  msgsrv nextOrder() {
    @¬(¬coffee ∧ tea) {
      @¬(¬coffee) {
        @¬tea { cm.makeCoffee(cash); }
        @tea { I2 }
      }
      @¬coffee { I1 }
    }
    @(¬coffee ∧ tea) { I6 }
  }

  msgsrv receivePayment() {
    cash = ?(1,2,3);
    self.nextOrder();
  }

  msgsrv returnChange(int c) {
    change = cash - c;
    cash = 0;
    self.receivePayment();
  }
}
```

```
@coffee reactiveclass CoffeeMaker() {
  knownrebecs {
    Controller ctrl;
  }

  statevars {
    boolean addingCoffee;
    int cost;
  }

  msgsrv initial() {
    addingCoffee = false;
    cost = 1;
  }

  msgsrv makeCoffee(int cash) {
    @¬(milk ∧ sugar) {
      @¬sugar {
        @¬milk {
          addingCoffee = true;
          self.complete();
        }
        @milk { I3 }
      }
      @sugar { I4 }
    }
    @(milk ∧ sugar) { I5 }
  }

  msgsrv complete() {
    ctrl.returnChange(cost);
    addingCoffee = false;
    cost = 1;
  }
}

@tea reactiveclass TeaMaker() {
  ...
}
```

Fig. 3. The annotated Rebeca model of the coffee machine family

is its definition. For simplicity, we do not discuss the formal definition of d in this paper. Informally, known rebecs and state variables are defined by their types. A message server is defined by its parameters and its sequence of statements. Finally, a reactive class is defined by its set of state variables, known rebecs, and message servers. We assume unique names for every known rebec, state variable, and message servers. Unique names for these entities can be obtained by adding the name of their corresponding reactive class as a prefix to their names.

Having $\mathcal{F}$ as the feature set, we assume that function $\mathcal{A}_i : M_i \rightarrow \Phi_{\mathcal{F}}$ returns the application condition by which each entity is annotated in M_i. Function $\mathcal{A}_0$ returns *true* for all entities in M_0. By applying δ_{i+1} on M_i, we may add new entities to M_i or change the annotations of existing ones. We do not eliminate any entity from M_i as removing or modifying an entity is handled by updating its corresponding annotation. Thus, all the definitions of an entity specified by different deltas, coexist in the annotated model. These definitions are distin-

guished by their annotations. Having an unambiguous delta model, only one of these definitions is applicable for a specific configuration. The effect of delta δ_{i+1} on model M_i is captured in the annotated model as follows.

Adding an Entity Suppose δ_{i+1} adds entity $e = (n, d)$ to the model. Note that M_i may include one or more entities with the same name. This happens when for some $j < i$, δ_{j-1} adds an entity which is then removed by δ_j and added again in δ_{j+1}. In this case, we compare definition d with every definition for n in M_i. To handle adding an entity $e = (n, d)$ we consider the following cases:

- When e does not exist in M_i (there is no entity in M_i with name n), we add e to M_i and annotate it with its corresponding application condition: $\mathcal{A}_i(e) = \Gamma(\delta_{i+1})$.
- If d is different from all existing definitions in M_i (for the entities with name n), a new entity is added to M_i along with its annotation $\mathcal{A}_i(e) = \Gamma(\delta_{i+1})$.
- Otherwise, if there exists an entity $e' = (n, d')$ where d and d' are the same, we update the annotation of e' as: $\mathcal{A}_i(e') = \mathcal{A}_{i-1}(e') \vee \Gamma(\delta_{i+1})$.

We consider the definition of two message servers the same if they have the same parameters and the same sequence of statements. Two state variables/known rebecs with the same name are equal if they have the same type. Definitions of two reactive classes are equal if they have the same set of state variables, known rebecs, and message servers with identical definitions.

Removing an Existing Entity To handle removing an entity (n, d), we modify the annotation of all entities in M_i with the name n. For each $e_k = (n, d_k) \in M_i$, we conjunct its annotation with $\neg\Gamma(\delta_{i+1})$: $\mathcal{A}_i(e_k) = \mathcal{A}_{i-1}(e_k) \wedge \neg\Gamma(\delta_{i+1})$.

Note that by the above conjunction, we preserve the higher priority of δ_{i+1} over previously applied deltas as when $\Gamma(\delta_{i+1})$ holds, it makes the entire formula *false*. On the other hand, if we apply another delta δ_j later ($j > i$) to add an entity with the same name and definition again, the new annotation will be $(\mathcal{A}_{i-1}(e_k) \wedge \neg\Gamma(\delta_{i+1})) \vee \Gamma(\delta_j)$. Consequently, the entity would be included in the model if a delta with higher priority adds the entity again.

Modifying the Implementation of a Message Server We assume that I_0 is the initial implementation of message server m. We consider δ_{m_i} to be the i^{th} delta that changes the implementation of m. By applying $\delta_{m_{k+1}}$, the implementation of m changes from I_k to I_{k+1}. If $\delta_{m_{k+1}}$ specifies $I_{\delta_{k+1}}$ as the new implementation of m, then I_k is obtained by the sequential composition of I_k and $I_{\delta_{k+1}}$ annotated by $\neg\Gamma(\delta_{m_{k+1}})$ and $\Gamma(\delta_{m_{k+1}})$ respectively. This way, $I_{\delta_{k+1}}$ is executed only when $\Gamma(\delta_{m_{k+1}})$ holds. Consequently, that the body of message server m is defined recursively as: $I_{k+1} = \{\ @\neg\Gamma(\delta_{m_{k+1}})I_k;\ \ @\Gamma(\delta_{m_{k+1}})I_{\delta_{k+1}}\ \}$

The Coffee Machine Example: Transformation. Figure 3 shows the annotated Rebeca model after applying the deltas δ_2, δ_3, δ_4, δ_1, δ_{12}, and δ_{34}. We omit the details of each implementation I_i. □

5.2 Justification

In this section, we explain how our proposed approach may be justified intuitively. Proving the correctness of the approach formally is in our future agenda. Note that the following arguments only holds for decided configurations.

A model of a product family consists of a core Rebeca model M_0 along with its corresponding delta model D. By applying the applicable deltas to M_0 with respect to their predefined order, $M_{\Delta|_c}$ is obtained which is the model of the individual product with configuration c. The semantics of the resulting Rebeca model can be described using an LTS which we denote it by $[\![M_{\Delta|_c}]\!]$. We may transform the delta-oriented model of the product family to its corresponding annotative model M_a. The semantics of M_a, denoted by $[\![M_a]\!]^\star$, is defined using an FTS. Moreover, we may project M_a over a configuration c to obtain the Rebeca model of an individual product which $[\![M_a|_c]\!]$ represents its semantics.

The correctness of the proposed transformation can be established by proving that the underlying transition systems of $M_{\Delta|_c}$ and $M_a|_c$ are bisimilar: $[\![M_{\Delta|_c}]\!] \approx [\![M_a|_c]\!]$. According to the transformation rules, both transition systems have the same set of message servers with same the behaviors as their action set. Therefore, starting from the initial states, both transition systems have the same set of enabled actions where taking each action implies the same behavior in both transition systems.

The variability-aware model checking is applicable on the FTS $[\![M_a]\!]^\star$. We can justify that such FTS includes the behavior of all products of the delta model by defining a refinement relation between two FTSs and proving that for every configuration c, $[\![M_{\Delta|_c}]\!] \sqsubseteq [\![M_a]\!]^\star$. Given two featured transition systems T_A and T_B, we say T_A refines T_B, denoted $T_A \sqsubseteq T_B$, if and only if there is a relation $\mathcal{R}$ between the states of their underlying transition systems such that $\mathcal{R}(s_{A_0}, s_{B_0})$. Moreover, If $\mathcal{R}(s_A, s_B)$ and there exist transition $t : s_A \xrightarrow{a} s'_A$ then there exists transition $t' : s_B \xrightarrow{a} s'_B$ such that $\mathcal{R}(s'_A, s'_B)$ and $\gamma(t) \Rightarrow \gamma(t')$. According to the proposed transformation rules and the presented semantics for annotated Rebeca models, $[\![M_{\Delta|_c}]\!] \sqsubseteq [\![M_a]\!]^\star$ can be justified intuitively for every configuration c. Note that every transition system (such as $[\![M_{\Delta|_c}]\!]$) is trivially an FTS with $\gamma(t) = \mathit{true}$ for every transition t.

5.3 Model Checking

We can use the model checking algorithm tailored for product families in [20] or use the late feature binding approach proposed in [17] to model check the resulting annotated model.

Variability Aware Model Checking In this approach, we use the underlying FTS of a Rebeca model to apply the variability-aware model checking technique developed to verify product families. This way, the model checker returns all the products that satisfy the given property along with counter-examples for those products that violate the property. To take benefit from such technique, we should alter the current compiler of Rebeca and adapt its model checker accordingly. This is one of our future work.

Traditional Model Checking To use the existing compiler and model checker of Rebeca, we transform the annotated model to a plain Rebeca model which handles variability within the model itself. Annotation $@\varphi$ for a statement s can be modeled using conditional statement `if(`φ`) s;` itself. To decide on including or excluding feature f, we use the late feature binding approach proposed in [17]

where such decision is made just before using a feature to optimize the number of generated states. For this purpose, we model each feature f using an integer variable v_f where its value represent if it is included ($v_f = 1$), excluded ($v_f = 0$), or it is not included nor excluded ($v_f = -1$) yet. We decide on the value of a feature variable by adding `if(`v_f `== -1)` v_f `= ?(0,1)`, just before its usage in the application condition of a statement. Such statement non-deterministically includes or excludes f when no decision is made for it yet. After transforming the annotated Rebeca model to a plain model, we model check it using existing model checker.

Result We have applied our approach on an extended version of the coffee machine example presented in this paper. We verified the annotated model against deadlock by replacing annotations with conditional statements, then using the existing model checker of Rebeca. Deriving every product from the delta model and model checking it separately leads to $20,596$ total states for all products. However, by transforming the delta model to an annotated model and applying the late binding technique, $1,360$ states are generated for the entire family.

6 Conclusion

In this paper, we presented an approach to model product families in a high-level and modular manner, using delta-oriented modeling. To model check such a model, we transform it to a corresponding annotated model with its semantics defined by featured transition systems. Such transition systems can be verified using a variability-aware model checking technique to obtain the products that satisfy the given property. We may also apply the late feature binding technique to verify the entire family using existing model checking techniques. The result of applying our proposed approach on a coffee machine case study shows that it is more efficient to transform a delta model to an annotated one, then model check the entire family, rather than deriving each product from the delta model and verify them individually.

References

1. Pohl, K., Böckle, G., Linden, F.J.v.d.: Software Product Line Engineering: Foundations, Principles and Techniques. Springer-Verlag New York, Inc. (2005)
2. Kang, K.C., Cohen, S.G., Hess, J.A., Novak, W.E., Peterson, A.S.: Feature-oriented domain analysis (FODA) feasibility study. Technical report, Carnegie-Mellon University Software Engineering Institute (November 1990)
3. Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: Delta-oriented programming of software product lines. In: Proc. Software product lines: going beyond. SPLC'10, Springer-Verlag (2010) 77–91
4. Ebert, C., Jones, C.: Embedded software: Facts, figures, and future. Computer **42** (April 2009) 42–52

5. Bruns, D., Klebanov, V., Schaefer, I.: Verification of software product lines with delta-oriented slicing. In: Proc. Formal verification of object-oriented software. FoVeOOS'10, Springer-Verlag (2011) 61–75
6. Thüm, T., Schaefer, I., Hentschel, M., Apel, S.: Family-based deductive verification of software product lines. In: Proc. Generative Programming and Component Engineering. GPCE '12, ACM (2012) 11–20
7. Hähnle, R., Schaefer, I.: A Liskov principle for delta-oriented programming. In: Proc. Leveraging Applications of Formal Methods, Verification and Validation. ISoLA'12, Springer-Verlag (2012) 32–46
8. Damiani, F., Owe, O., Dovland, J., Schaefer, I., Johnsen, E.B., Yu, I.C.: A transformational proof system for delta-oriented programming. In: Proc. Software Product Line Conference - Volume 2. SPLC '12, ACM (2012) 53–60
9. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press (2000)
10. Classen, A., Heymans, P., Schobbens, P.Y., Legay, A., Raskin, J.F.: Model checking lots of systems: efficient verification of temporal properties in software product lines. In: Proc. Int'l Conf. on Software Eng. ICSE'10 (2010) 335–344
11. Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using Rebeca. Fundam. Inf. **63**(4) (June 2004) 385–410
12. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Schaefer, I., Schäfer, J., Schlatte, R., Wong, P.Y.H.: Modeling spatial and temporal variability with the hats abstract behavioral modeling language. In: SFM. (2011) 417–457
13. Rebeca research group: Afra integrated verification environment for Rebeca, http://www.rebeca-lang.org
14. Plath, M., Ryan, M.: Feature integration using a feature construct. Sci. Comput. Program. **41**(1) (September 2001) 53–84
15. Larsen, K.G., Nyman, U., Wasowski, A.: Modal I/O automata for interface and product line theories. In: Proc. European Symposium on Programming. ESOP'07, Springer-Verlag (2007) 64–79
16. Sabouri, H., Khosravi, R.: An effective approach for verifying product lines in presence of variability models. In: Proc. Software product lines - Volume 2. (2010) 113–120
17. Sabouri, H., Jaghoori, M.M., de Boer, F.S., Khosravi, R.: Scheduling and analysis of real-time software families. In: Proc. Computer Software and Applications, IEEE Computer Society (2012)
18. Gruler, A., Leucker, M., Scheidemann, K.: Modeling and model checking software product lines. In: Proc. Formal Methods for Open Object-Based Distributed Systems. FMOODS '08, Springer-Verlag (2008) 113–131
19. Muschevici, R., Clarke, D., Proença, J.: Feature Petri Nets. In: Proc. Software product lines - Volume 2. (2010) 99–106
20. Classen, A., Cordy, M., Heymans, P., Legay, A., Schobbens, P.Y.: Model checking software product lines with SNIP. International Journal on Software Tools for Technology Transfer (2012) 1–24
21. Clarke, D., Helvensteijn, M., Schaefer, I.: Abstract delta modeling. In: Proc. Generative programming and component engineering. GPCE '10 (2010) 13–22
22. Jaghoori, M.M., Sirjani, M., Mousavi, M.R., Khamespanah, E., Movaghar, A.: Symmetry and partial order reduction techniques in model checking Rebeca. Acta Inf. **47**(1) (January 2010) 33–66