*Journal of Universal Computer Science, vol. 19, no. 2 (2013), 207-232*
*submitted: 7/5/12, accepted: 31/12/12, appeared: 28/1/13* © *J.UCS*

# Modeling and Verification of Reconfigurable Actor Families

**Hamideh Sabouri**
(School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
sabouri@ece.ut.ac.ir)

**Ramtin Khosravi**
(School of Electrical and Computer Engineering
University of Tehran, Tehran, Iran
r.khosravi@ut.ac.ir
School of Computer Science, Institute for Research in Fundamental
Sciences (IPM), Tehran, Iran)

**Abstract:** Software product line engineering enables proactive reuse among a set of related products through explicit modeling of commonalities and differences among them. Features are usually used to distinguish different products as a product is identified by its supported feature set that is represented by a configuration. Dynamic product lines enhance flexibility of a product by allowing run-time reconfiguration. In this paper, we focus on modeling and verification of families of concurrent and distributed systems that are reconfigurable. To this end, we introduce the notion of variability in actor models to achieve family of reconfigurable actors. Then, we present our methodology to model this concept using the actor-based modeling language Rebeca. The model checking backbone of Rebeca enables us to ensure establishment of certain constraints on reconfigurations. We show the applicability and effectiveness of our approach by applying it on a set of case studies.
**Key Words:** Dynamic software product lines, Model Checking, Actor models, Reconfiguration
**Category:** D.2.4, D.2.13

## 1 Introduction

In software product line engineering we focus on developing a product family, instead of developing several single products, by developing software applications using platforms and mass customization. To this end, commonalities and differences among products should be modeled explicitly [Pohl et al.(2005)]. Feature models are widely used for this purpose. A product is identified by its configuration representing a combination of features. Product family is the set containing all of the valid feature combinations [Kang et al.(1990)] (Section 3.1).

Actor model is a well-known model of concurrent and distributed computation [Agha(1990)] (Section 4.1). In this model, actors are primitive units of concurrent computation. An actor receives messages and in response, makes local decisions and sends a number of messages to other actors. In this paper, we introduce the notion of *actor family* to facilitate development and analysis of families of related concurrent and distributed systems by employing software product line paradigm. An actor family model consists of *featured actors* whose behavior varies according to different configurations. Reconfiguring actor families and featured actors are supported in our proposed approach which makes it usable for modeling dynamic product lines [Hallsteinsen et al.(2008)] as well.

To this end, we elaborate the semantics of actor families where the presence of an actor, the messages that it responds, and its local behavior vary between different configurations (Section 4.2). To realize the notion of actor family in practice, we present a methodology in Section 5 to model this concept using Rebeca modeling language [Sirjani et al.(2004)]. Rebeca, is an actor-based language with a formal foundation for modeling and verifying concurrent and distributed systems (Section 3.2). We extend the syntax of Rebeca to make modeling variations in the behavior convenient. However, the semantics of Rebeca is preserved as these extensions are describable using original syntax by a straightforward transformation. Moreover, we propose two approaches to make handling message passing among featured actors easier for the modeler (Section 5.3).

We may verify a model of actor families against properties using model checker of Rebeca, Modere [Jaghoori et al.(2006)]. The properties are expressed in linear temporal logic (LTL) for this purpose [Emerson(1990)]. In the context of distributed systems, we cannot enforce global monitoring of system's state to decide on reconfigurations. Instead, each actor makes a decision to alter the current configuration considering its own state. Therefore, we take benefit from the model checker of Rebeca to ensure that reconfigurations follow the intended rules and do not lead the system to invalid configurations. (Section 6.2).

The main contributions of this paper can be summarized as follows.

- Introducing actor families along with their semantics to facilitate modeling and analysis of families of related concurrent and distributed systems.
- Supporting reconfiguration of actor families to make our approach applicable for modeling dynamic product lines
- Presenting a methodology to model actor families in Rebeca modeling language by extending the syntax while preserving its semantics.
- Supporting coarse-grained variability (inclusion/exclusion of actors) and fine-grained variability (variability in responded messages and local behavior) in actor families.
- Providing facilities to coordinate message passing among featured actors elegantly.

– Taking benefit from model checking support of Rebeca to ensure that all obtainable configurations in an actor family conform to a set of predefined constraints, by verifying a set of properties

We use coffee Machine family [Fantechi and Gnesi(2008)] as the running example in the paper. A coffee machine may serve coffee, tea, or water. Adding extra milk to coffee may be supported optionally. Coffee machine holds a certain amount of milk and needs to be refilled when it gets empty.

## 2 Related Work

So far, several approaches have been developed to model product lines using transition systems [Larsen et al.(2007a), Larsen et al.(2007b), Classen et al.(2010), Sabouri and Khosravi(2010)], process algebra [Gruler et al.(2008)], Petri-nets [Muschevici et al.(2010)], and ABS modeling language [Clarke et al.(2010)]. In [Muschevici et al.(2010), Damiani and Schaefer(2011)], modeling and verification of dynamic product lines is considered. These approaches capture the behavior of the entire product family in a single model by including the variability information in it. In other words, it is specified in the model how the behavior changes when a feature is included or excluded.

Families of transition systems are modeled using modal transition systems (MTS) in [Larsen et al.(2007a)]. An MTS consists of *may* and *must* transitions. A may transition can be removed or can be left when refining an MTS which leads to different products. A must transition should appear in all of the refinements. Feature transition systems (FTS) are introduced in [Classen et al.(2010)] to model families of transition systems. In FTS, annotations are allocated to transitions to indicate which transitions of the model correspond to which features.

In [Gruler et al.(2008)], the authors model a product family using an extension of CCS process algebra named PL-CCS. PL-CCS, extends CCS by the *variant* operator which specifies alternative process. A CCS model can be derived from a PL-CCS model, by selecting one process from alternative process described by each of the variant operators.

Families of Petri-net models are modeled in [Muschevici et al.(2010)] using feature Petri-nets (FPN) and Dynamic feature Petri-nets (DFPN). An FPN has application conditions attached to its transitions. An application condition is a boolean logical formula over a set of features and indicates the feature combinations that enable a transition. A Petri-net model can be obtained by projecting an FPN onto a feature selection. The feature selection can be updated in DFPN by associating update expressions to transitions. Upon firing a transition, updates affect the feature selection by adding or removing features from it.

In [Clarke et al.(2010)], delta modeling language (DML) that is based on the concept of delta modeling [Schaefer et al.(2010)] is used to describe code-level variability. In delta-oriented programming, the implementation of a product family contains a *core module* and a set of *delta modules*. The core module consists of a set of classes that implement a possible product corresponding to a valid feature configuration. Delta modules specify the changes that should be applied on the core module to obtain a new product. In this approach, an application condition is defined over the set of features and it is associated to each delta module to determine for which feature configurations the modification of the delta module is applied. In [Damiani and Schaefer(2011)], the authors extend this work to support dynamic product lines as well. In this approach, a reconfiguration automaton describes how a configuration can be changed to other ones.

In [Sabouri and Khosravi(2011)], we add variability to Rebeca models to model check product families. The main focus of [Sabouri and Khosravi(2011)] is decreasing the cost of model checking product families by reducing the number of verified products and modeling product lines is not the main concern of this paper.

To the best of our knowledge, our work is the first formal attempt to introduce the notion of variability to the semantics of actor models that leads to the concept of actor families. Reconfiguration is supported in our proposed formalization. Furthermore, we use a high-level actor based language with model checking support to make modeling and verifying families of concurrent and distributed systems attainable.

## 3 Preliminaries

### 3.1 Feature Models

Commonalities and differences among different products are modeled explicitly in software product line engineering. Feature models 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 is a distinctive aspect, quality, or characteristic of a system [Kang et al.(1990)].

A basic 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 [Benavides et al.(2010)]. A feature model can be represented by a corresponding propositional logic formula in terms of a set of boolean variables [Batory(2005)]. Each boolean variable corresponds to a feature and its value indicates if the feature is included or excluded. The ultimate formula $\phi_F$ is the conjunction of implications from: (1) every child feature to its parent feature, (2) every parent to its mandatory child features, (3) every parent to

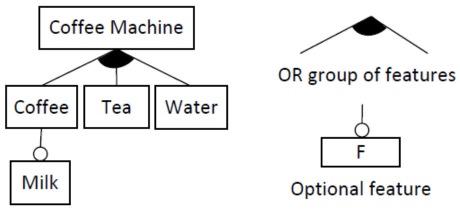


**Figure 1:** The feature model of the coffee machine example

or/xor of its children that have an *or*/*xor* relationship, (4) every feature $f$ to other features that $f$ requires, (5) and every feature $f$ to the negation of other features that $f$ excludes.

Using the set of features $\mathcal{F}$ and the propositional formula $\phi_F$ that represents the constraints among features, we define a feature model as follows and use it in the rest of the paper. An advantage of this method is that we are not limited to a specific notation for describing feature models and we use their semantics instead.

**Definition 1 Feature Model.** A feature model is a pair $(\mathcal{F}, \phi_{\mathcal{F}})$ where

– $\mathcal{F} = \{f_1, ..., f_m\}$ is the set of features
– $\phi_F$ is a propositional logic formula in terms of features that represents the constraints among them □

A configuration vector keeps track of inclusion or exclusion of features.

**Definition 2 Configuration Vector.** Having a set of features $\mathcal{F}$ with $m$ features, a configuration vector is defined as $\theta \in \{true, false\}^m$ where

– $\theta_i = true$ represents inclusion of $f_i$
– $\theta_i = false$ represents exclusion of $f_i$ □

We assume that $\Theta_{\mathcal{F}}$ is the set of all possible configurations over $\mathcal{F}$. A configuration $\theta$ over feature set $\mathcal{F}$ satisfies a propositional logic formula $\phi$ over $\mathcal{F}$ (denoted by $\theta \vDash \phi$) if substituting boolean variables of $\phi$ with *true*/*false* according to $\theta$ leads to a satisfiable propositional logic formula. A configuration is valid according to a feature model $(\mathcal{F}, \phi_{\mathcal{F}})$ when $\theta \vDash \phi_{\mathcal{F}}$.

*The Coffee Machine Example: Feature Model.* The corresponding feature model of the coffee machine is depicted in Figure 1. Coffee, tea, and water features have an *or* relationship implying that the machine serves one of these

```
reactiveclass Controller() {

  knownrebecs {
    CS cs;
  }

  statevars {
  }

  msgsrv initial() {
    self.nextOrder();
  }

  msgsrv nextOrder() {
    cs.serveCoffee();
  }

}
```

```
reactiveclass CS() {

  knownrebecs {
    Controller ctrl;
  }

  statevars {
  }

  msgsrv initial() {
  }

  msgsrv serveCoffee() {
    ctrl.nextOrder();
  }

}
```

```
main {
  CS cs(ctrl):();
  Controller ctrl(cs):();
}
```

**Figure 2:** Rebeca model of a coffee machine that only serves coffee

drinks at least. Adding extra milk to coffee is optional. Formally, this feature model can be described as $(\mathcal{F}, \phi_{\mathcal{F}})$ where:

$$\mathcal{F} = \{cm, c, t, w, m\}$$

$$\phi_{\mathcal{F}} = (c \rightarrow cm) \wedge (t \rightarrow cm) \wedge (w \rightarrow cm) \wedge (m \rightarrow c) \wedge (cm \rightarrow (c \vee t \vee w))$$

where $cm$, $c$, $t$, $w$, and $m$ represent coffee machine, coffee, tea, water, and milk features respectively. A configuration $\theta = \langle true, false, true, false, true \rangle$ over $\mathcal{F}$ is not a valid configuration according to $\phi_{\mathcal{F}}$ as $m \rightarrow c$ would be evaluated to *false* making $\phi_{\mathcal{F}} = false$ (we assume that $\theta_1$, $\theta_2$, $\theta_3$, $\theta_4$, and $\theta_5$ represent coffee machine, coffee, tea, water, and milk features respectively). $\square$

### 3.2 Rebeca Modeling Language

Rebeca is an actor-based language for modeling concurrent and distributed systems. Rebeca models such 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 variables, and each rebec has a single thread of execution that is triggered by reading messages from the top of an unbounded message queue. When a message is taken from the top of queue, its corresponding message server is invoked. The message server may change the value of some of the state variables and send messages to other rebecs. In [Razavi et al.(2011)], Rebeca has been extended with global variables to support hardware-software co-design.

*The Coffee Machine Example: Rebeca Model.* Figure 2 shows the Rebeca code of a coffee machine that only serves coffee. The controller rebec is for managing different orders which in this model is only coffee order. Upon receiving an order, it sends a message to the coffee server to make coffee. The coffee server rebec informs the controller to take the next order when it handled the request. Note that this model do not contain variability and only shows the original syntax of the language. □

## 4 Introducing Variability to Actor Models

Software product line engineering enables proactive reuse by developing a family of related products. There are two main approaches to develop software product lines: compositional approach and annotative approach [Kästner et al.(2008)]. In the compositional approach, features are implemented as distinct code units. These code units are reused when the corresponding units are composed to generate each product. In annotative approaches, features are implemented using implicit or explicit annotations in the source code. In [Kästner and Apel(2008)] these approaches are compared and their complementary strengths are stated. In the compositional approach, feature traceability is straightforward, but variability in the behavior is only provided in a coarse-grained manner. On the other hand, the annotative approach supports variability in a fine-grained manner. As a result, to be more practical, integration of compositional and annotative approaches is recommended.

We take benefit of the compositional approach as well as the annotative approach by considering each actor as a code unit where presence of each actor, the set of messages it responds, and its local behavior may be variable for different configurations. In this section, we first present the semantics of actor systems. Then, we elaborate the semantics of family of reconfigurable actors.

### 4.1 Actor Models

In actor model [Agha(1990)], actors are primitive units of concurrent computation. Each actor may receive messages from other actors (or itself). The received messages are stored in the queue of the actor. A message is executed after it is taken from the top of the queue and in response, the actor may change its local state and/or send a number of messages. When the execution of a message is finished, the next message is picked form the queue by the actor. We define an actor in terms of a finite set of messages and corresponding statements.

**Definition 3 Actor.** An actor is a pair $a = (M, V)$ where:

- $M = \{m_0, m_1, ..., m_k\}$ is the set of message servers. A message server has statements $\langle s_1, ..., s_l, \epsilon\rangle$ where $\epsilon$ denotes the end of message server execution. The *initial message* is denoted by $m_0$ which is the first message that will be picked from the queue by the actor.
- $V = \{v_1, ..., v_l\}$ is the set of state variables □

To describe the semantics of an actor model, we consider a *closed system* of actor set $A = \{a_1, ..., a_n\}$ where the actors execute concurrently and communicate with each other via asynchronous message passing. The state of the system consists of the local states of its individual actors.

**Definition 4 Local State of an Actor.** We define the local state of actor $a = (M, V)$ as a triple $\varsigma = \langle s, \mathcal{V}, q\rangle$:

- $s$ is the next statement to be executed
- $\mathcal{V}$ is an assignment of values to variables
- $q = [q_1, ..., q_k]$ is a queue containing messages □

In the initial state, the first statement of the initial message $m_0$ is ready to be executed, the variables are initialized to their default values, and the queue is empty. The local actor semantics and global actor semantics are defined as follows.

**Definition 5 Local Actor Semantics.** The local semantics of an actor $a$ is defined as a system of labeled transitions of the following forms

1. $\langle s_i, \mathcal{V}, q\rangle \xrightarrow{\tau} \langle s_j, \mathcal{V}', q\rangle$ denotes execution of a non-send statement $s_i$ where $s_i \neq \epsilon$, $s_j$ is the next statement that should be executed after $s_i$, and $\mathcal{V}'$ represents new values of variables after execution of a statement
2. $\langle s_i, \mathcal{V}, q\rangle \xrightarrow{a_i!m} \langle s_j, \mathcal{V}, q\rangle$ denotes sending message $m$ to actor $a_i$ where $s_i \neq \epsilon$ and $s_j$ is the next statement that should be executed after $s_i$
3. $\langle \epsilon, \mathcal{V}, q\rangle \xrightarrow{a?m} \langle s_1, \mathcal{V}, q'\rangle$ denotes taking a message from top of the queue whose first statement is $s_1$ □

**Definition 6 Global Semantics.** The semantics of a closed system of actors $A = \{a_1, ..., a_n\}$ is defined as

$$\frac{\varsigma_i \xrightarrow{\tau} \varsigma_i'}{\varsigma_1, ..., \varsigma_i, ..., \varsigma_n \xrightarrow{\tau} \varsigma_1, ..., \varsigma_i', ..., \varsigma_n}$$

$$\frac{\varsigma_i \xrightarrow{a_i?m} \varsigma_i'}{\varsigma_1, ..., \varsigma_i, ..., \varsigma_n \xrightarrow{a_i?m} \varsigma_1, ..., \varsigma_i', ..., \varsigma_n}$$

$$\frac{\varsigma_i \xrightarrow{a_j!m} \varsigma_i'}{\varsigma_1, ..., \varsigma_i, ..., \varsigma_j, ..., \varsigma_n \xrightarrow{a_j!m} \varsigma_1, ..., \varsigma_i', ..., \varsigma_j', ..., \varsigma_n}$$

where $\varsigma_j'$ results from $\varsigma_j$ by adding the message $m$ to its queue. □

Intuitively, each actor has a single thread of execution which is triggered by picking a message from the top of the queue. When a message is taken from the queue, the corresponding message server is executed which may change the value of some variables and send a number of messages to the other actors. The concurrency is modeled by interleaving the execution of messages.

### 4.2 Family of Reconfigurable Actors

In a family of reconfigurable actors, an individual actor can be (re)configured as well as the actor model itself. At the level of actor model, each actor may be included in some configurations and excluded in others. However, inclusion of an actor in a configuration does not necessarily mean that it responds to all incoming messages as at the level of individual actors each message server may be supported only in certain configurations. Furthermore, the local behavior of each message server may change based on different configurations itself. We refer to such actors as *featured actors* as their behavior varies based on the features selected in a configuration. An actor model family is a closed system consisting of featured actors.

#### 4.2.1 Application Conditions

An *application condition* $\psi$ is a propositional logic formula over features and it is used to specify the subset of products in which an actor, a message server, or a statement is available.

**Definition 7 Application Condition.** An application condition over a set of features $\mathcal{F} = \{f_1, ..., f_m\}$ is formed as:

$$\psi ::= true \mid false \mid f_i \mid \neg\psi \mid \psi_1 \wedge \psi_2 \mid \psi_1 \vee \psi_2$$

where $f_i \in \mathcal{F}$. □

We assume that $\Psi_{\mathcal{F}}$ is the set of all possible application conditions over $\mathcal{F}$ and $\Gamma : A \cup M \cup S \mapsto \Psi_{\mathcal{F}}$ maps actors, message servers, and statements to their corresponding application condition, where $A$, $M$, and $S$ are the sets of all actors, message servers, and statements in a closed system of actors.

Application conditions are evaluated based on configurations. A configuration vector $\theta \in \{true, false\}^m$ over feature set $\mathcal{F} = \{f_1, ..., f_m\}$ is used to keep track of availability of features in the actor model. The satisfiability of application condition $\psi$ given the configuration vector $\theta$ is denoted by $\theta \vDash \psi$. We may also check application conditions using a SAT-solver (e.g. [Moskewicz et al.(2001)]) to ensure their conformance to feature model. An application condition $\psi$ is satisfiable with respect to a feature model $(\mathcal{F}, \phi_{\mathcal{F}})$ if $\psi \wedge \phi_{\mathcal{F}}$ is satisfiable. Non-satisfiability of $\psi$ with respect to $\phi_{\mathcal{F}}$ means that $\psi$ does not hold in any product that conforms to the given feature model. Consequently, the corresponding code unit that is guarded by $\psi$ is never executed. An application condition $\psi$ is violable with respect to the feature model if $\neg(\psi \wedge \phi_{\mathcal{F}})$ is satisfiable. Similarly, when $\psi$ is not violable with respect to $\phi_{\mathcal{F}}$, we can conclude that $\psi$ holds in all products and the corresponding code unit is always executed.

*The Coffee Machine Example: Application Conditions.* Application condition $\psi = c \wedge m$ denotes the set of products that support both coffee and milk features. Application condition $\psi = \neg c \wedge m$ is not satisfiable with respect to the feature model as there is no valid product that includes milk while excluding coffee feature. Finally, $\psi = c \vee t \vee w$ is an example of an application condition that is not violable according to the feature model as every valid product serves one type of drink at least. □

### 4.2.2 Semantics

Each actor keeps its own copy of the configuration vector $\theta$ which is $\vartheta$. This copy is updated to the latest configuration when the actor takes a message from the queue. The behavior of a message server is determined according to $\vartheta$ that is equal to $\theta$ when it starts executing. The execution of the message server may be interrupted by transforming control to another actor. However, when its execution continues, the behavior is still specified based on $\vartheta$ which is the configuration that the message server started its execution with it, even if the current configuration of the system is $\theta'$.

**Definition 8 Local State of a Featured Actor.** We define the local state of featured actor $a = (M, V)$ as a four tuple $\varsigma = \langle s, \mathcal{V}, q, \vartheta \rangle$:

- $s$ is the next statement to be executed
- $\mathcal{V}$ is an assignment of values to variables
- $q = [q_1, ..., q_k]$ is a queue containing messages
- $\vartheta$ is a local copy of the configuration vector □

The configuration vector $\theta$ can be altered by actors when executing their message servers. The local semantics of a featured actor is defined using guarded transitions $\xrightarrow{g:\, l}$ where the transition is taken when the guard $g$ holds.

**Definition 9 Local Featured Actor Semantics.** Having configuration vector $\theta$, the local semantics of an actor $a$ is defined as a system of guarded labeled transitions of the form $\varsigma \xrightarrow{g:\ l} \varsigma'$ where

1. $\langle s_i, \mathcal{V}, q, \vartheta \rangle \xrightarrow{\psi:\ \tau} \langle s_j, \mathcal{V}', q, \vartheta \rangle$ denotes execution of a non-send statement $s_i$ where $s_i \neq \epsilon$, $s_j$ is the next statement that should be executed after $s_i$, $\mathcal{V}'$ represents new values of variables after execution of the statement, and $\psi = \Gamma(s_i)$
2. $\langle s_i, \mathcal{V}, q, \vartheta \rangle \xrightarrow{\psi:\ \rho} \langle s_j, \mathcal{V}, q, \vartheta \rangle$ denotes reconfiguration through non-send statement $s_i$ where $s_i \neq \epsilon$, $s_j$ is the next statement that should be executed after $s_i$, $\psi = \Gamma(s_i)$, and $\rho$ distinguishes execution of statements result in reconfiguration from rest of non-send statements
3. $\langle s_i, \mathcal{V}, q, \vartheta \rangle \xrightarrow{\psi:a_i!m} \langle s_j, \mathcal{V}, q, \vartheta \rangle$ denotes sending message $m$ to actor $a_i$ where $s_i \neq \epsilon$, $s_j$ is the next statement that should be executed after $s_i$ and $\psi = \Gamma(s_i)$
4. $\langle s_i, \mathcal{V}, q, \vartheta \rangle \xrightarrow{\psi:\ \alpha} \langle s_j, \mathcal{V}, q, \vartheta \rangle$ denotes skipping statement $s_i$ where $s_i \neq \epsilon$, $s_j$ is the next statement that should be executed after $s_i$, and $\alpha \in \{\tau, \rho, a_i!m\}$ and $\psi = \neg\Gamma(s_i)$
5. $\langle \epsilon, \mathcal{V}, q, \vartheta \rangle \xrightarrow{\psi:\ a?m} \langle s_1, \mathcal{V}, q', \vartheta' \rangle$ denotes taking message $m$ from top of the queue which its first statement is $s_1$ and where $\psi = \Gamma(m) \wedge \Gamma(a)$
6. $\langle \epsilon, \mathcal{V}, q, \vartheta \rangle \xrightarrow{\psi:\ a?m} \langle \epsilon, \mathcal{V}, q', \vartheta \rangle$ denotes dropping message $m$ from top of the queue where $\psi = \neg(\Gamma(m) \wedge \Gamma(a))$ □

According to the above semantics, when a featured actor is executing a message server, the statements that their application conditions do not hold are skipped and the state variables keep their values upon moving to the next statement. A featured actor responds to a message taken from its queue by putting the first statement of the corresponding message server as the next statement to be executed, when the application conditions of the actor and message server hold. Otherwise, the message is dropped and $\epsilon$ remains in the local state. The global semantics of actor families is defined as follows.

**Definition 10 Global Semantics of Actors Family.** The semantics of a closed system of featured actors $A = \{a_1, ..., a_n\}$ is defined as

$$\frac{\varsigma_i \xrightarrow{\psi:\tau} \varsigma_i' \quad \vartheta \vDash \psi \quad \varsigma_i = \langle s, \mathcal{V}, q, \vartheta \rangle}{\varsigma_1, ..., \varsigma_i, ..., \varsigma_n, \theta \xrightarrow{\tau} \varsigma_1, ..., \varsigma_i', ..., \varsigma_n, \theta}$$

$$\frac{\varsigma_i \xrightarrow{\psi:\rho} \varsigma_i' \quad \vartheta \vDash \psi \quad \theta' = \Lambda(s, \theta) \quad \varsigma_i = \langle s, \mathcal{V}, q, \vartheta \rangle}{\varsigma_1, ..., \varsigma_i, ..., \varsigma_n, \theta \xrightarrow{\rho} \varsigma_1, ..., \varsigma_i', ..., \varsigma_n, \theta'}$$

$$\frac{\varsigma_i \xrightarrow{\psi:a_i?m} \varsigma'_i \quad \theta \vDash \psi \quad \varsigma'_i = \langle s, \mathcal{V}, q, \theta \rangle}{\varsigma_1, ..., \varsigma_i, ..., \varsigma_n, \theta \xrightarrow{a_i?m} \varsigma_1, ..., \varsigma'_i, ..., \varsigma_n, \theta}$$

$$\frac{\varsigma_i \xrightarrow{\psi:a_j!m} \varsigma'_i \quad \vartheta \vDash \psi \quad \varsigma_i = \langle s, \mathcal{V}, q, \vartheta \rangle}{\varsigma_1, ..., \varsigma_i, ..., \varsigma_j, ..., \varsigma_n, \theta \xrightarrow{a_j!m} \varsigma_1, ..., \varsigma'_i, ..., \varsigma'_j, ..., \varsigma_n, \theta}$$

where $\varsigma'_j$ results from $\varsigma_j$ by adding the message $m$ to its queue. □

We distinguish statements that alter the configuration using action $\rho$ to capture the notion of reconfiguration in the global semantics. The execution of such a statement may cause changing $\theta$ which is reflected in the second rule of the global semantics. We assume that reconfiguration effect function $\Lambda : S \times \Theta_{\mathcal{F}} \mapsto \Theta_{\mathcal{F}}$ returns the new configuration after reconfiguration through a statement where $S$ is the set of all statements and $\Theta_{\mathcal{F}}$ is set of all possible configurations over feature set $\mathcal{F}$. By $\theta' = \Lambda(s_i, \theta)$ we denote that statement $s_i$ changes the configuration vector from $\theta$ to $\theta'$.

## 5 Methodology

In this section, we propose a methodology to model the concept of family of reconfigurable actors using Rebeca. To this end, we extend the syntax of Rebeca and we show that these extensions do not affect the semantics of the language as they can be described using the original syntax as well. The reason of these extensions is to simplify modeling dynamic product lines. Figure 3 shows the Rebeca model of the coffee machine family.

Our approach is not limited to Rebeca as the proposed methodology can be adapted for other actor-based languages as well. We select Rebeca as it is an actor-based modeling language which was designed in an effort to bridge the gap between formal verification approaches and real-world applications. It is equipped with a direct model checker named Modere [Jaghoori et al.(2006)]. Furthermore, an approach is proposed in [Sabouri and Khosravi(2011)] to reduce the cost of verifying product families modeled by Rebeca.

### 5.1 Representing Configuration Vector

To represent configuration vectors in Rebeca, we consider a global boolean variable $v_{f_i}$ for each feature. We refer to such variables as feature variables.

**Definition 11 Feature Variable.** A feature variable $v_{f_i}$, representing feature $f_i$, is a global boolean variable where:

– $v_{f_i} = \mathit{false}$ denotes exclusion of $f_i$

– $v_{f_i} = true$ denotes inclusion of $f_i$ □

The value of such variables all together form the configuration vector. Using variables to represent features instead of annotation tags is to support reconfiguration by means of changing the value of these variables. Defining these variables globally makes the latest configuration observable for all rebecs immediately.

An alternative option would be considering a configurator rebec to manage the configuration vector and reconfigurations. Such a rebec has feature variables as its state variables and is responsible of initializing and changing the variables upon receiving reconfiguration requests from other rebecs. After altering the configuration vector, it informs other rebecs of the latest configuration. The drawback of this approach to manage (re)configuration is its message passing overhead that leads to significant growth in state space.

Note that because of atomic execution of message servers in Rebeca, we do not have to keep a local copy of the configuration vector in each rebec. When a message server starts its execution in configuration $\theta$, its entire behavior is determined according to this configuration. The only exception is when message server $m$ reconfigures $\theta$ by assigning value to feature variable $v_{f_i}$, then uses the same feature variable in an application condition of one of its statements. In this case, a local copy of $v_{f_i}$ should be kept and be updated when execution of $m$ starts. Despite this fact, we presented our formalization based on non-atomic execution of messages to avoid restricting the proposed approach to its current implementation.

*The Coffee Machine Example: Feature Variables.* The feature variables of the coffee machine example are defined globally as $v_{cm}$, $v_c$, $v_t$, $v_w$, $v_m$. □

### 5.2 Modeling Featured Actors

In Rebeca, the behavior of an actor is described using a reactive class. To capture the semantics of featured actors, we should embed the corresponding behavior of dropping messages from the queues and variation in local behavior of actors, in a Rebeca model. To justify the proposed methodology intuitively, we elaborate our modeling approach for each of the rules presented in local semantics of featured actors. For this purpose, we use application conditions defined over feature variables.

#### 5.2.1 Dropping Messages

Rule 6 in definition of local semantics of featured actors denotes dropping message $m$ by actor $a$ when it is not supported in current configuration $\theta$ as the application conditions of $a$ or $m$ do not hold. To represent application conditions of rebecs and message servers, we extend the syntax of Rebeca by the notation $@\psi$ as follows.

```
reactiveclass Controller() {

  knownrebecs {
    CS cs;
    TS ts;
    WS ws;
  }

  statevars { }

  msgsrv initial() {
    self.nextOrder();
  }

  msgsrv nextOrder() {
    send?(cs.serveCoffee(),
          ts.serveTea(),
          ws.serveWater());
  }

}

reactiveclass Filler() @v_m {

  knownrebecs {
    CS cs;
  }

  statevars { }

  msgsrv initial() { }

  msgsrv fill() {
    cs.filed();
  }

}

main {
  CS cs(ctrl):();
  TS ts(ctrl):();
  WS ws(ctrl):();
  Controller ctrl(cs,ts,ws):();
}
```

```
reactiveclass TS() @v_t {

  knownrebecs {
    Controller ctrl;
  }

  statevars {
  }

  msgsrv initial() {
  }

  msgsrv serveTea() {
    ctrl.nextOrder();
  }

}

reactiveclass WS() @v_w {

  knownrebecs {
    Controller ctrl;
  }

  statevars {
  }

  msgsrv initial() {
  }

  msgsrv serveWater() {
    ctrl.nextOrder();
  }

}
```

```
reactiveclass CS() @v_c {

  knownrebecs {
    Controller ctrl;
    Filler f;
  }

  statevars {
    int milk;
  }

  msgsrv initial() {
    milk = 5;
  }

  msgsrv serveCoffee() {
    if(v_m)
      self.addMilk();
    else
      ctrl.nextOrder();
  }

  msgsrv addMilk() @v_m {
    milk = milk - 1;
    if(milk == 0) {
      v_m = false;
      f.fill();
    }
    ctrl.nextOrder();
  }

  msgsrv filled() @v_m {
    milk = 5;
    v_m = true;
  }

}
```

**Figure 3:** Rebeca model of the coffee machine family

```
reactiveclass R() @ψ_r {
  msgsrv m() @ψ_m {
    s_1; ...; s_k;
  }
}
```

In this notation, $\psi_r$ and $\psi_m$ represent the application conditions of reactive class $R$ and message server $m$ respectively. Informally, the above syntax denotes that a rebec instantiated from $R$ is included when $\psi_r$ holds. In addition, message server $m$ is supported when $\psi_m$ holds in the current configuration. An incoming message $m$ would be dropped when one of these application conditions does not hold. However, the above notation does not change the semantics of Rebeca as it can be modeled using the original syntax by guarding the entire body of a message server using an `if` statement on conjunction of corresponding application conditions of the rebec and the message server as follows.

```
reactiveclass R() {
  msgsrv m() {
    if(ψ_r ∧ ψ_m) {
      s_1; ...; s_k;
    }
  }
}
```

As a result, when message $m$ is taken from the queue, its corresponding statements are executed only when the rebec is available and the application condition of the message server holds. Otherwise, no statement is executed and the state variables keep their value while the message is removed from the queue.

*The Coffee Machine Example: Dropping Messages.* In Figure 3, the application condition of the `addMilk` message server of the `CS` rebec is $v_m$. Moreover, the application condition of the rebec `CS` is $v_c$. Consequently, a message to this rebec would be dropped if $\neg(v_c \wedge v_m)$. We could model these application conditions by guarding the entire body of `addMilk` and `filled` message servers with `if`$(v_c \wedge v_m)$ and the body of other message servers with `if`$(v_c)$. However, this makes modeling dropping messages a tedious and error-prone task. □

### 5.2.2 Skipping Statements

According to the rule 4 in the local semantics of featured actors, variability in local behavior is represented by skipping statement $s$ when its application condition does not hold in the current configuration. In Rebeca, we model this concept by associating an application condition $\psi$ to statement $s$ through guarding it by "`if(`$\psi$`) s`". Consequently, the statement $s$ is executed only when its application condition holds and otherwise it is skipped.

*The Coffee Machine Example: Variability in Behavior.* In Figure 3, the behavior of `serveCoffee` message server of the `CS` rebec varies based on inclusion/exclusion of the milk feature. When this feature is included milk is added to coffee, otherwise, the coffee server sends a message to controller to take the next order. □

### 5.2.3 Intuitive Justification

To justify the proposed methodology for modeling featured actors in Rebeca intuitively, we elaborate our modeling approach for each of the rules presented in the definition of local semantics of featured actors.

*Rule 1* describes execution of a non-send statement which may change the value of state variables. In Rebeca, execution of each statement `v = e;` or `if(g)` within message server $m$ of rebec $r$ is a representation of this rule where $v$ is a state variable of $a$, $e$ is a mathematical expression, and $g$ is a boolean expression over state variables. As we described earlier, the application condition $\Gamma(s)$ is modeled using an `if` condition. Consequently, a statement is not executed when its application condition does not hold. As an example, consider the statement `milk = milk - 1;` of the `addMilk` message server in Figure 3 which changes the value of the `milk` state variable after execution.

*Rule 2* denotes reconfiguration of the configuration vector through a statement. This rule is modeled in Rebeca by statement $v_{f_i}$ `= e;` where $v_{f_i}$ is a feature variable and $e$ is a boolean expression over feature variables and state variables of the corresponding rebec. The application condition $\Gamma(s)$ is modeled by guarding the statement by condition `if(`$\Gamma(s)$`)`. In Figure 3, the statement $v_m$ `= true` in the `filled` message server is an example of this transition.

*Rule 3* denotes sending message $m$ to actor $a_i$. This fact is modeled using send statements in Rebeca in form of $\texttt{a}_i$`.m();`. The application condition $\Gamma(s)$ is modeled by guarding the send statement by condition `if(`$\Gamma(s)$`)`. In Figure 3, the statement `f.fill();` in the `addMilk` message server is an example of sending message `fill` to the rebec `f`.

*Rule 4* denotes skipping a statement when its application condition does not hold. We explained modeling this rule in Section 5.2.2.

*Rule 5,6* denote taking a message from the queue and executing the corresponding message server when the application conditions of the message server and corresponding actor hold and otherwise dropping the message. We described the representations of these rules in Rebeca in Section 5.2.1. Recall that we do not have to update the local copy of the configuration vector as in Rule 5 as the message servers in Rebeca execute atomically.

## 5.3 Coordinating Featured Actors

In actor families, it is preferable to ensure that the application conditions of $m$ and $r$ hold in the current configuration, before sending a message $m$ to a rebec $r$. This check can be performed by putting an `if` statement before each send statement: `if(`$\psi_r \wedge \psi_m$`) a.m()`. Although this solution prevents sending messages that would not be served, it may cause potential deadlocks. For example, consider the behavior of the `nextOrder` message server of the `Controller` which intends to send a drink request non-deterministically to the corresponding rebec. One may model such behavior as:

```
msgsrv nextOrder() {
 order = ?(1,2,3);
 if(order == 1)
  if(v_c) cs.serveCoffee();
 if(order == 2)
  if(v_t) ts.serveTea();
 if(order == 3)
  if(v_w) ws.serveWater();
}
```

In the above implementation, `order` is an integer variable and its value specifies the requested drink. In the first glance, this seems an appropriate implementation for our desired goal. However, it has a serious flaw causing deadlock. Assume that `order = 2` and $v_t$ `= false`. As a result, no message is sent by the `nextOrder` message server and the execution progress stops as all the queues are empty.

In such circumstances, it is reasonable to send an alternative message when one is not available. Especially when modeling product lines containing features with *xor* relationship, we should select one behavior from a set of corresponding alternative behaviors. If a rebec have $k$ alternative messages to send, it should examine all $2^k$ possible cases of inclusion/exclusion of those messages, then select an appropriate message to send considering the current configuration. For the `nextOrder` message server the following implementation solves the problem:

```
msgsrv nextOrder() {
 if(v_c ∧ v_t ∧ v_w) order = ?(1,2,3);
 if(¬v_c ∧ v_t ∧ v_w) order =?(2,3);
 if(v_c ∧ ¬v_t ∧ v_w) order = ?(1,3);
 if(v_c ∧ v_t ∧ ¬v_w) order = ?(1,2);
 if(¬v_c ∧ ¬v_t ∧ v_w) order = ?(3);
 if(¬v_c ∧ v_t ∧ ¬v_w) order = ?(2);
 if(v_c ∧ ¬v_t ∧ ¬v_w) order = ?(1);
 if(order == 1) cs.serveCoffee();
 if(order == 2) ts.serveTea();
 if(order == 3) ws.serveWater();
}
```

Handling such situations is a tedious work to do. To solve this issue elegantly, we add non-deterministic send statements to Rebeca syntax.

### 5.3.1 Non-deterministic Send Statements

A non-deterministic send statement, `send?` $(a_i.m_j(), ..., a_x.m_y())$, is added to Rebeca syntax to simplify modeling coordination among rebecs. This statement evaluates the application conditions of a set of given messages and their corresponding actors, then non-deterministically selects an available message from the set. The information about application conditions of rebecs and message servers are obtainable in the model through $@\psi$ notations. A non-deterministic send statement `send?`$(a_1.m_1(), a_2.m_2())$ can be represented using the original syntax of Rebeca as:

`if`$((\Gamma_A(a_1) \wedge \Gamma_M(m_1)) \wedge (\Gamma_A(a_2) \wedge \Gamma_M(m_2)))$ `choice = ?(1,2);`
`if`$((\Gamma_A(a_1) \wedge \Gamma_M(m_1)) \wedge \neg(\Gamma_A(a_2) \wedge \Gamma_M(m_2)))$ `choice = ?(1);`
`if`$(\neg(\Gamma_A(a_1) \wedge \Gamma_M(m_1)) \wedge (\Gamma_A(a_2) \wedge \Gamma_M(m_2)))$ `choice = ?(2);`

```
if(choice == 1) a1.m1();
if(choice == 2) a2.m2();
```

The above representation can be extended to select a message among more than two messages as well. More complex pattern for sending messages can be resolved using a coordinator rebec to preserve modularity of the model and keeping their behavior simple. This approach is described next.

*The Coffee Machine Example: Non-deterministic Send Statement.* We use a non-deterministic send statement in the `Controller` rebec to model receiving a drink order non-deterministically and send serve request to the appropriate rebec. □

### 5.3.2 Coordinator Rebecs

A coordinator rebec is responsible of managing variability in message passing among rebecs by selecting a number of messages from message set $\mathcal{M}$ considering their availability in the current configuration. Such a reactive class can be either described by the modeler or it can be generated automatically based on a set of predefined policies.

**Definition 12 Policy.** A policy is a four tuple $P = (\mathcal{M}, min, max, \succ)$ where

- $\mathcal{M} = \{m_1, ..., m_k\}$ is a set of message servers with application condition $\psi_{m_i}$ for each $m_i \in \mathcal{M}$
- $min$ and $max$ $(0 \leq min \leq max \leq k)$ are the minimum and maximum number of messages that should be sent
- $\succ \subseteq \mathcal{M} \times \mathcal{M}$ defines priorities between messages □

We define $F_{min}(\psi_1, ..., \psi_k)$ that returns *true* when at least *min* formulas from the set $\{\psi_1, ..., \psi_k\}$ hold, otherwise it returns *false*. A policy $P = (\mathcal{M}, min, max, \succ)$ where $min \leq max$ is feasible with respect to the feature model if $\phi_{\mathcal{F}} \rightarrow F_{min}(\psi_{m_1}, ..., \psi_{m_k})$ is satisfiable. In other words, a policy is feasible if at least *min* messages are available in all possible valid configurations.

Upon receiving a request to send messages, a coordinator rebec sends a set of messages $\mathcal{M}_s$ with the following properties to the corresponding rebecs:

- $\mathcal{M}_s \subseteq \mathcal{M}$
- $min \leq |\mathcal{M}_s| \leq max$
- $m_i \in \mathcal{M}_s \Rightarrow \theta \vDash \psi_{m_i}$
- $m_i \in \mathcal{M}_s \Rightarrow \nexists m_k \in \mathcal{M}_s \cdot (\theta \vDash \psi_{m_k}) \wedge (m_k \succ m_i)]$

In practice, for a policy defined over $\mathcal{M} = \{m_1, ..., m_k\}$, the corresponding code of the coordinator is composed of at most $2^k$ `if` statements. Each `if` statement specifies the messages that should be sent based on its condition on available message servers. This set of `if` statements may be added by the modeler or can be generated automatically.

*The Coffee Machine Example: Coordinator Rebec.* We do not use a coordinator rebec in our model of coffee machine family. The behavior of the `nextOrder` message server could be modeled using a coordinator rebec with policy

$$P = (\{cs.serveCoffee(), ts.serveTea(), ws.serveWater()\}, 1, 1, -)$$

. This policy is feasible because of the *or* relationship among coffee, tea, and water features. However, if we change the minimum number of messages to 2, the policy $P$ would not be feasible as it is not applicable on a valid configuration that only includes the coffee feature. □

#### 5.3.3 Message Losses

Even if we check the availability of rebec $r$ and message server $m$ before sending message $m$ to rebec $r$, the message may be still dropped by $a$. A message is stored in the queue of the receiver rebec. Meanwhile, reconfiguration may be preformed by another rebec leading to a configuration that excludes $m$. Consequently, message $m$ would be dropped from the queue as its application condition does not hold. An alternative solution is to allow a rebec to put an excluded message in the end of its queue (by sending it to itself) instead of dropping it.

## 6 Model Checking Reconfigurable Actor Families

We may use model checker tool of Rebeca to verify actor families against properties. We may also prove that a set of constraints on (re)configurations hold in the model.

### 6.1 Properties on Behavior

We model check actor families against a set of properties expressed in linear temporal logic (LTL) using the direct model checker of Rebeca. An LTL formula over the set of $AP$ of atomic propositions is formed according to the following grammar:

$$\varphi ::= \mathit{true} \mid \mathit{false} \mid p \mid \neg\varphi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \Box\varphi \mid \Diamond\varphi \mid \bigcirc\varphi \mid \varphi_1 \mathrm{U} \varphi_2$$

In the above grammar, $p \in AP$, and $\Box$, $\Diamond$, $\bigcirc$, and U stand for globally, eventually, next, and until operators respectively. The result of verifying a model against property $\varphi$, is either satisfaction of $\varphi$ or a counter-example describing the behavior leading to violation of the property.

*The Coffee Machine Example: Properties.* One possible property of the coffee machine family is $\varphi = \Box(\mathit{milk} \geq \mathit{0})$ which ensures that milk container is filled after getting empty, before adding milk to another coffee order. □

### 6.2 Constraints on (Re)configurations

In the context of families of concurrent and distributed systems, we cannot globally monitor the state of the system and reconfigure the system using predefined rules. Instead, changes to the configuration vector are made by individual rebecs based on their own state variables. However, to ensure that reconfigurations during the execution of an actor model conform to the intended constraints, we verify the model against certain set of properties, defined over feature variables, using model checking tool of Rebeca.

#### 6.2.1 Feature Model Constraint

A constraint that should be always held during reconfiguration is the one implied by the feature model. This constraint is represented by $\phi_{\mathcal{F}}$. We can prove that reconfigurations in a model conform to feature model constraint by model checking the invariant $\varphi = \Box(\phi_{\mathcal{F}})$.

*The Coffee Machine Example: Feature Model Constraint.* To ensure that all obtainable configurations in the model of the coffee machine family are valid according to the feature model, we verify the model against the following property: $\varphi = \Box[(c \rightarrow cm) \wedge (t \rightarrow cm) \wedge (w \rightarrow cm) \wedge (m \rightarrow c) \wedge (cm \rightarrow (c \vee t \vee w))]$ □

#### 6.2.2 Reconfiguration Rules

We may also define transformation rules among different configurations using a reconfiguration graph [Damiani and Schaefer(2011)]. This graph indicates possible configurations that may be achieved from a specific configuration via reconfiguration. In such a graph, each node corresponds to a valid configuration and each edge represents including or excluding a feature. An edge $\theta \xrightarrow{+f} \theta'$ denotes that configuration $\theta$ may be reconfigured to $\theta'$ by including feature $f$. Similarly, edge $\theta \xrightarrow{-f} \theta'$ represents reconfiguring $\theta$ to $\theta'$ by excluding feature $f$.

We may investigate conformance of the model to these rules using a set of properties. Consider a reconfiguration graph in which configuration $\theta_i$ has $k$ outgoing edges to configurations $\theta_{i_1}, ..., \theta_{i_k}$. By verifying the following property, we ensure that when the model of actor family has configuration $\theta_i$, it will be reconfigured to one of the configurations $\theta_{i_1}, ..., \theta_{i_k}$ in its next state.

$$\varphi_i = \Box(\theta_i \rightarrow \bigcirc(\theta_i \vee \theta_{i_1} \vee ... \vee \theta_{i_k}))$$

Note that a message server may change more than one element of the configuration vector using a sequence of statements. This may lead to invalidation of the configuration vector before all the changes are applied. However, atomic execution of message servers in Rebeca prevents the violation of the property caused by transient configurations.

*The Coffee Machine Example: Reconfiguration Rules.* Consider a reconfiguration graph for the coffee machine family with two nodes $\theta_1 = \langle ?, \mathit{true}, ?, ?, \mathit{true}\rangle$ and $\theta_2 = \langle ?, \mathit{true}, ?, ?, \mathit{false}\rangle$ and two edges $\theta_1 \xrightarrow{-m} \theta_2$ and $\theta_2 \xrightarrow{+m} \theta_1$. This graph depicts that we may exclude the milk feature and include it again in reconfigurations. The following properties are used to check if the model conforms to such rules:

$$\varphi_1 = \Box[(v_c \wedge v_m) \rightarrow \bigcirc((v_c \wedge v_m) \vee (v_c \wedge \neg v_m))]$$
$$\varphi_2 = \Box[(v_c \wedge \neg v_m) \rightarrow \bigcirc((v_c \wedge \neg v_m) \vee (v_c \wedge v_m))]$$

The first property ensures that starting from a configuration including milk and coffee, the model may either reconfigured to a configuration that excludes milk or keeps its current configuration. The second property, checks reconfiguring a model with a configuration that excludes milk, to a configuration including the milk feature. □

## 7 Evaluation

In this section, we discuss the overheads of our proposed methodology. To show the applicability of our approach, we present the results of applying it to model and verify a set of case studies. Moreover, we show the impacts of the overheads by employing alternative approaches in modeling actor families.

### 7.1 Overheads of the Proposed Methodology

The main problem of model checking is its high computational and memory costs as a result of state space explosion. This fact limits the applicability of model checking technique for large systems.

To support reconfiguration, we add feature variables to Rebeca models which increases the number and size of states. An alternative approach is to omit feature variables and use a variability-aware model checker like [Classen et al.(2010)]. In [Classen et al.(2010)], the model checker uses application conditions to mark each state by the feature combinations that a state is reachable in them. However, it does not support reconfiguring model during execution, and verification of properties on reconfigurations. In our approach, we may verify dynamic product lines as well as static ones with the cost of generating a larger state space.

To handle complex patterns of sending messages to different rebecs based on their availability in the current configuration, we proposed using coordinator rebecs. The main advantages of using coordinator rebecs are preserving modularity of the model and keeping the behaviors of rebecs simple. However, they may lead to larger state space because of their message passing overhead: instead of sending messages directly to the corresponding rebecs, a rebec sends a message to the coordinator. Then, the coordinator manages to send messages based on the current configuration and predefined policies.

### 7.2 Experimental Results

We applied our approach to model and verify the following set of case studies.

- **Cash desk** [Dovland et al.(2005)]: A controller manages to get the total price of a set of products. In addition, it handles payment of the purchased products. The price of products can be read using a scanner or entered by a keyboard. The payment can be done using card or cash. Variability in message passing can be managed using a non-deterministic send statement when getting prices of the purchased items and in the payment step. The model may be reconfigured to disable payment through card. We verify the model to ensure that always the payment is greater or equal than the total price.

- **Elevator**: An elevator serves five floors. It may me equipped with a weight sensor. It may also support a VIP floor which has higher priority than other floors. However, when there are many requests form different floors, the VIP feature is disabled. We verify the model against deadlock.

- **Vending machine** [Fantechi and Gnesi(2008)]: A vending machine receives different requests to serve coffee, tea, or water. Moreover, it is possible to add extra milk and sugar to coffee and tea. A request is sent to the machine based on the available products non-deterministically using a non-deterministic send statement. When the milk container is empty, the model is reconfigured such that it excludes the milk feature. We verify the model against deadlock.

Table 1: The number of states of verifying the individual products, the entire product family, and product family with coordinator, against a behavioral property ($\varphi$), the feature model constraint ($\phi_{\mathcal{F}}$), and a reconfiguration rule ($\varphi_r$).

| | Individual Products | Product Family | Coordinator |
|---|---|---|---|
| | Total States | States | States |
| Cash Desk: $\varphi$ | 470,280 | 437,560 | 472,496 |
| Cash Desk: $\phi_{\mathcal{F}}$ | 470,280 | 437,560 | 472,496 |
| Cash Desk: $\varphi_r$ | 971,381 | 898,000 | 978,920 |
| Elevator: $\varphi$ | 567,604 | 567,580 | - |
| Elevator: $\phi_{\mathcal{F}}$ | 567,604 | 567,580 | - |
| Elevator: $\varphi_r$ | 1,041,341 | 920,530 | - |
| Vending Machine: $\varphi$ | 12,800 | 2,088 | 2,520 |
| Vending Machine: $\phi_{\mathcal{F}}$ | 12,800 | 2,088 | 2,520 |
| Vending Machine: $\varphi_r$ | 20,520 | 3,960 | 4,680 |
| Mine Pump: $\varphi$ | 15,629 | 5,992 | 5,261 |
| Mine Pump: $\phi_{\mathcal{F}}$ | 8,664 | 3,201 | 3,400 |
| Mine Pump: $\varphi_r$ | 21,279 | 7,898 | 8,214 |

– **Mine pump system** [Kramer(1983), Classen et al.(2010)]: A controller is responsible to turn the pump on or off based on the data it receives form different sensors. A sensor measures the water level. The controller should turn the pump off when the water level is low. Two other sensors are optional and they measure the level of CH4 and CO gases in the environment. The controller should turn the pump off if the level of each of these gases is high. The update message should be sent to the available sensors to get the most recent data. The sensors for measuring CH4 and CO level may be disabled and enabled again. We verify the model to ensure that when the water level is low, the pump will be turned off.

To handle variability in message passing, we model each case study with and without using a coordinator actor to show its overhead. Moreover, we verify the models to ensure that their behavioral properties hold and the model conforms to the feature model constraints and the reconfiguration rules. We present the number of states of verifying individual products and the entire product family to show the efficiency of our approach in modeling and verification of entire actor family (instead of modeling and model checking individual products separately).

According to Table 1, verifying actor family entirely leads to fewer states comparing to model checking each actor model individually. The difference between the number of generated states is considerable in the case of vending machine and mine pump controller. The reason is that the products of these two case studies have more commonalities in their behavior that leads to more common states. Handling message passing among actors using coordinator increases the number of states because of its message passing overhead. However, the increment is not significant. Thus, we can take benefit from modular modeling using coordinator with a low cost.

## 8 Conclusion

In this paper, we introduced the concept of actor family along with its semantics as a basis to model families of concurrent and distributed systems. We proposed a methodology to model this concept using Rebeca which is an actor-based modeling language. Dynamic software product lines are supported in our proposed approach as we provided facilities for reconfiguration. In addition, we described two approaches to handle message passing among featured actors elegantly. We can verify an actor family model against behavioral properties. Furthermore, we defined a set of properties to investigate if all configurations that are obtained when executing the model are consistent with the feature model and conform to the set of predefined reconfiguration rules. The result of applying our approach on a set of case studies showed the effectiveness of modeling and verifying actor family entirely.

## Acknowledgement

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

## References

[Agha(1990)] Agha, G.: “Actors: A model of concurrent computation in distributed systems”; MIT Press, Cambridge, MA, USA; (1990).

[Batory(2005)] Batory, D. S.: “Feature models, grammars, and propositional formulas”; Proceedings of the 9th international conference on Software product lines; SPLC’05; 7–20; 2005.

[Benavides et al.(2010)] Benavides, D., Segura, S., Ruiz-Cortés, A.: “Automated analysis of feature models 20 years later: A literature review”; Inf. Syst.; 35 (2010), 615–636.

[Clarke et al.(2010)] Clarke, D., Muschevici, R., Proenca, J., Schaefer, I., Schlatte, R.: “Variability modelling in the ABS language”; Proceedings of the 9th International Symposium on Formal Methods for Components and Objects,; volume 6957 of FMCO’10; Springer-Verlag, 2010.

[Classen et al.(2010)] 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”; Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering; ICSE ’10; 335–344; ACM, 2010.

[Damiani and Schaefer(2011)] Damiani, F., Schaefer, I.: “Dynamic delta-oriented programming”; Proceedings of the 15th International Software Product Line Conference, Volume 2; SPLC ’11; 34:1–34:8; ACM, 2011.

[Dovland et al.(2005)] Dovland, J., Johnsen, E. B., Owe, O.: "Verification of concurrent objects with asynchronous method calls"; Proceedings of the IEEE International Conference on Software - Science, Technology & Engineering; SWSTE '05; 141–150; IEEE Computer Society, Washington, DC, USA, 2005.

[Emerson(1990)] Emerson, E. A.: "Temporal and modal logic"; Handbook of theoretical computer science; (1990), 995–1072.

[Fantechi and Gnesi(2008)] Fantechi, A., Gnesi, S.: "Formal modeling for product families engineering"; Proceedings of the 2008 12th International Software Product Line Conference; SPLC '08; 193–202; IEEE Computer Society, Washington, DC, USA, 2008.

[Gruler et al.(2008)] Gruler, A., Leucker, M., Scheidemann, K.: "Modeling and model checking software product lines"; Proceedings of the 10th international conference on Formal Methods for Open Object-Based Distributed Systems; FMOODS '08; 113–131; Springer-Verlag, 2008.

[Hallsteinsen et al.(2008)] Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: "Dynamic software product lines"; Computer; 41 (2008), 93–95.

[Jaghoori et al.(2006)] Jaghoori, M., Movaghar, A., Sirjani, M.: "Modere: The model-checking engine of Rebeca"; ACM Symposium on Applied Computing - Software Verification Track; (2006), 1810–1815.

[Kang et al.(1990)] 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 (1990).

[Kästner and Apel(2008)] Kästner, C., Apel, S.: "Integrating compositional and annotative approaches for product line engineering"; Proceedings of the GPCE Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE); University of Passau, 2008.

[Kästner et al.(2008)] Kästner, C., Apel, S., Kuhlemann, M.: "Granularity in software product lines"; Proceedings of the 30th international conference on Software engineering; ICSE '08; 311–320; ACM, 2008.

[Kramer(1983)] Kramer, J.: "Conic: An integrated approach to distributed computer control systems"; IEE Proceedings; 130 (1983), 1–10.

[Larsen et al.(2007a)] Larsen, K. G., Nyman, U., Wasowski, A.: "Modal I/O automata for interface and product line theories"; Proceedings of the 16th European Symposium on Programming; ESOP'07; 64–79; Springer-Verlag, 2007a.

[Larsen et al.(2007b)] Larsen, K. G., Nyman, U., Wasowski, A.: "Modeling software product lines using color-blind transition systems"; Int. J. Softw. Tools Technol. Transf.; 9 (2007b), 5, 471–487.

[Moskewicz et al.(2001)] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., Malik, S.: "Chaff: engineering an efficient SAT solver"; Proceedings of the 38th annual Design Automation Conference; DAC '01; 530–535; ACM, 2001.

[Muschevici et al.(2010)] Muschevici, R., Clarke, D., Proenca, J.: "Feature Petri nets"; Second Proceedings of the 14th international conference on Software product lines; 99–106; 2010.

[Plath and Ryan(2001)] Plath, M., Ryan, M.: "Feature integration using a feature construct"; Sci. Comput. Program.; 41 (2001), 1, 53–84.

[Pohl et al.(2005)] Pohl, K., Böckle, G., Linden, F. J. v. d.: Software Product Line Engineering: Foundations, Principles and Techniques; Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[Razavi et al.(2011)] Razavi, N., Behjati, R., Sabouri, H., Khamespanah, E., Shali, A., Sirjani, M.: "Sysfier: Actor-based formal verification of systemc"; ACM Trans. Embed. Comput. Syst.; 10 (2011), 2, 19:1–19:35.

[Sabouri and Khosravi(2010)] Sabouri, H., Khosravi, R.: "An effective approach for verifying product lines in presence of variability models"; Second Proceedings of the 14th international conference on Software product lines; 113–120; 2010.

[Sabouri and Khosravi(2011)] Sabouri, H., Khosravi, R.: "Reducing the model checking cost of product lines using static analysis techniques"; Proceedings of FACS11; 2011.

[Schaefer et al.(2010)] Schaefer, I., Bettini, L., Damiani, F., Tanzarella, N.: "Delta-oriented programming of software product lines"; Proceedings of the 14th international conference on Software product lines: going beyond; SPLC'10; 77–91; Springer-Verlag, Berlin, Heidelberg, 2010.

[Sirjani et al.(2004)] Sirjani, M., Movaghar, A., Shali, A., de Boer, F.: "Modeling and verification of reactive systems using Rebeca"; Fundamenta Informaticae; 63 (2004), 4, 385–410.