

# Modelling and Analysing Variability in Product Families: Model Checking of Modal Transition Systems with Variability Constraints

Maurice H. ter Beek<sup>a,\*</sup>, Alessandro Fantechi<sup>a,b</sup>, Stefania Gnesi<sup>a</sup>,  
Franco Mazzanti<sup>a</sup>

<sup>a</sup>*Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo", CNR  
Via G. Moruzzi 1, 56124 Pisa, Italy*

<sup>b</sup>*Dipartimento di Sistemi e Informatica, Università di Firenze  
Via S. Marta 3, 50139 Firenze, Italy*

---

## Abstract

We present the formal underpinnings of a modelling and analysis framework for the specification and verification of variability in product families. We address variability at the behavioural level by modelling the family behaviour by means of a Modal Transition System (MTS) with an associated set of variability constraints expressed over action labels. An MTS is a Labelled Transition System (LTS) which distinguishes between optional and mandatory transitions. Steered by the variability constraints, the inclusion or exclusion of labelled transitions in an LTS refining the MTS determines the family's possible product behaviour. We formalise this as a special-purpose refinement relation for MTSs, which differs fundamentally from the classical one, and show how to use it for the definition and derivation of valid product behaviour starting from product family behaviour. We also present a variability-aware action-based branching-time modal temporal logic to express properties over MTSs, and demonstrate a number of results regarding the preservation of logical properties from family to product behaviour. These results pave the way for the more efficient family-based analyses of MTSs, limiting the need for product-by-product analyses of LTSs. Finally, we define a high-level modal process algebra for the specification of MTSs. The complete framework is implemented in a model-checking tool: given the behaviour of a product family modelled as an MTS with an additional set of variability constraints, it allows the explicit generation of valid product behaviour as well as the efficient on-the-fly verification of logical properties over family and product behaviour alike.

*Keywords:* Model checking, Modal transition systems, Temporal logic,

---

\*Corresponding author

*Email addresses:* [maurice.terbeek@isti.cnr.it](mailto:maurice.terbeek@isti.cnr.it) (Maurice H. ter Beek),  
[alessandro.fantechi@unifi.it](mailto:alessandro.fantechi@unifi.it) (Alessandro Fantechi), [stefania.gnesi@isti.cnr.it](mailto:stefania.gnesi@isti.cnr.it)  
(Stefania Gnesi), [franco.mazzanti@isti.cnr.it](mailto:franco.mazzanti@isti.cnr.it) (Franco Mazzanti)

## 1. Introduction

Software Product Line Engineering (SPLE) [31, 61] is by now a full-fledged software engineering approach aimed at developing, in a cost-effective manner, a family of software-intensive systems by systematic reuse. Individual products share an overall reference model or architecture of the product family, but they differ with respect to specific features. A feature is a (user-visible) increment in functionality of a product. SPLE reduces time-to-market, increases product quality, and lowers production costs. The common and variable parts of products are defined in terms of features and managing variability is about identifying the variation in a shared family model to encode exactly which combinations of features constitute valid products. The actual configuration of products during application engineering is then reduced to selecting desired options in the variability model. Hence, the overall production process is organised so as to maximise commonalities and at the same time minimise the cost of variations.

Variability modelling and analysis of software-intensive systems traditionally focusses on structural rather than behavioural properties and constraints. It is important to model and analyse their variability also at the behavioural level, in order to provide a form of quality assurance. This was first perceived in the context of UML 2.0 [46, 69]. Consequently, Modal Transition Systems (MTSs) were recognised in [40] as a promising model to describe in a compact way all possible operational behaviour of the products of a product family. In a nutshell, an MTS [1] is an LTS that distinguishes between admissible (‘may’) and necessary (‘must’) transitions. Following [40], variants and extensions of MTSs were studied in order to elaborate a suitable formal modelling structure to describe variability in terms of behaviour, including modal I/O automata [52], variable I/O automata [55], and MTSs with logical variability constraints [4, 5, 36]. This triggered a growing interest in modelling behavioural variability, which led to the application of formal models different from MTSs, but still with a transition system semantics, including first and foremost the highly elaborated framework based on featured transition systems [26, 27], but also process-algebraic approaches [12–14, 42, 45, 67], Petri nets [59], and finite state machines [57]. As a result, behavioural analysis techniques like model checking [6, 23] became available for the verification of (temporal) logic properties of product families, resulting in special-purpose model checkers [15, 16, 25, 32].

In this paper, we focus on one such approach. We present the full formal underpinnings of a modelling and analysis framework, some of whose aspects were introduced in [4, 5, 15, 16]. It is based on a specific subset of MTSs, whose elements are equipped with an additional set of logical variability constraints expressed over actions. These constraints allow one to capture all common variability notions known from feature models [43, 49, 63], since it is well-known that plain MTSs cannot efficiently (in a compact way) model, e.g., the notions of alternative and mutually exclusive features. Considering the may transitions as

optional and the must transitions as mandatory, an MTS can be interpreted as a family of LTSs such that each family member corresponds to a specific selection of optional transitions. In this way, a single MTS can model a product family since it allows a compact representation of the family’s behaviour, by means of states and actions, shared by all products, and variation points, by means of may and must transitions, used to differentiate among products. A specific selection of labelled transitions respecting the variability constraints expressed over action labels, determines possible product behaviour (modelled as LTSs). In this paper, we formalise this for the first time as a special-purpose refinement relation for MTSs, which differs fundamentally from classical MTS refinement, and we show how to use it to formally define and derive valid (i.e., configurable) product behaviour starting from the behaviour of a product family.

Subsequently, we recall v-CTL, a variability-aware action-based branching-time modal temporal logic that was introduced with the sole purpose of reasoning over the syntactic structure of MTSs [15]. To this aim, it provides novel interpretations of some classical modal and temporal operators. In this paper, we demonstrate a number of results regarding the preservation of specific v-CTL properties from family to product behaviour. These results pave the way for family-based analyses of MTSs, limiting the need for enumerative product-by-product analyses of LTSs. Based on model-checking techniques for v-CTL, we updated the Variability Model Checker VMC [15, 16] in such a way that it now allows one to perform two kinds of behavioural variability analysis on a product family modelled as an MTS with additional variability constraints:

1. The actual set of valid product behaviour can explicitly be generated from the MTS, after which the MTS and the resulting LTSs can independently be verified against a logical property;
2. A logical property can be verified directly over the MTS, relying on the fact that under certain syntactic conditions, validity over the MTS guarantees validity of the same property for all the family’s valid products (LTSs).

Finally, we formally define the modal process algebra that VMC accepts as specification of an MTS and we illustrate the applicability of the overall framework and its associated tool by means of an example family of coffee machines.

The paper is based on previous publications (in particular, [4, 5, 15, 16]), but it contains new material and results. The formal definition of refinement of MTSs has been completely revisited for the specific purpose of defining and deriving LTSs modelling valid product behaviour, which has required the introduction of new notions like consistent and valid products. The preservation by refinement of v-CTL formulae has been formally defined and proved. The full syntax and semantics of the high-level modal process algebra used to specify MTSs in VMC has been defined and the associated set of variability constraints may now contain any Boolean expression over the action labels.

The paper is organised as follows. After presenting a running product family example in Section 2, we introduce the modelling framework of MTSs with additional variability constraints in Section 3. In Section 4, we define the action-based branching-time modal temporal logic, v-CTL, for the formulation and

analysis of behavioural variability in MTSs. The associated model-checking tool, VMC, for the specification and verification of behavioural variability in product families modelled as MTSs is described in Section 5. Section 6 discusses related models and tools and, finally, Section 7 contains some concluding remarks.

## 2. Running Example of Variability in Product Families

Throughout the paper, we will use a family of (simplified) coffee machines as running example, variants of which have been used in [4, 5, 8, 12, 14, 16, 17, 19, 27, 28, 36, 37, 42, 59]. It is described by the following list of requirements:

1. Each time a user desires a beverage, initially (s)he must insert a coin: either a euro, exclusively in case of coffee machines for the European market, or a dollar, exclusively in case of coffee machines for the Canadian market;
2. After having inserted a coin, the user has to be offered the option to choose whether or not (s)he wants sugar in her/his beverage, after which (s)he has to be offered to select a beverage;
3. The choice of beverages offered by a coffee machine may vary (the options being cappuccino, coffee, and tea), but every coffee machine must offer at least one beverage. Furthermore:
  - (a) tea may only be offered by coffee machines for the European market;
  - (b) whenever a coffee machine offers cappuccino, then it must offer coffee as well;
  - (c) whenever a coffee machine offers coffee, then it either delivers always espresso or always regular coffee;
4. After the user has chosen her/his beverage, the coffee machine delivers it and, as soon as the user has taken her/his beverage, the coffee machine must return in its idle state.

Note that these requirements contain both structural constraints, defining differences in configuration (in terms of features) among products, and operational constraints, defining the behaviour of products through admitted sequences (temporal orderings) of actions/operations (which implement features). An example of the former is the fact that every coffee machine must offer at least one beverage, but in a Canadian coffee machine this must be either coffee or cappuccino (and coffee), whereas an example of the latter is that a coin must have been inserted before a beverage can be chosen.

To illustrate a number of technical definitions, mainly in the next section, we will use an even more simplistic family of coffee machines that deliver a coffee upon the insertion of either a euro or a dollar.

## 3. Modelling Behavioural Variability in Product Families

Similar to earlier approaches based on variants of MTSs [36, 40, 52, 55], we use MTSs for describing in a compact and abstract way the behaviour of an entire product family and their underlying model of LTSs for describing individual product behaviour.

**Definition 1** (LTS). A Labelled Transition System (LTS) is a tuple  $(Q, \Sigma, \bar{q}, \delta)$ , where  $Q$  is a finite and non-empty set of states,  $\Sigma$  is a finite set of actions,  $\bar{q} \in Q$  is an initial state, and  $\delta \subseteq Q \times \Sigma \times Q$  is a transition relation. We sometimes write  $q \xrightarrow{a} q'$  if  $(q, a, q') \in \delta$  and we call it an  $a$ -transition (from source  $q$  to target  $q'$ ).

**Definition 2** ((full) path). Let  $\mathcal{L} = (Q, \Sigma, \bar{q}, \delta)$  be an LTS and let  $q \in Q$ . Then,  $\sigma$  is a path from  $q$  if  $\sigma = q$  (empty path) or  $\sigma = q_1 a_1 q_2 a_2 q_3 \dots$  with  $q_1 = q$  and  $q_i \xrightarrow{a_i} q_{i+1}$  for all  $i > 0$  (possibly infinite path); its  $i$ -th state is denoted by  $\sigma(i)$  and its  $i$ -th action is denoted by  $\sigma\{i\}$ .

A state  $q \in Q$  is *reachable* (in  $\mathcal{L}$ ) if there exists a path  $\sigma$  from  $\bar{q}$  to  $q$ , i.e.,  $\sigma(i) = q$  for some  $i > 0$ .

A state  $q \in Q$  is *final* (a.k.a. a sink state) if it is without outgoing transitions.

An action  $a \in \Sigma$  is *reachable* (in  $\mathcal{L}$ ) if there exists a path  $\sigma$  from  $\bar{q}$  such that  $\sigma\{i\} = a$ , for some  $i > 0$ .

A *full path* is a path that cannot be extended any further, i.e., it is infinite or it ends in a final state. The set of all full paths from  $q$  is denoted by  $\text{path}(q)$ .

We also say that a state (action) can be *reached* (occurs) in an LTS if it is reachable.

An MTS is an LTS with a distinction among admissible (*may*) and necessary (*must*) transitions [53].

**Definition 3** (MTS). A Modal Transition System (MTS) is a tuple  $(Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$  such that  $(Q, \Sigma, \bar{q}, \delta^\diamond)$  is an LTS (its underlying LTS) and  $\delta^\square \subseteq \delta^\diamond$ . An MTS distinguishes the *may transition relation*  $\delta^\diamond$ , expressing admissible transitions, and the *must transition relation*  $\delta^\square$ , expressing necessary transitions. We sometimes write  $q \xrightarrow{a}^\diamond q'$  for  $(q, a, q') \in \delta^\diamond$  and  $q \xrightarrow{a}^\square q'$  for  $(q, a, q') \in \delta^\square$  and we sometimes call them *may  $a$ -transition* and *must  $a$ -transition*, respectively.

In the sequel, we will refer to transitions in  $\delta^\diamond \setminus \delta^\square$  as *optional transitions*<sup>1</sup>.

Note that any transition is thus either a must transition or an optional transition (and in any case it is by definition also a may transition). For reasons that will become clear later, we assume that no must  $a$ -transition exists if the set of optional transitions contains an  $a$ -transition. This is the assumption of *coherence*: an action either is optional or is not.

**Definition 4** (must path). Let  $\mathcal{F}$  be an MTS and let  $\sigma = q_1 a_1 q_2 a_2 q_3 \dots$  be a non-empty full path in its underlying LTS. Then,  $\sigma$  is a *must path* (from  $q_1$ ) in  $\mathcal{F}$  if  $q_i \xrightarrow{a_i}^\square q_{i+1}$  for all  $i > 0$ . A *must path*  $\sigma$  is denoted by  $\sigma^\square$ .

Graphically, an MTS is represented as a directed edge-labelled graph, where nodes model states and edges model transitions; in addition, a small arrow indicates its initial state. Solid edges model necessary transitions (i.e.,  $\delta^\square$ ) while dotted edges model optional transitions (i.e.,  $\delta^\diamond \setminus \delta^\square$ ). The edges are labelled

---

<sup>1</sup>In [40], these are called *maybe* transitions.

with actions that are executed as the result of state changes. A sequence of state changes and the executed actions is a path in the graph; it is a must path if all edges involved are solid ones.

**Example 1.** Figure 1(a) depicts an MTS modelling a simplistic family of coffee machines: upon the insertion of either a euro or a dollar, a coffee is delivered. A number of LTSs that can be obtained by step-by-step refining (unfolding) its optional behaviour are depicted in Figs. 1(b)-1(l). In the sequel we will define precisely which of these are considered products of this family of coffee machines.

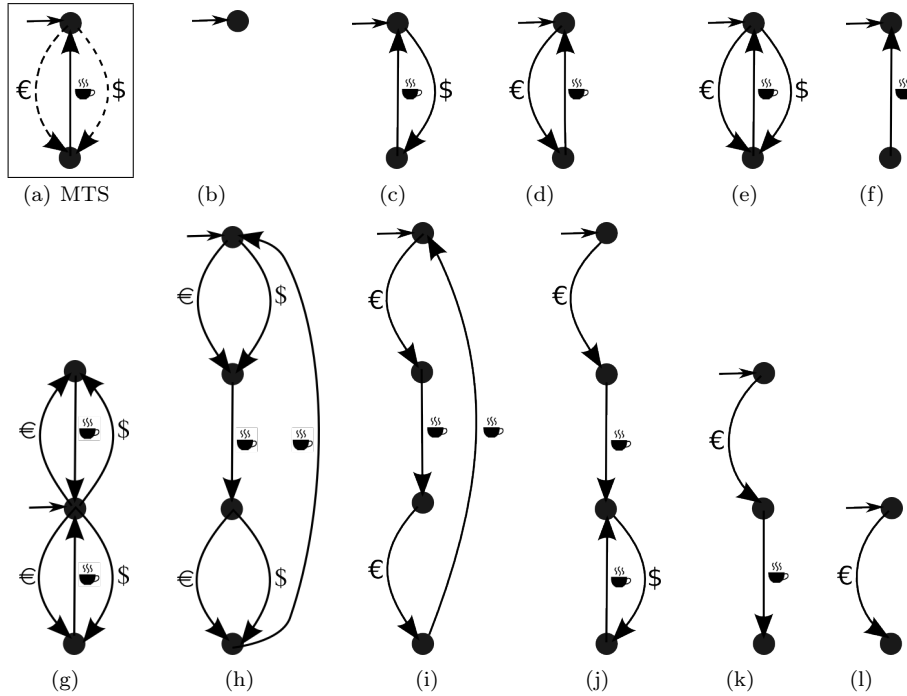


Figure 1: (a)-(l) A family MTS and some potential product LTSs

### 3.1. Resolving Variability by Refinement

An MTS thus describes the possible behaviour of a product family, including variability modelled through optional transitions, i.e., admissible (may), but not necessary (must) transitions (the dotted edges in Fig. 1(a)). The idea is that the family's products, i.e., ordinary LTSs, can be obtained by *resolving* this variability. Resolving variability boils down to deciding for each particular optional behaviour whether or not it is to be included in a specific product. This leads to the need for a notion of conformance that defines whether the product behaviour of an LTS conforms to the required family behaviour described by the MTS or not.

Traditionally, implementations of an MTS are LTSs that capture the idea of refining a partial description into a more detailed one, reflecting increased knowledge on the admissible (but not necessary) behaviour. We know from [40] that the usual (strong and weak refinement) semantics over MTSs are not capable of capturing a notion of conformance that is suitable for SPLE. This is because even if only the observable behaviour is considered, it can still be the case that MTS behaviour is not preserved consistently throughout LTSs, in the sense that the decision to include (i.e., implement) optional transitions can vary from one occurrence to the other (cf. Examples 4 and 5).

In this section, we show how we can nevertheless make use of the classical notions of refinement and implementation among transition systems in our definition of a behavioural conformance for MTSs. This conformance relation should match the SPLE notion that each product of a product family is a refinement of that family, based on the understanding that an LTS (product behaviour) conforms to the MTS (family behaviour).

**Definition 5** (refinement). *An MTS  $\mathcal{F}_p = (Q_p, \Sigma, \bar{q}_p, \delta_p^\diamond, \delta_p^\square)$  is a refinement of an MTS  $\mathcal{F} = (Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$ , denoted by  $\mathcal{F} \preceq \mathcal{F}_p$ , if and only if there exists a refinement relation  $\mathcal{R} \subseteq Q \times Q_p$  such that  $(\bar{q}, \bar{q}_p) \in \mathcal{R}$  and for any  $a \in \Sigma$  and for all  $(q, q_p) \in \mathcal{R}$ , the following holds:*

1. *whenever  $q \xrightarrow{a} \square q'$ , for some  $q' \in Q$ , then there exists a  $q'_p \in Q_p$  such that  $q_p \xrightarrow{a} \square q'_p$  and  $(q', q'_p) \in \mathcal{R}$ , and*
2. *whenever  $q_p \xrightarrow{a} \diamond q'_p$ , for some  $q'_p \in Q_p$ , then there exists a  $q' \in Q$  such that  $q \xrightarrow{a} \diamond q'$  and  $(q', q'_p) \in \mathcal{R}$ .*

Intuitively,  $\mathcal{F}_p$  refines  $\mathcal{F}$  if any must transition of  $\mathcal{F}$  exists in  $\mathcal{F}_p$  and every may transition in  $\mathcal{F}_p$  originates from  $\mathcal{F}$ . Obviously, an MTS in which  $\delta^\diamond \subseteq \delta^\square$  (i.e.,  $\delta^\square = \delta^\diamond$ ) is actually an LTS.

Note that an  $a$ -transition of an MTS that is present in all its refinements is by definition a must transition of the MTS. Hence, an  $a$ -transition that is present in some, but not all, of its refinements is a may transition. Obviously, it cannot be the case that an  $a$ -transition (be it may or must) is not present in any of its refinements.

Since all must transitions are preserved by refinement, so are the must paths. The symmetric, for paths consisting of only may transitions, is also true.

**Proposition 1.** *Let  $\mathcal{F}$  and  $\mathcal{F}_p$  be MTSs such that  $\mathcal{F} \preceq \mathcal{F}_p$ . Then:*

1. *If  $\sigma = q_1 a_1 q_2 a_2 q_3 \cdots$  is a must path of  $\mathcal{F}$ , then there exists a must path  $\sigma_p = q_{p1} a_1 q'_{p2} a_2 q'_{p3} \cdots$  of  $\mathcal{F}_p$  such that  $(q_i, q'_{pi}) \in \mathcal{R}$  for all  $i > 0$ ;*
2. *If  $\sigma_p = q_{p1} a_1 q'_{p2} a_2 q'_{p3} \cdots$  is a path of  $\mathcal{F}_p$ , then there exists a path  $\sigma = q_1 a_1 q_2 a_2 q_3 \cdots$  of  $\mathcal{F}$  such that  $(q_i, q'_{pi}) \in \mathcal{R}$  for all  $i > 0$ .*

Definition 5 thus also defines when an LTS is a refinement of an MTS. LTSs that are refinements of an MTS  $\mathcal{F}$  are called *implementations* of  $\mathcal{F}$ .

**Definition 6** (implementation). *An LTS  $\mathcal{F}_p$  is an implementation of an MTS  $\mathcal{F}$  if and only if  $\mathcal{F} \preceq \mathcal{F}_p$ . We will refer to the refinement relation between an MTS and an LTS as an implementation relation. The set of all implementations of  $\mathcal{F}$  is denoted by  $\mathcal{I}(\mathcal{F})$ .*

Given an MTS  $\mathcal{F}$ , the refinement relation induces a preorder over MTSs with top MTS  $\mathcal{F}$ , and whose bottom elements are the LTSs that are implementations of  $\mathcal{F}$ .

**Example 2** (Example 1 continued). *The LTSs depicted in Figs. 1(b)-1(k) are implementations of the MTS in Fig. 1(a). The LTS in Fig. 1(l) is not, because it does not contain the only must transition of the MTS even though its source state is reachable.*

The implementation relation allows LTSs with unreachable states (as an effect of pruning *may* transitions). Since, also for verification purposes, we are interested only in the reachable states representing effectively possible behaviour, we restrict our attention to implementations without unreachable states. Moreover, the number of implementations is in general infinite, because we can apply any number of unfoldings (and even duplications) similar to the ones that resulted in the LTS in Figs. 1(g)-1(i). In fact, in [36, 40] it was noted that refinement need not preserve an MTS's branching structure.

In the SPLE context, instead, we believe it useful to have a simpler notion of refinement which always preserves the original branching structure (up to the removal of unreachable states) of the MTS in the products, corresponding to the modellers' intuition, and which gives to the implementations the choice of just turning dotted edges into solid edges or removing them altogether. Hence, before removing unreachable states, any implementation has exactly the same states as the MTS. As we will see shortly, this stricter notion of refinement has the immediate advantage of leading to a limited set of product behaviour. For our purposes, the minimal products thus suffice, so we will consider the minimal of all possible implementations according to *strong bisimulation equivalence* [58].

**Definition 7** (bisimulation). *Let  $\mathcal{L}_1 = (Q_1, \Sigma_1, \bar{q}_1, \delta_1)$  and  $\mathcal{L}_2 = (Q_2, \Sigma_2, \bar{q}_2, \delta_2)$  be two LTSs. We say that  $\mathcal{L}_1$  and  $\mathcal{L}_2$  are (strongly) bisimilar, denoted by  $\mathcal{L}_1 \sim \mathcal{L}_2$ , if and only if there exists a (strong) bisimulation equivalence  $\mathcal{B} \subseteq Q_1 \times Q_2$  such that  $(\bar{q}_1, \bar{q}_2) \in \mathcal{B}$  and for any  $a \in \Sigma$  and for all  $(q_1, q_2) \in \mathcal{B}$ , the following holds:*

1. *whenever  $q_1 \xrightarrow{a} q'_1$ , for some  $q'_1 \in Q_1$ , then  $\exists q'_2 \in Q_2$  such that  $q_2 \xrightarrow{a} q'_2$  and  $(q'_1, q'_2) \in \mathcal{B}$ , and*
2. *whenever  $q_2 \xrightarrow{a} q'_2$ , for some  $q'_2 \in Q_2$ , then  $\exists q'_1 \in Q_1$  such that  $q_1 \xrightarrow{a} q'_1$  and  $(q'_1, q'_2) \in \mathcal{B}$ .*

**Definition 8** (product). *An LTS  $\mathcal{F}_p = (Q_p, \Sigma, \bar{q}_p, \delta_p)$  is a product of an MTS  $\mathcal{F}$  if and only if  $\mathcal{F}_p$  is the minimal (w.r.t. the number of states) element of one of the classes of equivalences induced by strong bisimulation over  $\mathcal{I}(\mathcal{F})$ . The set of all products of  $\mathcal{F}$  is denoted by  $\mathcal{I}_p(\mathcal{F})$ .*



This definition cuts out the LTSs with unreachable states and with useless unfoldings and duplications.

**Theorem 1.** *If  $\mathcal{L}$  is the minimal element of a class of bisimulation equivalent LTSs, then all its states are reachable.*

*Proof.* Assume that  $\mathcal{L}$  is minimal and that it has an unreachable state  $q$ . Consider the LTS  $\mathcal{L}'$  obtained from  $\mathcal{L}$  by recursively navigating all its transitions. Then  $\mathcal{L}'$  is bisimulation equivalent to  $\mathcal{L}$  by construction, but by definition it contains no state corresponding to  $q$ . Hence  $\mathcal{L}$  is not minimal.  $\square$

**Example 3** (Example 1 continued). *The LTSs depicted in Figs. 1(b)-1(e), 1(j), and 1(k) are thus products of the MTS in Fig. 1(a). The LTS in Fig. 1(f) is not, because it has more elements than the bisimilar LTS in Fig. 1(b). Likewise the LTSs in Figs. 1(g) and 1(h), which have more elements than the bisimilar LTS in Fig. 1(e), are not, and neither is the LTS in Fig. 1(i), which is bisimilar to the smaller LTS in Fig. 1(d).*

We adopt the same (often implicit) assumption that underlies the behavioural variability models based on MTSs [4, 5, 36, 40], I/O automata [52, 55], LTSs [45], featured transition systems [26–29], and mCRL2 [17] mentioned in Section 1, viz. an action models a piece of functionality (a *feature*, if you like). This provides motivation for the assumption of *coherence* put forward in the beginning of Sect. 3, reflecting the fact that a functionality (or, a feature) either *is* optional or is not. Now, recall that an MTS models the variability of a product family through its optional transitions, which need to be resolved to obtain the family’s products. This means that it must be decided whether or not an optional transition of an MTS should occur in an LTS modelling a product.

For our application in SPLE, we need variability to be resolved in a *consistent* way, i.e., the choice of ‘implementing’ or ‘not implementing’ a transition in a product must be *consistent* throughout the product. This is because it reflects the fact that a functionality (or a feature) either is or is not present in a product, independently of its behavioural context. Informally, this notion of consistency states the following. Given an MTS and an LTS modelling one of its products, we want that for all actions  $a$ , either all or none of the  $a$ -transitions in the MTS occur (i.e., are ‘implemented’) as must  $a$ -transitions in its product. Two illustrative examples follow the formal definition.

**Definition 9** (consistent product). *Let  $\mathcal{F} = (Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$  be an MTS and  $\mathcal{F}_p = (Q_p, \Sigma, \bar{q}_p, \delta_p)$  be a product of  $\mathcal{F}$ , i.e.,  $\mathcal{F}_p \in \mathcal{I}_p(\mathcal{F})$ , through the implementation relation  $\mathcal{R} \subseteq Q \times Q_p$ . Then,  $\mathcal{F}_p$  is said to be consistent (with  $\mathcal{F}$ ) if and only if for all  $a \in \Sigma$ , the following holds:*

- *if there exist a  $(q, a, q') \in \delta^\diamond$  and a  $(q_p, a, q'_p) \in \delta_p$ , with  $(q, q_p) \in \mathcal{R}$  and  $(q', q'_p) \in \mathcal{R}$ , then for all  $(r, a, r') \in \delta^\diamond$  for which there exists an  $r_p \in Q_p$  such that  $(r, r_p) \in \mathcal{R}$ , there must exist an  $r'_p \in Q_p$  such that  $(r_p, a, r'_p) \in \delta_p$  and  $(r', r'_p) \in \mathcal{R}$ .*

*The set of all consistent products of  $\mathcal{F}$  is denoted by  $\mathcal{I}_{cp}(\mathcal{F})$ .*

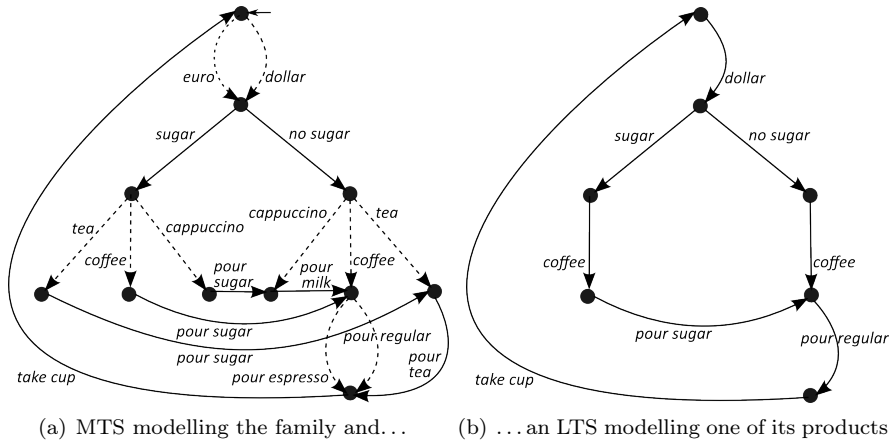


Figure 2: (a)-(b) Modelling the family of coffee machines and a (Canadian) coffee machine

**Example 4** (Example 1 continued). *The LTSs depicted in Figs. 1(j) and 1(k) are not consistent products of the MTS in Fig. 1(a). This can be seen as follows. First recall that the consistency assumption states that whenever it is decided to ‘implement’ the  $\text{€}$ -transition in a product, then this has to be done in a consistent way. In this particular example, this means that whenever the  $\text{€}$ -transition is implemented from their initial states, as is currently the case in both products, then it should also be implemented from the states that are reached after this action  $\text{€}$  and the action  $\text{☞}$  have each occurred once, which is however not the case in Figs. 1(j) and 1(k). The LTSs depicted in Figs. 1(b)-1(e) are consistent products. In the next section, we will deal with the (undesired) trivial product in Fig. 1(b) and we will discuss whether the one in Fig. 1(e) is desired.*

A convenient result of the consistency assumption is that it makes the number of products finite. In Example 4, e.g., we have seen that the LTS depicted in Fig. 1(k) is not a consistent product of the MTS in Fig. 1(a) and the same obviously holds for any further unfolding of that MTS.

**Example 5.** *Now consider the MTS depicted in Fig. 2(a). It is an attempt to model all possible (product) behaviour conceived for the example family of coffee machines from Section 2 in a compact way. Note that it has variation points to model the choices among the different types of coin, beverage, and coffee. Moreover, note that the MTS allows each of the beverages to be delivered either with or without sugar. Without the requirement for consistent product behaviour, a product that can deliver coffee only without sugar would be allowed. An example of such an undesired product is depicted in Fig. 3(a): as soon as the user chooses for a sugared beverage, (s)he has less options than if (s)he had chosen for an unsugared beverage. The LTSs in Figs. 2(b) and 3(b) instead model consistent Canadian and European products, respectively.*

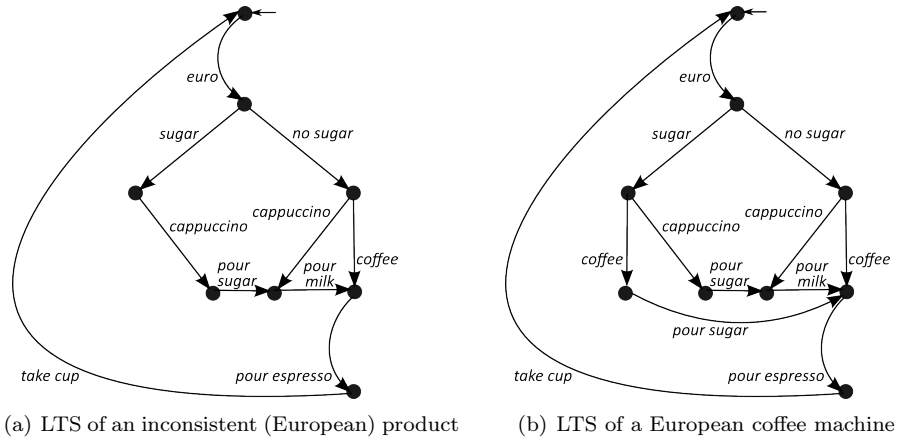


Figure 3: (a)-(b) Modelling products of the family of coffee machines

In [18], consistency is guaranteed in a different way (and called *persistence*). They define so-called *parametric* MTSs, where it is allowed to choose in a consistent (persistent) way whether or not to implement a transition in a product by using parameters with a priori fixed (Boolean) values that settle this choice for the entire product.

### 3.2. Deriving Products from Product Families

The products of a family modelled by an MTS are thus LTSs that are obtained by resolving the variability inherent to MTSs. In Section 3.1, we have seen how to resolve this variability by means of semantic definitions based on refinement that are common in the literature on MTSs. However, for our specific purpose of applying MTSs in SPLE, an operational syntactic definition of derivation exists. It is presented in this section and it is the one actually implemented in our VMC tool that will be discussed in Section 5.

Intuitively, in each LTS modelling a product of an MTS, all must transitions of the MTS are included, together with a subset of its optional transitions. As before (cf. Def. 9), we assume that the choice of including an optional transition needs to be consistent. More formally, a derived product LTS is obtained from an MTS in the following way: the LTS has the same set of actions and the same initial state, but it contains a subset of the set of states and a subset of the set of transitions such that:

1. all its states are reachable from the initial state;
2. all must transitions of the MTS are included in the LTS, except of course those must transitions whose source states are not reachable in the LTS;
3. for any action  $a$ , whenever an  $a$ -transition of the MTS is included in the LTS, then any other (optional)  $a$ -transition in the MTS (from a state that is reachable in the LTS) is also included.

This operational derivation procedure is formalised in the following definition.

**Definition 10** (derived product LTS). *Let  $\mathcal{F} = (Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$  be an MTS. Then, the set  $\{\mathcal{P}_i = (Q_i, \Sigma, \bar{q}, \delta_i) \mid i > 0\}$  of derived product LTSs of  $\mathcal{F}$ , denoted by  $\mathcal{P}_{\mathcal{F}}$ , is obtained from  $\mathcal{F}$  by considering each pair  $(Q_i, \delta_i)$ , with  $Q_i \subseteq Q$  and  $\delta_i \subseteq \delta^\diamond$ , such that the following holds:*

1. every  $q \in Q_i$  is reachable in  $\mathcal{P}_i$ ;
2. there exists no  $(q, a, q') \in \delta^\square \setminus \delta_i$  such that  $q \in Q_i$ ;
3. for any  $a \in \Sigma$ , whenever  $(p, a, p') \in \delta^\diamond \cap \delta_i$ , then for all  $q \in Q_i$  such that  $(q, a, q') \in \delta^\diamond$  it must be the case that also  $(q, a, q') \in \delta_i$ .

The following example illustrates the reason for the first two requirements in Def. 10; the need for the third requirement is to guarantee consistency.

**Example 6** (Example 1 continued). *The set of derived product LTSs of the MTS in Fig. 1(a) that is generated by Def. 10 consists of the LTSs depicted in Fig. 1(b)-1(e). Note that the LTS in Fig. 1(f) is not a derived product LTS of this MTS because it contains an unreachable state, while none of the LTSs in Figs. 1(g)-1(k) can of course be a derived product LTSs of the MTS since they contain more states than the MTS. The LTS in Fig. 1(l), finally, is not a derived product LTS because it does not contain the only must transition of the MTS even though its source state is reachable.*

We now show that Def. 10 is an alternative for the definition of consistent products (cf. Def. 9).

**Theorem 2.** *Let  $\mathcal{F}$  be an MTS. The set of all derived product LTSs of  $\mathcal{F}$  equals, modulo bisimulation, the set of consistent products of  $\mathcal{F}$ :  $\mathcal{P}_{\mathcal{F}} \simeq \mathcal{I}_{cp}(\mathcal{F})$  (i.e., every derived product LTS is the representative of an equivalence class whose minimal element is a consistent product and, conversely, every consistent product is equivalent modulo bisimulation to a derived product).*

*Proof.* Let  $\mathcal{F} = (Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$  and let  $\mathcal{F}_p = (Q_p, \Sigma, \bar{q}_p, \delta_p)$  be a derived product LTS of  $\mathcal{F}$ .

We start by proving that  $\mathcal{F}_p$  is an implementation of  $\mathcal{F}$ , i.e., we prove that  $\mathcal{F}_p \in \mathcal{I}(\mathcal{F})$ . From Def. 10 it follows directly that (i) all states that are maintained in  $\mathcal{F}_p$  maintain also their outgoing must transitions and (ii)  $\delta_p$  may include further transitions that originally were optional in  $\mathcal{F}$ . These two statements correspond to the definition of refinement, which means, since  $\mathcal{F}_p$  is an LTS, that we can now prove that  $\mathcal{F}_p$  is indeed an implementation. First, since a product of an MTS is the minimal element of the equivalence class induced by bisimulation over  $\mathcal{I}(\mathcal{F})$ ,  $\mathcal{F}_p$  is bisimilar to a product of  $\mathcal{F}$ . Subsequently, requirement 3 of Def. 10 guarantees the consistency of  $\mathcal{F}_p$ .

On the other hand, suppose that we have a consistent product  $\mathcal{F}_p$  of  $\mathcal{F}$  that is not bisimilar to any derived product LTS of  $\mathcal{F}$ . This means that the class of the partition over  $\mathcal{I}(\mathcal{F})$ , induced by bisimulation equivalence, that contains  $\mathcal{F}_p$ , contains no derived product LTSs. The implementations of  $\mathcal{F}$ , due to the refinement relation, maintain all must transitions and a subset of the optional transitions. By the consistency requirement, if an optional  $a$ -transition is not

included in a product, neither are all other  $a$ -transitions. Hence, an equivalence class made of consistent implementations can be uniquely characterised by a subset of the optional transitions of  $\mathcal{F}$ . Moreover, an equivalence class made of derived products can be uniquely characterised by its subset of preserved (included) optional transitions of  $\mathcal{F}$ , which could not have been chosen in the derivation process. However, since all pairs of subsets  $Q_i \subseteq Q$  and  $\delta_i \subseteq \delta^\square \cup \delta^\diamond$ , respecting consistency, are considered in a derivation, it cannot be the case that the above subset of preserved optional transitions is not chosen in a derivation. Hence, every consistent product is bisimilar to an implementation.  $\square$

From Def. 10, it becomes immediately clear that an MTS has at most  $2^n$  derived product LTSs, where  $n$  is the number of differently labelled optional transitions in the MTS (because each such set of transitions in  $\delta^\diamond \setminus \delta^\square$  labelled with the same action is either absent or present in a product LTS).

### 3.3. Modelling Additional Variability Constraints

We have just seen that the MTS depicted in Fig. 2(a) has several variation points to model the choices among the different types of coin, beverage, and coffee prescribed by the requirements for a family of coffee machines in Section 2. However, also its underlying LTS (obtained by turning all optional transitions into must transitions) is a consistent product, even though it satisfies neither requirement 1 nor requirements 3(a-c): a user may alternate the insertion of euros and dollars, obtain a tea upon the insertion of a dollar, etc. This is inherent to the use of MTSs for capturing the behaviour of product families: while capable of faithfully modelling some degree of optional or mandatory behaviour that may or must, respectively, be preserved in its products (LTSs), an MTS cannot efficiently model the constraints regarding *alternative* (neither ‘or’ nor ‘xor’) behaviour nor regarding the so-called *excludes* and *requires* cross-tree constraints known from feature models [4, 43, 49]. Such constraints can express the fact that the presence of a feature (mutually) excludes or, on the contrary, requires the presence of another feature. Their formal definitions follow shortly. The reason that plain MTSs cannot efficiently model these constraints in a compact way resides in the fact that the decision whether to preserve an optional transition of an MTS in a product (LTS) or not, can be made independently of the decision whether or not to preserve another optional transition of the MTS, as is illustrated in the following example.

**Example 7** (Example 1 continued). *The informal description of the simplistic family of coffee machines modelled by the MTS in Fig. 2(a) (“upon the insertion of either a euro or a dollar, a coffee is delivered”) is of course rather vague. The coffee machine modelled by the LTS in Fig. 1(b) is arguably not an intended product of the family, but what about the products that allow to insert only dollars (Fig. 1(c)), only euros (Fig. 1(d)), or to interchange the insertion of both (Fig. 1(e)): are these to be considered valid products? Most real-world coffee machines deliver coffee exclusively upon the insertion of one type of coin, e.g., euros in Europe and dollars in Canada. Note that a simple constraint requiring*

product LTSs to contain either the \$-transition or the €-transition would suffice (i.e., only the LTSs in Figs. 1(c) and 1(d) would model valid products).

A single MTS, possibly with a new initial state, could of course join the LTSs depicted in Figs. 1(c) and 1(d), at the cost of duplicating the ☛-transition, but this would be highly inefficient and not at all compact in case of a family that should accommodate separate products for numerous different coins (e.g., \$, €, £, ¥) and in case of more rich product behaviour (e.g., coffee, tea, cappuccino) that is for a large part shared among the various products (e.g., sugar, take cup) but with subtle variations (e.g., pour milk only in case of cappuccino).

In [4, 5] we presented a preliminary solution to cope with this incapacity of covering all common variability constraints, which we now extend significantly<sup>2</sup>. We associate with an MTS modelling a product family a set of *variability constraints* that must be taken into account when deriving the set of product LTSs. Recall that an action is *reachable* in an LTS if there is a path in that LTS (i.e., starting from its initial state) on which it is executed (occurs).

**Definition 11** (variability constraints). *Let  $\mathcal{L} = (Q, \Sigma, \bar{q}, \delta)$  be an LTS and let  $a_i \in \Sigma$  for all  $i$ . The following variability constraints on actions from  $\Sigma$  may be defined over  $\mathcal{L}$ , for  $m \geq 2$  and  $n \geq 3$ :*

$a_1$  ALT  $\dots$  ALT  $a_m$  : precisely one among the actions  $a_1, \dots, a_m$  is reachable in  $\mathcal{L}$ ;

$b_1$  OR  $\dots$  OR  $b_m$ , where  $b_i$ , with  $1 \leq i \leq m$ , is either  $a_i$  or  $\neg a_i$  : at least one among the conditions on actions  $a_1, \dots, a_m$  holds, i.e.,  $b_i = a_i$  is reachable in  $\mathcal{L}$  or  $b_i = \neg a_i$  is not reachable in  $\mathcal{L}$ ;

$a_1$  EXC  $a_2$  : at most one of the actions  $a_1$  and  $a_2$  is reachable in  $\mathcal{L}$ ;

$a_1$  REQ  $a_2$  : action  $a_2$  must occur in  $\mathcal{L}$  whenever  $a_1$  is reachable in  $\mathcal{L}$ ;

$a_1$  REQ ( $a_2$  ALT  $\dots$  ALT  $a_n$ ) : precisely one among the actions  $a_2, \dots, a_n$  is reachable in  $\mathcal{L}$  if  $a_1$  is reachable in  $\mathcal{L}$ ;

$a_1$  REQ ( $a_2$  OR  $\dots$  OR  $a_n$ ) : at least one among the actions  $a_2, \dots, a_n$  is reachable in  $\mathcal{L}$  if  $a_1$  occurs in  $\mathcal{L}$ .

Note that ALT can also be expressed in terms of OR and EXC<sup>3</sup>. Moreover, since the most recent extension in [8], implemented in VMC v6.2 (cf. Section 5), allows the ‘OR’ constraint to contain either  $a_i$  (as before) or its negation  $\neg a_i$ , we can express any Boolean function over  $\Sigma$  in its conjunctive normal form. We nevertheless choose to keep the specific variability constraints introduced

<sup>2</sup>In [4, 5] we did not consider the so-called ‘OR’ constraint among actions, nor did we consider any  $n$ -ary constraints or the combined constraints involving ‘REQ’. The possibility to negate some of the action literals in the ‘OR’ constraint is a novelty that we introduced in [8].

<sup>3</sup>For instance,  $a_j$  ALT  $a_k$  can be expressed as  $(a_j$  OR  $a_k)$  and  $(a_j$  EXC  $a_k)$ .

in Def. 11 because they reflect prominent constraint relations in feature modelling [43, 49]. We hope that this eases the uptake of our framework by SPL practitioners.

We now define which consistent products (derived product LTSs) of an MTS with an associated set of variability constraints are to be considered *valid* products of the product family modelled by that MTS.

**Definition 12** (valid product). *Let  $\mathcal{F} = (Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$  be an MTS with a set  $\mathcal{V}$  of variability constraints (on actions from  $\Sigma$ ), and let  $\mathcal{P}_i \in \mathcal{I}_{cp}(\mathcal{F})$  be a consistent product of  $\mathcal{F}$ . Then,  $\mathcal{P}_i$  is a valid product of  $\mathcal{F}$  with variability constraints  $\mathcal{V}$  if and only if  $\mathcal{P}_i$  satisfies all variability constraints in  $\mathcal{V}$ .*

**Example 8** (Example 1 continued). *Given the MTS depicted in Fig. 1(a) extended with the set  $\mathcal{V} = \{\text{€ ALT \$}\}$  of variability constraints, its only valid products are the LTSs depicted in Figs. 1(c) and 1(d).*

**Example 9** (Example 5 continued). *For the family modelled by the MTS in Fig. 2(a), we can now characterise its set of valid products (coffee machines) as precisely those product LTSs that can be derived from the MTS according to Def. 9 and satisfy each of the following variability constraints:*

1. euro ALT dollar;
2. cappuccino OR coffee OR tea;
3. dollar EXC tea;
4. cappuccino REQ coffee;
5. coffee REQ (pour espresso ALT pour regular).

*It is easy to verify that the consistent European product described by the LTS in Fig. 3(b) satisfies each of the above constraints. Hence, it models a valid (European) product. Recall from Example 5 that the one in Fig. 3(a) does not.*

Summarising, we thus propose to complement an MTS with a set of variability constraints in order to obtain a model capable of expressing families of LTSs and with characteristics that make it suitable for SPLE. Given this behavioural (semantic) model, we also need a logic that can be interpreted over it to be able to address the problem of checking properties over product families specified as MTSs (with additional variability constraints). In the next section, we present a suitable logic.

#### 4. v-ACTL: A Logic for Expressing and Analysing Variability

In this section, we first present a logic that allows one to reason over MTSs with variability constraints and which moreover provides a semantics for the latter, thus allowing us to refine the notion of valid products of an MTS with an associated set of variability constraints. Then, we study the preservation of properties from MTSs to their valid products, which results in the identification of two fragments of the logic which are such that all formulae expressed in them, and which hold for an MTS, preserve their validity for all valid products (LTSs).

We introduce *variability-aware* ACTL (v-ACTL) as a logic, interpreted over MTSs, in which several logics that we introduced over the years culminate [4, 5]. The result is an action-based branching-time temporal logic for variability in the style of the action-based logic ACTL [33, 41]. More precisely, next to the standard operators of propositional logic, v-ACTL contains the classical box and — by duality — diamond modal operators, the existential and universal path quantifiers, and the (action-based)  $F$  (‘eventually’) and — by duality —  $G$  (‘globally’) operators. Furthermore, for the box, diamond, and  $F$  operators, v-ACTL also contains ‘boxed’ variants; these can be distinguished in a logic over MTSs by taking into account the specific modality (may or must) of the involved transitions and paths (e.g., requiring a path to consist of must transitions only).

In particular, v-ACTL is a simplification of the logic available in the VMC tool that we will present in Section 5 [15] by leaving out the (weak) Until operators and including directly the  $F$  operator. The reason for limiting ourselves to these two temporal operators in this paper is threefold<sup>4</sup>. First, the resulting logic suffices for specifying the additional variability constraints from Def. 11. Second, it allows a more concise presentation of the forthcoming results on preservation by refinement in Sections 4.2 and 4.3. Third, it suffices for specifying a considerable number of interesting properties for product families in the presence of variability.

v-ACTL defines action formulae (denoted by  $\chi$ ), state formulae (denoted by  $\phi$ ), and path formulae (denoted by  $\pi$ ).

**Definition 13** (syntax of action formulae). *Action formulae are built as follows over a set  $\mathcal{A}$  of atomic actions  $\{a, b, \dots\}$ :*

$$\chi ::= \text{true} \mid a \mid \neg\chi \mid \chi \wedge \chi$$

Action formulae are thus Boolean compositions of actions. As usual, *false* abbreviates  $\neg \text{true}$ ,  $\chi \vee \chi'$  abbreviates  $\neg(\neg\chi \wedge \neg\chi')$ , and  $\chi \implies \chi'$  abbreviates  $\neg\chi \vee \chi'$ .

**Definition 14** (semantics of action formulae). *The satisfaction of an action formula  $\chi$  by an action  $a$ , denoted by  $a \models \chi$ , is defined as follows:*

$$\begin{aligned} a &\models \text{true} \text{ always holds} \\ a &\models b \text{ iff } a = b \\ a &\models \neg\chi \text{ iff } a \not\models \chi \\ a &\models \chi \wedge \chi' \text{ iff } a \models \chi \text{ and } a \models \chi' \end{aligned}$$

---

<sup>4</sup>The extended version of v-ACTL as implemented in VMC contains also the least and greatest fixed-point operators, which provide a semantics for recursion used for “finite looping” and “looping”, respectively, as in [50], as well as no less than eight versions of the Until operator, based on combinations of action-based, weak (a.k.a. unless), and ‘boxed’ Until variants. The latter variants distinguish between may and must versions of the involved transitions and paths, similar to  $[\ ]^\square$ ,  $\langle \rangle^\square$ ,  $F^\square$ , and  $F^\square\{\}$ .



**Definition 15** (syntax of v-CTL). *The syntax of v-CTL is:*

$$\begin{aligned}\phi & ::= \text{true} \mid \neg\phi \mid \phi \wedge \phi \mid [\chi]\phi \mid [\chi]^\square\phi \mid E\pi \mid A\pi \\ \pi & ::= F\phi \mid F^\square\phi \mid F\{\chi\}\phi \mid F^\square\{\chi\}\phi\end{aligned}$$

Intuitively, the specificity of the ‘boxed’ variants of the respective classical modal and temporal operators can be understood as follows:

$[\chi]\phi$ : in all next states reachable by a *may* transition executing an action satisfying  $\chi$ ,  $\phi$  holds;

$[\chi]^\square\phi$ : in all next states reachable by a *must* transition executing an action satisfying  $\chi$ ,  $\phi$  holds;

$F\phi$ : there exists a future state in which  $\phi$  holds;

$F^\square\phi$ : there exists a future state in which  $\phi$  holds and all transitions until that state are must transitions;

$F\{\chi\}\phi$ : there exists a future state, reached by an action satisfying  $\chi$ , in which  $\phi$  holds;

$F^\square\{\chi\}\phi$ : there exists a future state, reached by an action satisfying  $\chi$ , in which  $\phi$  holds and all transitions until that state are must transitions.

Some further operators can be derived as usual. First,  $\langle\chi\rangle\phi$  abbreviates  $\neg[\chi]\neg\phi$ : a next state exists, reachable by a *may* transition executing an action satisfying  $\chi$ , in which  $\phi$  holds. Second,  $\langle\chi\rangle^\square\phi$  abbreviates  $\neg[\chi]^\square\neg\phi$ : a next state exists, reachable by a *must* transition executing an action satisfying  $\chi$ , in which  $\phi$  holds. Third,  $AG\phi$  abbreviates  $\neg EF\neg\phi$ : in all states on all paths,  $\phi$  holds.

The formal semantics of v-CTL is given by MTSs *without* additional variability constraints. Recall from Def. 2 that a full path is a path that cannot be extended any further, i.e. it is infinite or it ends in a final state.

**Definition 16** (semantics of v-CTL). *Let  $(Q, \mathcal{A}, \bar{q}, \delta^\diamond, \delta^\square)$  be an MTS, with  $q \in Q$ , and let  $\sigma$  be a full path. The satisfaction relation  $\models$  of v-CTL is:*

$$\begin{aligned}q & \models \text{true always holds;} \\ q & \models \neg\phi \text{ iff } q \not\models \phi; \\ q & \models \phi \wedge \phi' \text{ iff } q \models \phi \text{ and } q \models \phi'; \\ q & \models [\chi]\phi \text{ iff for all } q' \in Q \text{ such that } q \xrightarrow{a} q' \text{ and } a \models \chi, \text{ we have } q' \models \phi; \\ q & \models [\chi]^\square\phi \text{ iff for all } q' \in Q \text{ such that } q \xrightarrow{a} q' \text{ and } a \models \chi, \text{ we have } q' \models \phi; \\ q & \models E\pi \text{ iff there exists a } \sigma' \in \text{path}(q) \text{ such that } \sigma' \models \pi; \\ q & \models A\pi \text{ iff for all } \sigma' \in \text{path}(q) : \sigma' \models \pi; \\ \sigma & \models F\phi \text{ iff there exists a } j \geq 1 \text{ such that } \sigma(j) \models \phi; \\ \sigma & \models F^\square\phi \text{ iff there exists a } j \geq 1 \text{ such that } \sigma(j) \models \phi \text{ and } \forall 1 \leq i < j : \\ & \quad (\sigma(i), \sigma\{i\}, \sigma(i+1)) \in \delta^\square;\end{aligned}$$

$\sigma \models F\{\chi\} \phi$  iff there exists a  $j \geq 1$  such that  $\sigma\{j\} \models \chi$  and  $\sigma(j+1) \models \phi$ ;  
 $\sigma \models F^\square\{\chi\} \phi$  iff there exists a  $j \geq 1$  such that  $\sigma\{j\} \models \chi$  and  $\sigma(j+1) \models \phi$ ,  
 and for all  $1 \leq i \leq j : (\sigma(i), \sigma\{i\}, \sigma(i+1)) \in \delta^\square$ .

Since its purpose is to reason over the syntactic structure of an MTS, v-ACTL thus interprets some classical modal and temporal operators in two different ways, by explicitly considering the may and must modalities of the transitions and paths of the semantic model of MTSs.

#### 4.1. Analysis in the Presence of Variability

We recall from Section 3.3 that an MTS model of a product family is complemented with a set of variability constraints that an MTS otherwise cannot capture, i.e., the constraints regarding *alternative* (both ‘or’ and ‘xor’) behaviour and regarding the *excludes* and *requires* cross-tree constraints of feature models.

It is now possible to give a concise logical characterisation of all variability constraints introduced in Def. 11 by noting that the ‘reachability of an action  $a$ ’ in a (product) LTS can be expressed by the formula  $EF^\square\{a\} \text{ true}$  (or simply  $EF\{a\} \text{ true}$ , since the interpretation of the ‘boxed’ variant of  $EF$  collapses on the classic interpretation in the case of LTSs). Let  $\{a_i \mid 1 \leq i \leq n\}$  be a set of actions. Then, the variability constraints from Def. 11 can be captured with the following v-ACTL formulae, with  $m \geq 2$  and  $n \geq 3$ :

$$\begin{aligned}
 a_1 \text{ ALT } \dots \text{ ALT } a_m &: \bigvee_{1 \leq i \leq m} ((EF^\square\{a_i\} \text{ true}) \wedge \bigwedge_{1 \leq j \neq i \leq m} (\neg EF^\square\{a_j\} \text{ true})); \\
 (\neg)a_1 \text{ OR } \dots \text{ OR } (\neg)a_m &: \bigvee_{1 \leq i \leq m} (\neg)EF^\square\{a_i\} \text{ true}; \\
 a_1 \text{ EXC } a_2 &: \neg((EF^\square\{a_1\} \text{ true}) \wedge (EF^\square\{a_2\} \text{ true})); \\
 a_1 \text{ REQ } a_2 &: (EF^\square\{a_1\} \text{ true}) \implies (EF^\square\{a_2\} \text{ true}); \\
 a_1 \text{ REQ } (a_2 \text{ ALT } \dots \text{ ALT } a_n) &: (EF^\square\{a_1\} \text{ true}) \implies \\
 &\quad (\bigvee_{2 \leq i \leq n} (EF^\square\{a_i\} \text{ true}) \wedge \bigwedge_{2 \leq j \neq i \leq n} (\neg EF^\square\{a_j\} \text{ true})); \\
 a_1 \text{ REQ } (a_2 \text{ OR } \dots \text{ OR } a_n) &: (EF^\square\{a_1\} \text{ true}) \implies \bigvee_{2 \leq i \leq n} (EF^\square\{a_i\} \text{ true}).
 \end{aligned}$$

Now that all variability constraints are formalised in v-ACTL, we can refine the definition of a valid product (LTS) (cf. Def. 12 from Section 3.3). Let  $\mathcal{F} = (Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$  be an MTS with a set  $\mathcal{V}$  of variability constraints expressed in v-ACTL. Then, the set of all *valid products* (LTSs) of the MTS  $\mathcal{F}$  with variability constraints  $\mathcal{V}$ , denoted by  $\mathcal{I}_{vp}(\mathcal{F}, \mathcal{V})$ , is defined as follows:

$$\begin{aligned}
 \mathcal{I}_{vp}(\mathcal{F}, \mathcal{V}) &= \{\mathcal{P}_i \in \mathcal{I}_{cp}(\mathcal{F}) \mid \mathcal{P}_i \models \phi \text{ for all } \phi \in \mathcal{V}\} \\
 &\simeq \{\mathcal{P}_i \in \mathcal{P}_{\mathcal{F}} \mid \mathcal{P}_i \models \phi \text{ for all } \phi \in \mathcal{V}\},
 \end{aligned}$$

where  $\simeq$  denotes equality modulo bisimulation (cf. Theorem 2).

Figure 4 illustrates the various definitions involved in the stepwise refinement of an MTS  $\mathcal{F}$  into LTSs, among which its unique valid product with respect to the variability constraint  $\mathcal{V} = \{a \text{ REQ } c\}$ .

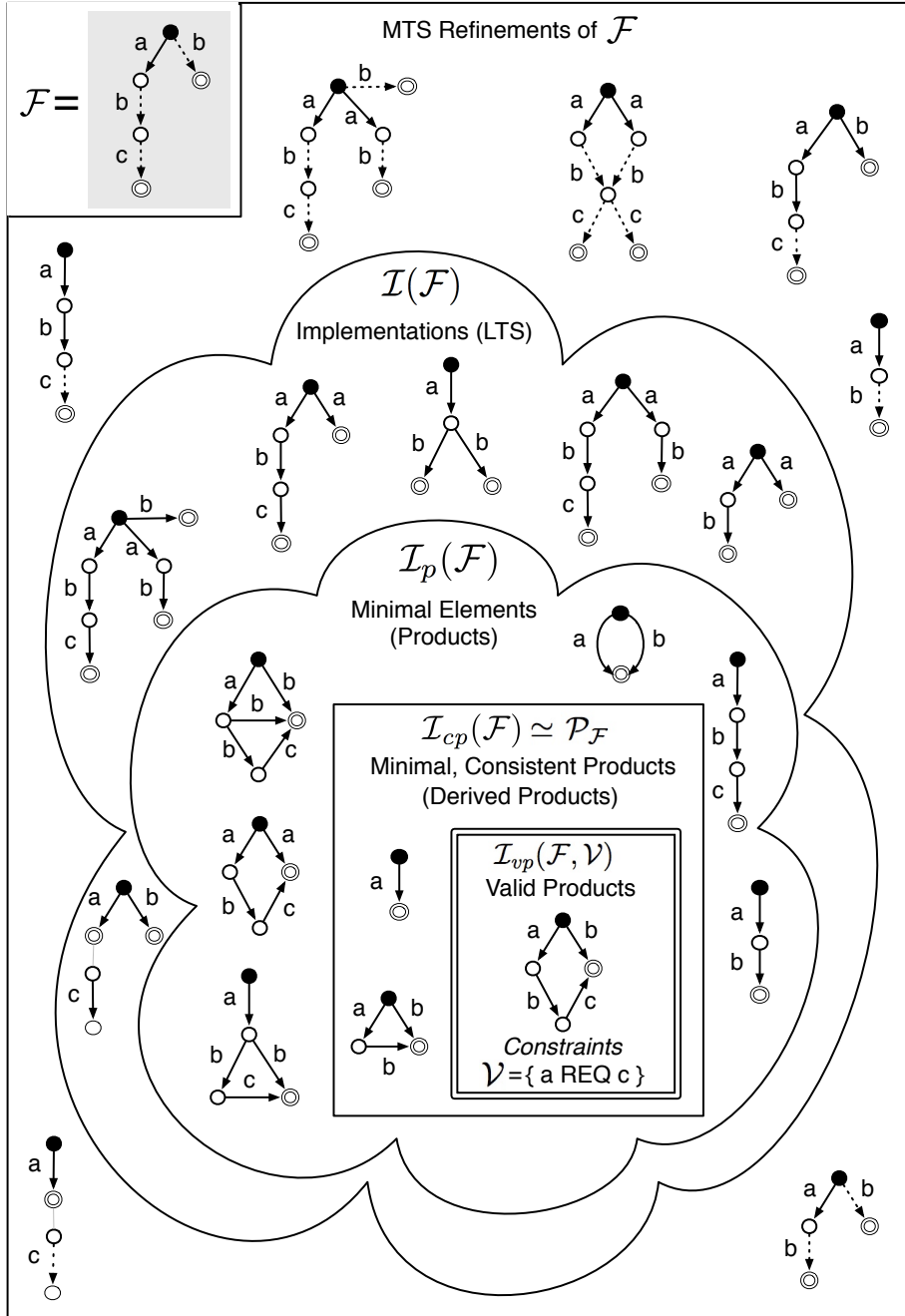


Figure 4: A universe of MTS refinements

Note that since LTSs contain only must transitions, when v-CTL formulae are verified over products, the ‘boxed’ operators of v-CTL simply collapse onto their classic interpretations.

Apart from verifying variability constraints, we would like to use v-CTL to verify (temporal) logic properties of product families and of all their products. To this aim, however, we note that it is not always easy to interpret the result of verifying a v-CTL formula over an MTS, since in general a v-CTL formula that is true for an MTS, may be false for some of its implementations, and vice versa, as the following examples illustrate.

**Example 10** (Example 5 continued). *The property ‘Whenever a coffee is selected, a coffee is eventually delivered’ can be formulated as:*

$$AG [\textit{coffee}] AF^\square \{ \textit{pour espresso} \vee \textit{pour regular} \} \textit{true}$$

Note that we use the ‘boxed’ variant of the classic  $F$  operator to express the fact that eventually coffee is actually poured. As a result, this formula does not hold for the MTS modelling the family of coffee machines depicted in Fig. 2(a), but it does hold in the product LTSs depicted in Figs. 2(b) and 3(b), i.e. the above formula expresses a property that holds for these Canadian and European coffee machines.

**Example 11.** Consider the MTSs  $\mathcal{F}$  and  $\mathcal{F}_p$  in Fig. 5(a). Clearly  $\mathcal{F} \preceq \mathcal{F}_p$  with  $(q, q_p) \in \mathcal{R}$ . Since there is no must transition from  $q$  in  $\mathcal{F}$ , it is trivially the case that  $q \models [a]^\square \phi$ . However, from the must transition  $(q_p, a, q'_p)$  in  $\mathcal{F}_p$  and the fact that  $q'_p \models \neg\phi$ , it follows immediately that  $q_p \not\models [a]^\square \phi$ .

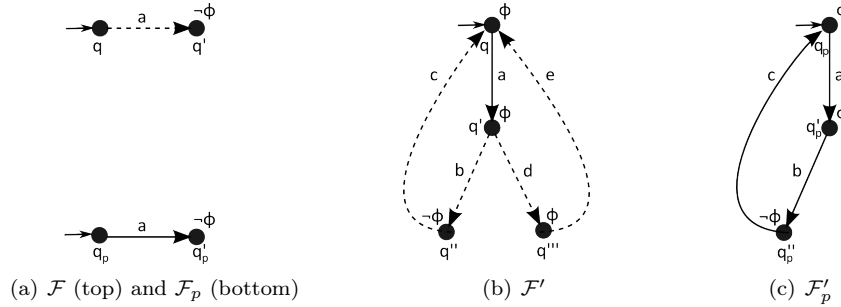


Figure 5: (a)-(c) Counterexamples for preservation by refinement and by live MTS

In the next sections, we will discuss the inheritance of analysis results from families to products. We study under which conditions a v-CTL property that is true for an MTS is also true for its product LTSs, and likewise for the preservation of properties that are false. The final aim is to achieve a specific type of *family-based verification* [66]: once a property is verified for a family model, one knows that the result also holds for any of its product models, without the need to explicitly verify the property over the products

(as opposed to *product-based verification*, in which every product has to be examined individually). The above examples provide some intuition for the reason why, in general, it might not be the case that the result of a v-ACTL formula is preserved from an MTS to its product LTSs. Hence, we need to look for fragments of v-ACTL for which such preservation results do exist.

Already in [5], it was noted that the refinement relation is similar to the well-known notion of simulation between LTSs, and hence the classical result of preservation of the universal fragment of CTL by simulation [24] can be reused for v-ACTL. In the following two sections, we precisely define the various fragments of v-ACTL that are preserved and by which kind of refinement.

#### 4.2. Preservation of Properties by Refinement

In this section, we will define a fragment of v-ACTL with the following characteristic: all formulae expressed in it that hold for an MTS preserve their validity for all refinements of that MTS, and hence, in particular, for all its valid product LTSs.

**Definition 17** (preservation by refinement). *A v-ACTL formula  $\phi$  is said to be preserved by refinement if for any two MTSs  $\mathcal{F}$  and  $\mathcal{F}_p$  such that  $\mathcal{F} \preceq \mathcal{F}_p$ , we have that whenever  $\mathcal{F} \models \phi$ , then  $\mathcal{F}_p \models \phi$ .*

We now introduce some fragments of v-ACTL and then demonstrate that they are preserved by refinement.

**Definition 18** (syntax of v-ACTL<sup>□</sup>). *The fragment v-ACTL<sup>□</sup> of v-ACTL is defined as follows:*

$$\begin{aligned} \phi ::= & \text{false} \mid \text{true} \mid \phi \wedge \phi \mid \phi \vee \phi \mid [\chi]\phi \mid \langle \chi \rangle^\square \phi \mid EF^\square \phi \mid EF^\square \{\chi\} \phi \mid \\ & AF^\square \phi \mid AF^\square \{\chi\} \phi \mid AG \phi \mid \neg \psi \end{aligned}$$

where

$$\psi ::= \text{false} \mid \text{true} \mid \psi \wedge \psi \mid \psi \vee \psi \mid \langle \chi \rangle \psi \mid EF \psi \mid EF\{\chi\} \psi \mid \neg \phi$$

Note that v-ACTL<sup>□</sup> consists of two parts. The first fragment (v-ACTL<sup>+</sup>) is such that any formula expressed in it that is true for an MTS  $\mathcal{F}$  is also true for all MTSs  $\mathcal{F}_p$  such that  $\mathcal{F} \preceq \mathcal{F}_p$ . The second fragment (v-ACTL<sup>-</sup>), which in v-ACTL<sup>□</sup> appears negated, is such that any formula expressed in it that is false for an MTS  $\mathcal{F}$  is also false for all MTSs  $\mathcal{F}_p$  such that  $\mathcal{F} \preceq \mathcal{F}_p$ .

**Theorem 3** (preservation by refinement). *Any formula  $\phi$  of v-ACTL<sup>□</sup> is preserved by refinement.*

*Proof.* Let  $\mathcal{F}$  and  $\mathcal{F}_p$  be two MTSs such that  $\mathcal{F} \preceq \mathcal{F}_p$ . We show via structural induction on  $\phi$  that for a pair of states  $q$  of  $\mathcal{F}$  and  $q_p$  of  $\mathcal{F}_p$ , with  $(q, q_p) \in \mathcal{R}$ , we have that whenever  $q \models \phi$ , then  $q_p \models \phi$ :

- *false* and *true* are trivially preserved by refinement.

- if  $\phi, \phi'$  are preserved by refinement, then obviously so is  $\phi \wedge \phi'$ .
- if  $\phi, \phi'$  are preserved by refinement, then obviously so is  $\phi \vee \phi'$ .
- if  $\phi$  is preserved by refinement, then so is  $[\chi] \phi$ .  
Indeed, if  $q \models [\chi] \phi$ , then in all next states  $q'_i$  in  $\mathcal{F}$ , reachable by a may transition executing an action satisfying  $\chi$ ,  $\phi$  holds. By the definition of refinement, only a subset of these states will have a corresponding state  $q'_{p_i}$  in  $\mathcal{F}_p$ , reachable by a may transition from  $q_p$ , such that  $(q'_i, q'_{p_i}) \in \mathcal{R}$ . However, by induction any such state will satisfy  $\phi$ , which implies that  $q_p \models [\chi] \phi$ .
- if  $\phi$  is preserved by refinement, then so is  $\langle \chi \rangle^\square \phi$ .  
Indeed, if  $q \models \langle \chi \rangle^\square \phi$ , then there exists a next state  $q'$  in  $\mathcal{F}$ , reachable by a must transition executing an action satisfying  $\chi$ , in which  $\phi$  holds. By the definition of refinement, there must exist a corresponding state  $q'_p$  in  $\mathcal{F}_p$ , reachable by a must transition from  $q_p$ , such that  $(q', q'_p) \in \mathcal{R}$ , which hence satisfies  $\phi$ . It follows that  $q_p \models \langle \chi \rangle^\square \phi$ .
- if  $\phi$  is preserved by refinement, then so is  $EF^\square \phi$ .  
Indeed, if  $q \models EF^\square \phi$ , then there exists a must path from  $q$  to a state  $q'$  in  $\mathcal{F}$  in which  $\phi$  holds. By the definition of refinement and Proposition 1, it follows that there exists a must path from  $q_p$  to a state  $q'_p$  in  $\mathcal{F}_p$  such that  $(q', q'_p) \in \mathcal{R}$  and  $q'_p$  satisfies  $\phi$ . Hence  $q_p \models EF^\square \phi$ .
- if  $\phi$  is preserved by refinement, then so is  $EF^\square \{ \chi \} \phi$ .  
The argument is similar to the previous case.
- if  $\phi$  is preserved by refinement, then so is  $AF^\square \phi$ .  
Indeed, if  $q \models AF^\square \phi$ , then either there is no transition from  $q$ , which means that there is no transition from  $q_p$  either and thus  $q_p \models AF^\square \phi$ , or all paths from  $q$  are must paths, each of which contains a state  $q'_i$  in  $\mathcal{F}$  in which  $\phi$  holds. By the definition of refinement and Proposition 1, all of these states will have a corresponding state  $q'_{p_i}$  in  $\mathcal{F}_p$ , reachable by a must path from  $q_p$ , such that  $(q', q'_{p_i}) \in \mathcal{R}$  and each  $q'_{p_i}$  satisfies  $\phi$ . Hence  $q_p \models AF^\square \phi$ .
- if  $\phi$  is preserved by refinement, then so is  $AF^\square \{ \chi \} \phi$ .  
The argument is similar to the previous case.
- if  $\phi$  is preserved by refinement, then so is  $AG \phi$ .  
Indeed, if  $q \models AG \phi$ , then either there is no transition from  $q$ , which means that there is no transition from  $q_p$  either and thus  $q_p \models AG \phi$ , or all the states  $q'_i$  in  $\mathcal{F}$  satisfy  $\phi$ . In the latter case, by the definition of refinement, only some of these states  $q'_i$  will have a corresponding state  $q'_{p_i}$  in  $\mathcal{F}_p$ , reachable from  $q_p$ , such that  $(q'_i, q'_{p_i}) \in \mathcal{R}$ . However, by induction any such state will satisfy  $\phi$  and hence  $q_p \models AG \phi$ .

Finally, the preservation of a negation  $\neg\psi$  follows from the fact that for any two MTSs  $\mathcal{F}$  and  $\mathcal{F}_p$  such that  $\mathcal{F} \preceq \mathcal{F}_p$ , after all,  $\mathcal{F}_p$  is a subgraph of  $\mathcal{F}$  (after all, by definition no transitions are added during refinement). As a result, existential formulae concerning reachability, like those expressible in  $v\text{-ACTL}^-$ , cannot be true in  $\mathcal{F}_p$  while false in  $\mathcal{F}$ . Hence, if  $q \models \neg\psi$ , then  $q_p \models \neg\psi$ .  $\square$

Hence, formulae that are expressed exclusively with operators from the fragment  $v\text{-ACTL}^\square$  of  $v\text{-ACTL}$  are preserved by refinement according to Def. 17. A formula preserved by refinement is obviously also preserved by implementation, and by the product relation, defined in Section 3.1, which means that Theorem 3 can also be applied in the specific application of our theory of MTSs in SPLE. In that case, the verification result of a  $v\text{-ACTL}^\square$  formula over an MTS continues to hold for all its products (LTSs), thus allowing family-based verification.

**Example 12** (Example 5 continued). *The property ‘Whenever a cappuccino is selected, milk is eventually poured’ can be formulated as:*

$$AG [\text{cappuccino}] AF^\square \{\text{pour milk}\} \text{ true}$$

*Since this  $v\text{-ACTL}^\square$  formula holds for the MTS modelling the family of coffee machines depicted in Fig. 2(a), Theorem 3 implies that it also holds for all its (valid) products, i.e., including the ones represented by the LTSs depicted in Fig. 3. Note that it trivially holds for the valid product depicted in Fig. 2(b).*

The following example shows that the contrary does not hold: if a  $v\text{-ACTL}^\square$  formula holds for all products of an MTS, there is no guarantee that it holds for the MTS.

**Example 13.** *Consider the MTS in Fig. 6(a) and all its products in Fig. 6(b), i.e., not necessarily consistent. The property ‘If action b occurs, then also action a occurs’ can be formulated as:*

$$EF \{b\} \text{ true} \implies EF \{a\} \text{ true}, \text{ i.e., } (\neg EF \{b\} \text{ true}) \vee EF \{a\} \text{ true} \quad (1)$$

*It is clear that this  $v\text{-ACTL}$  formula holds for all products of the MTS. However, Formula 1 is not a  $v\text{-ACTL}^\square$  formula (due to  $EF \{a\}$ ). If we want to verify the above property over the MTS and be able to draw conclusions for all its products, then we need to specifically consider the modalities of the MTS and use the ‘boxed’ variant of the action-based  $F$  operator, i.e.,*

$$(\neg EF \{b\} \text{ true}) \vee EF^\square \{a\} \text{ true} \quad (2)$$

*However, this  $v\text{-ACTL}^\square$  formula does not hold for the MTS, while it does hold for all products. Note, moreover, that Formula 2 actually formalises a rather different property than the one above, viz. ‘In no product an action b occurs, or in all products an action a occurs’.*

Furthermore, we note that a set of valid products (modelled by LTSs) can be derived from different MTSs satisfying different properties, as the next example shows. This is particularly evident in the presence of variability constraints.

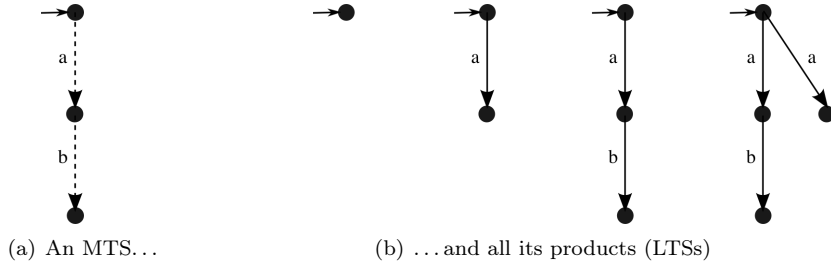


Figure 6: Counterexample for bottom-up ‘preservation’

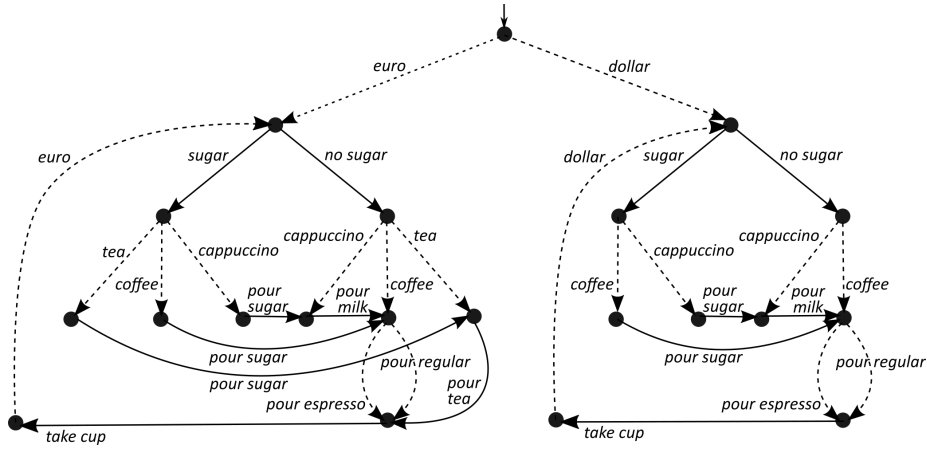


Figure 7: Alternative MTS modelling the family of European and Canadian coffee machines

**Example 14** (Example 5 continued). *With respect to the variability constraints listed in Example 9, the alternative MTS depicted in Fig. 7 generates exactly the same valid products as the MTS depicted in Fig. 2(a). However, the latter is a much more compact, abstract model. As usual, abstraction comes at a price: the  $v\text{-ACTL}^\square$  formula*

$$[\text{euro}] \neg EF \{ \text{dollar} \} \text{ true}$$

*holds in the alternative MTS of Fig. 7, but not in the original one of Fig. 2(a). This formula obviously does hold in all valid products of these MTSs.*

*For quite larger product families (think, e.g., of a family of coffee machines accommodating numerous different coins, as sketched in Example 7), the alternative modelling of Fig. 7 is not a viable alternative. In such cases, the trade-off between abstraction and accuracy would probably result in favour of abstraction, thus preferring efficient family-based verification on a compact, abstract MTS over something close to product-based verification on a large, detailed MTS.*



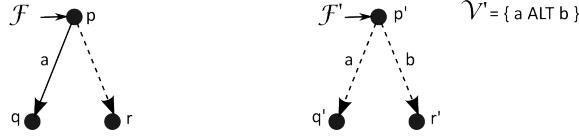


Figure 8: Two MTSs with live sets of actions

#### 4.3. Preservation of Properties by Live MTSs

In this section, we will define a wider fragment of v-CTL with the following characteristic: all formulae expressed in it that hold for an MTS preserve their validity for all valid product LTSs of that MTS. Recall from the definition of a valid product in Def. 12 that this means that we specifically consider the variability constraints associated with the MTSs. This wider fragment of v-CTL is preserved by all valid products if the MTS is ‘live’, in the sense that every path is infinite.

We now formally define *live MTSs* based on *live sets of actions* and *live states*, after which a result similar to Theorem 3 is shown to hold, but this time taking the possible variability constraints associated with an MTS into account.

**Definition 19** (live states). *Let  $\mathcal{F} = (Q, \Sigma, \bar{q}, \delta^\diamond, \delta^\square)$  be an MTS, with an associated set  $\mathcal{V}$  of variability constraints, and let  $q \in Q$ . Then, the state  $q$  is live if one of the following holds:*

- *there exists a  $q' \in Q$  such that  $q \xrightarrow{a}_\square q'$ , for some  $a \in \Sigma$ , or*
- *the set of actions  $\Gamma \subseteq \Sigma$  for which there exists a  $q' \in Q$  such that  $q \xrightarrow{a}_\diamond q'$ , for some  $a \in \Gamma$ , contains a live set of actions,*

where a live set of actions is defined as follows:

- *if  $\mathcal{V}$  contains at least one of the following type of constraints, with  $\{a_i \mid 0 \leq i \leq n, n \geq 2\} \subseteq \Sigma$ :*  
 $a_1 \text{ ALT } \dots \text{ ALT } a_n$   
 $a_1 \text{ OR } \dots \text{ OR } a_n$   
*then  $\{a_i \mid 1 \leq i \leq n\}$  is a live set of actions.*

This definition implies that a live state of an MTS does not occur as a final state in any of its products. Consider, e.g., the MTS  $\mathcal{F}$  and  $\mathcal{F}'$ , with set  $\mathcal{V}'$  of variability constraints, in Fig. 8. Obviously,  $a$  and  $b$  form a live set of actions due to the presence of the variability constraint  $a \text{ ALT } b$  in  $\mathcal{V}'$ . Hence, it is easy to see that the states  $p$  and  $p'$  are thus live states of the MTSs  $\mathcal{F}$  and  $\mathcal{F}'$ , respectively, and that these are indeed not final states in any valid product LTS of these MTSs.

We can now lift the notion of liveness to MTSs.

**Definition 20** (live MTS). *An MTS  $\mathcal{F}$  is said to be live if all its states are live.*

We now have that if an MTS is live, then both the MTS and all its valid products have only infinite full paths (note that this is not the case for the MTSs in Fig. 8). This allows us to provide a version of preservation by refinement that is limited to live MTSs and valid products (cf. Def. 17).

**Definition 21** (preservation by live MTS). *A v-ACTL formula  $\phi$  is said to be preserved by live MTS if for a live MTS  $\mathcal{F}$  with variability constraints  $\mathcal{V}$ , we have that whenever  $\mathcal{F} \models \phi$ , then  $\mathcal{F}_p \models \phi$  for all valid products  $\mathcal{F}_p \in \mathcal{I}_{vp}(\mathcal{F}, \mathcal{V})$ .*

Next, we define a slightly wider fragment of v-ACTL by adding to v-ACTL<sup>□</sup> the (possibly action-based) AF construct (“always possible”), after which we demonstrate that these are preserved by live refinement.

**Definition 22** (syntax of v-ACTLlive<sup>□</sup>). *The syntax of v-ACTLlive<sup>□</sup> of v-ACTL is defined as follows:*

$$\begin{aligned} \phi ::= & \text{false} \mid \text{true} \mid \phi \wedge \phi \mid \phi \vee \phi \mid [\chi] \phi \mid \langle \chi \rangle^\square \phi \mid EF^\square \phi \mid EF^\square \{ \chi \} \phi \mid \\ & AF \phi \mid AF \{ \chi \} \phi \mid AF^\square \phi \mid AF^\square \{ \chi \} \phi \mid AG \phi \mid \neg \psi \end{aligned}$$

where

$$\psi ::= \text{false} \mid \text{true} \mid \psi \wedge \psi \mid \psi \vee \psi \mid \langle \chi \rangle \psi \mid EF \psi \mid EF \{ \chi \} \psi \mid \neg \phi$$

**Theorem 4** (preservation by live MTS). *Any formula  $\phi$  of v-ACTLlive<sup>□</sup> is preserved by live MTS.*

*Proof.* Let  $\mathcal{F}$  be a live MTS with variability constraints  $\mathcal{V}$  and let  $\mathcal{F}_p \in \mathcal{I}_{vp}(\mathcal{F}, \mathcal{V})$ . Then by definition  $\mathcal{F} \preceq \mathcal{F}_p$ . We show via structural induction on  $\phi$  that for a pair of states  $q$  of  $\mathcal{F}$  and  $q_p$  of  $\mathcal{F}_p$ , with  $(q, q_p) \in \mathcal{R}$ , we have that whenever  $q \models \phi$ , then  $q_p \models \phi$ . We inherit all clauses of Theorem 3 for standard preservation by refinement (if a formula is preserved by refinement, it is obviously preserved by live MTS). Hence, the following two clauses complete the proof:

- if  $\phi$  is preserved by live MTS, then so is  $AF \phi$ .  
Recall that a live MTS has only infinite full paths (as all its states are live). Indeed, if  $q \models AF \phi$ , then all (infinite) paths from  $q$  contain a state  $q'_i$  in  $\mathcal{F}$  in which  $\phi$  holds. By the definition of refinement and Proposition 1, the (infinite) paths from  $q_p$  are a subset of those from  $q$  and by the liveness of the MTS, for any valid product this subset cannot be empty. Moreover, any path from  $q_p$  in this subset thus contains a state satisfying  $\phi$ . Hence  $q_p \models AF \phi$ .
- if  $\phi$  is preserved by live MTS, then so is  $AF \{ \chi \} \phi$ .  
The argument is similar to the previous case. □

The following example shows a v-ACTLlive<sup>□</sup> formula that is (thus) preserved by live MTS, but, in general, not by MTSs that are not live.

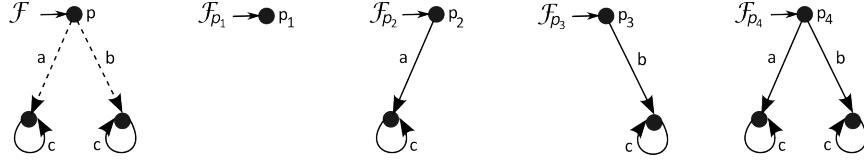


Figure 9: Counterexample for preservation by MTS in case the MTS is not live

**Example 15.** Consider the MTS  $\mathcal{F}$  in Fig. 9 (left). It is easy to see that  $p \models AF\{c\}$  true. Now consider all its (valid) products  $\mathcal{I}_{vp}(\mathcal{F}, \emptyset) = \mathcal{I}_p(\mathcal{F}) = \{\mathcal{F}_{p_1}, \mathcal{F}_{p_2}, \mathcal{F}_{p_3}, \mathcal{F}_{p_4}\}$ , depicted to the right of  $\mathcal{F}$  in Fig. 9. Note that  $\mathcal{F}_{p_i}$ , for  $i \in \{2, 3, 4\}$ , is live, while  $\mathcal{F}_{p_1}$  is not. In fact, we see that  $p_i \models AF\{c\}$  true, for  $i \in \{2, 3, 4\}$ , whereas  $p_1 \not\models AF\{c\}$  true.

The next example provides some intuition for the fact that even for the specific case of live MTSs, it is in general still not the case that all v-CTL formulae are preserved. All we know is that formulae that are expressed exclusively with operators from the fragment  $v\text{-CTL}^{\square}$  of v-CTL are preserved by live MTS.

**Example 16.** Consider the MTSs  $\mathcal{F}'$  and  $\mathcal{F}'_p$  in Figs. 5(b) and 5(c). Clearly  $\mathcal{F}' \preceq \mathcal{F}'_p$  with  $(q, q_p) \in \mathcal{R}$ . Since there exists a path in  $\mathcal{F}'$  on which  $\phi$  always holds, namely,  $qaq'dq'''eqaq' \dots$ , it is clear that  $q \models EG\phi$ . In  $\mathcal{F}'_p$ , however, no such path exists as any (infinite) path contains the state  $q''_p$  in which  $\phi$  does not hold. Hence,  $q_p \not\models EG\phi$ .

For the specific application of our theory of MTSs in SPLE, the result presented in Theorem 4 is an improvement over that of Theorem 3, since it allows family-based verification in a larger number of cases (because formulae that are preserved may now also contain AF constructs<sup>5</sup>).

So far we have seen that if an MTS is live, the validity of a  $v\text{-CTL}^{\square}$  formula (i.e., a  $v\text{-CTL}^{\square}$  formula plus AF constructs) is preserved in all valid products. However, the liveness of the states of the MTS is actually relevant only for the preservation of the validity of the evaluation of the AF constructs, since the preservation of the validity of the other  $v\text{-CTL}^{\square}$  operators does not depend on the liveness property of states of the MTS (cf. Theorem 3). A so-called *lazy*, on-the-fly evaluation of a  $v\text{-CTL}^{\square}$  formula may be carried out by evaluating a fragment of the original formula over a fragment of the state space of the MTS. A lazy evaluation of a formula evaluates subformulae only when their truth value is actually needed, which in case of Boolean connectives like or and and means that the truth value is returned as soon as the correct value is known (a.k.a. *short-circuit* or *minimal* evaluation). Moreover, the logical or and and connectives are evaluated in a specific order (e.g., from left-to-right)

<sup>5</sup>An AF construct is any of the constructs  $AF\phi$ ,  $AF\{\chi\}\phi$ ,  $AF^{\square}\phi$ , or  $AF^{\square}\{\chi\}\phi$  allowed by the syntax of v-CTL (cf. Def. 15).

and reachability operators like  $F\phi$  first evaluate  $\phi$  in the current state and then recursively evaluate the remainder of the path(s) in a specific order. For instance, it is clear that whenever  $\phi$  evaluates to true in  $\phi \vee \psi$  or to false in  $\phi \wedge \psi$ , then there is no need to evaluate  $\psi$ .

Therefore, requiring the whole MTS to be live (in order to preserve the validity of a formula over all its valid products) is in many cases a much stricter than necessary assumption. It would indeed be sufficient to require the liveness of all the states that are actually used by the recursive evaluation of the AF constructs. Consequently, it could be a task of the model checker, while it is evaluating the AF constructs, to check also the liveness of the states used for this operation, and to report at the end of the evaluation whether the result for the  $v\text{-ACTLive}^\square$  formula can be trusted to hold for all valid products (in which case we say that the MTS is *sufficiently live* with respect to this formula). This is precisely what is being done by our model checker VMC, that is presented in the next section. Hence the notion of sufficiently live MTSs is formula-dependent and tool-dependent (another tool might choose to expand some other parts of a formula first).

The next example illustrates this reasoning by providing three cases in which a  $v\text{-ACTLive}^\square$  formula with an action-based AF construct is preserved from an MTS to all its valid products (LTSSs), even though the MTS is not live.

**Example 17** (Example 15 continued). *Consider the MTS  $\mathcal{F}^1$  in Fig. 10, with the associated set  $\mathcal{V} = \{a \text{ ALT } b\}$  of variability constraints. Even though the MTS is not live (because its final states are not live) and Theorem 4 thus cannot be applied, the  $v\text{-ACTLive}^\square$  formula  $AF\{c\}$  true needs to be evaluated only in live states of  $\mathcal{F}^1$ . Since  $\mathcal{F}^1 \models AF\{c\}$  true, the fact that the MTS is thus sufficiently live with respect to this formula implies that also  $\mathcal{P} \models AF\{c\}$  true for all valid products  $\mathcal{P}$  of  $\mathcal{F}^1$  (i.e.,  $\mathcal{F}_{p_1}^1$  and  $\mathcal{F}_{p_2}^1$  depicted in Fig. 10 immediately to the right of  $\mathcal{F}^1$ ).*

*Now consider the MTS  $\mathcal{F}^2$  in Fig. 10. Given the absence of a variability constraint, the initial and final states are not live and also  $\mathcal{F}^2$  is thus not a live MTS. However, given the  $v\text{-ACTLive}^\square$  formula  $([a] \langle c \rangle^\square \text{ true}) \vee AF\{d\}$  true, the AF construct is not actually evaluated because of the laziness of the choice operator. Only the first alternative is actually evaluated to conclude that  $\mathcal{F}^2 \models ([a] \langle c \rangle^\square \text{ true}) \vee AF\{d\}$  true. Hence the formula  $([a] \langle c \rangle^\square \text{ true}) \vee AF\{d\}$  true is preserved by all valid products according to the fact that this MTS is sufficiently live with respect to this formula, since no states are actually used by the AF construct, i.e.,  $\mathcal{P} \models ([a] \langle c \rangle^\square \text{ true}) \vee AF\{d\}$  true for all valid products  $\mathcal{P}$  of  $\mathcal{F}^2$ .*

*Finally, consider the MTS  $\mathcal{F}^3$  in Fig. 10(right). In this case, only the source and target states of the  $c$ -transition are live. Hence, neither  $\mathcal{F}^3$  is a live MTS. However, for the  $v\text{-ACTLive}^\square$  formula  $[a] AF\{e\}$  true, the state space fragment over which the AF construct is actually evaluated is live. Hence, this MTS is sufficiently live with respect to this formula and the formula  $[a] AF\{e\}$  true is thus preserved by all valid products. Since  $\mathcal{F}^3 \models [a] AF\{e\}$  true, it follows that  $\mathcal{P} \models [a] AF\{e\}$  true for all valid products  $\mathcal{P}$  of  $\mathcal{F}^3$ .*

In the next section, we present the model-checking tool VMC [15, 16] in

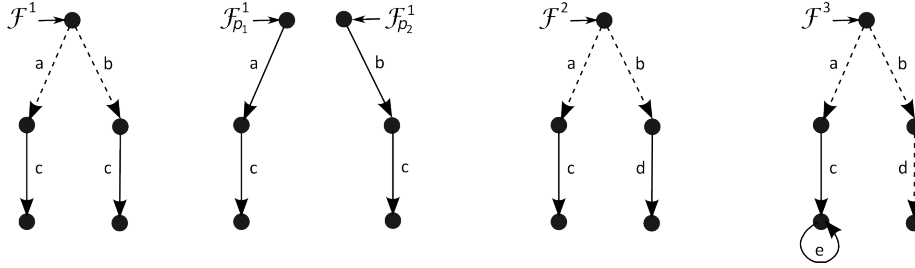


Figure 10: Preservation by sufficiently live MTSs

which the specification and verification framework discussed so far has been implemented. Its most recent version automatically notifies the user of the preservation of a model-checking result (from a family model to its products), whenever VMC is used to verify a v-CTL formula over an MTS to which one of the preservation results from this section applies, i.e., either Theorem 3 or 4 for formulae expressed in v-CTL<sup>□</sup> or v-CTLLive<sup>□</sup>, respectively, or the latter's extension to sufficiently live MTSs<sup>6</sup>.

## 5. VMC: A Tool for Modelling and Analysing Product Families

The framework developed in the previous sections has been implemented in a tool, whose usage is analysed in this section. For issues concerning the implementation of the model-checking algorithms and for architectural details we refer the reader to the appropriate tool papers [15, 16].

VMC is a tool for the modelling and analysis of behavioural variability in product families. It is the most recent product of the KandISTI family of model checkers [10] that have been developed at ISTI-CNR over the past two decades, including FMC [41], UMC [9], and CMC [38]. Each of these allows for efficient verification, by means of explicit-state on-the-fly model checking, of functional properties expressed in a specific action- and state-based branching-time temporal logic derived from the family of logics based on ACTL [33], the action-based version of CTL [22]. The shared model-checking engine underlying these model checkers has been highly optimised, as a result of which millions of states can now be verified in a few minutes. The on-the-fly nature of this family of model checkers means that in general not the whole state space needs to be generated and explored. This feature improves performance and allows one to partially verify also finite fragments of infinite-state systems. Furthermore, the family of model checkers offers advanced explanation techniques, such as the step-by-step illustration of counterexamples, which is particularly useful when model checking branching-time formulae.

<sup>6</sup>The preservation results for sufficiently live MTSs also hold for several operators from the aforementioned extended version of v-CTL that is implemented in VMC, including the least and greatest fixed point and (action-based and/or weak) Until operators.

The most recent advances of VMC (implemented in v6.2) concern an extension of its input language to a modal process algebra sustaining modal synchronisation operators (discussed below), value-passing communication (not treated here, cf. [11]),  $n$ -ary variability constraints and the possibility to negate actions in the ‘OR’ constraint allowing to express any Boolean function over the actions in its conjunctive normal form (reported in Section 3.3), and, finally, a thoroughly revised version of the supported v-ACTL logic with user notifications in case a family-based analysis result is guaranteed to be preserved by all products (reported in Section 4.3).

### 5.1. Modelling Product Families with Process Algebra

MTSs are not suitable to directly specify the behaviour of a complex system, possibly consisting of several components. In such cases, it is better to describe the system in an abstract high-level language that is interpreted over MTSs. The abstract syntax of the input language of VMC is based on the *process algebra* paradigm: each process represents a basic component of a possibly distributed system, and each system is thus defined inductively by composing the processes. Information on the modality of the transitions (may, must) must also be considered in the syntax of the process algebra used to specify the MTS. Our choice is to model it as a special additional parameter associated to the basic actions of the algebra. As a result, we consider a CSP-like process algebra in which the parallel composition operator synchronises all selected common actions (possibly with different modalities) of the processes involved in the composition. This is different from the CCS-based approaches in [14, 42, 45, 56]<sup>7</sup>.

**Definition 23** (syntax of the VMC input language). *Let  $\mathcal{A}$  be a set of actions, let  $a \in \mathcal{A}$ , and let  $L \subseteq \mathcal{A}$ . Processes are built from terms and actions according to the abstract syntax:*

$$\begin{aligned}
N & ::= [P] \\
[P] & ::= (K = T)^* P \\
P & ::= K \mid P / L / P \\
T & ::= nil \mid K \mid A.T \mid T + T \\
A & ::= a \mid a(\text{may})
\end{aligned}$$

where  $[P]$  denotes the complete system and  $K$  is a process identifier from the set of process definitions of the form  $K = T$ .

If  $L = \emptyset$ , then we sometimes write  $P // P$ . The set  $\{M, N, \dots\}$  of *systems* is denoted by  $\mathcal{N}$  and the set  $\{P, Q, \dots\}$  of *processes* is denoted by  $\mathcal{P}$ .

A process can thus be one of the following:

*nil* : a terminated process that has finished execution;

---

<sup>7</sup>Actually, VMC now accepts also value-passing in this process algebra, introduced in [11].

$K$  : a process identifier that can be used to model recursive sequential processes;

$A.P$  : a process that executes action  $A$  and then behaves as  $P$ ;

$P + Q$  : a process that non-deterministically chooses to behave as either  $P$  or  $Q$ ;

$P /L/ Q$  : a process formed by the parallel composition of  $P$  and  $Q$  that synchronously executes actions in  $L$  and independently executes (interleaves) other actions.

Note that parallel composition may only occur at the top-level of a VMC specification (i.e., a system consists of a parallel composition of sequential processes). Also note that VMC distinguishes must actions  $a$  and optional actions  $a(\text{may})$ . Each action modality is treated differently in the rules of the operational semantics over MTSs. Recall that, in an MTS, we depict must transitions (i.e.,  $\delta^\square$ ) by solid edges ( $\longrightarrow$ ) and optional transitions (i.e.,  $\delta^\diamond \setminus \delta^\square$ ) by dotted edges ( $\dashrightarrow$ ).

**Definition 24** (semantics of the VMC input language). *The operational semantics of a system  $N \in \mathcal{N}$  is described by the MTS  $(\mathcal{N}, \mathcal{A}, N, \delta^\diamond, \delta^\square)$ , where  $\delta^\diamond$  and  $\delta^\square$  are defined as the least relations that satisfy the set of transition rules and axioms in Figs. 11 and 12.*

$$\begin{array}{c}
\begin{array}{cc}
(\text{SYS}_\square) \frac{P \xrightarrow{a} P'}{[P] \xrightarrow{a} [P']} & (\text{SYS}_\diamond) \frac{P \dashrightarrow P'}{[P] \dashrightarrow [P']} \\
(\text{ACT}_\square) \frac{}{a.P \xrightarrow{a} P} & (\text{ACT}_\diamond) \frac{}{a(\text{may}).P \dashrightarrow P} \\
(\text{OR}_\square) \frac{P \xrightarrow{a} P'}{P + Q \xrightarrow{a} P'} & (\text{OR}_\diamond) \frac{P \dashrightarrow P'}{P + Q \dashrightarrow P'} \\
(\text{INT}_\square) \frac{P \xrightarrow{\ell} P'}{P /L/ Q \xrightarrow{\ell} P' /L/ Q} \ell \notin L & (\text{INT}_\diamond) \frac{P \dashrightarrow P'}{P /L/ Q \dashrightarrow P' /L/ Q} \ell \notin L \\
(\text{PAR}_\square) \frac{P \xrightarrow{a} P' \quad Q \xrightarrow{a} Q'}{P /L/ Q \xrightarrow{a} P' /L/ Q'} a \in L & (\text{PAR}_\diamond) \frac{P \dashrightarrow P' \quad Q \dashrightarrow Q'}{P /L/ Q \dashrightarrow P' /L/ Q'} a \in L \\
(\text{PAR}_\boxtimes) \frac{P \xrightarrow{a} P' \quad Q \dashrightarrow Q'}{P /L/ Q \dashrightarrow P' /L/ Q'} a \in L
\end{array}
\end{array}$$

Figure 11: The operational semantics of the input language of VMC in SOS style, with  $a, \ell \in \mathcal{A}$

$$\begin{array}{ccc}
P + Q \equiv Q + P & P /L/ Q \equiv Q /L/ P & \\
P + (Q + R) \equiv (P + Q) + R & P /L/ (Q /L/ R) \equiv (P /L/ Q) /L/ R & \\
P \equiv P + 0 & P \equiv P[{}^Q/K] \text{ whenever } K = Q & 
\end{array}$$

Figure 12: Structural congruence relation  $\equiv \subseteq \mathcal{P} \times \mathcal{P}$

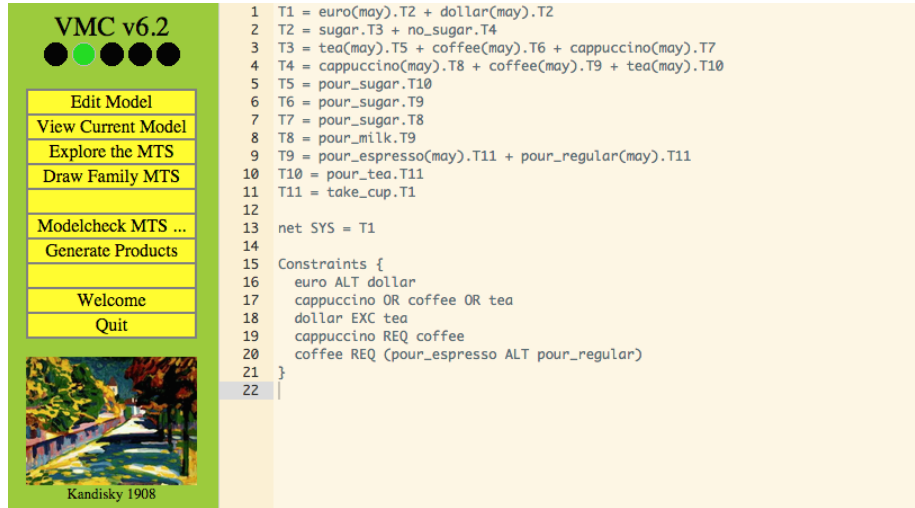
As usual, inference rules are expressed in terms of a (possibly empty) set of premises (above the line) and a conclusion (below the line). The reduction

relation is defined in Structural Operational Semantics (SOS) style (i.e., by induction on the structure of the terms denoting a process) modulo the structural congruence relation  $\equiv \subseteq \mathcal{P} \times \mathcal{P}$  defined in Fig. 12. Considering terms up to a structural congruence allows one to identify different ways of denoting the same process and the expansion of recursive process definitions.

From the inference rule  $\text{PAR}_{\boxtimes}$  in Fig. 11, it can be concluded that the synchronisation of  $a(\text{may})$  with  $a$  results in  $a(\text{may})$ . This is the common interpretation of synchronisation for MTSs [1]. Note, finally, that, when restricted to must actions (i.e., LTSs), the rules for action prefixing, non-deterministic choice, and parallel composition collapse onto the standard ones.

The complete definition of a product family in VMC now consists of two parts: a system specified in the process algebra of Def. 23 together with a (possibly empty) set of variability constraints specified according to Def. 11<sup>8</sup>. VMC thus hides the formulation of these variability constraints in v-ACTL (cf. Section 4.1) from the end-user.

**Example 18** (Example 5 continued). *The family of coffee machines modelled by the MTS in Fig. 2(a) and the variability constraints listed in Example 9 can be specified in VMC as depicted in Fig. 13.*



```

1 T1 = euro(may).T2 + dollar(may).T2
2 T2 = sugar.T3 + no_sugar.T4
3 T3 = tea(may).T5 + coffee(may).T6 + cappuccino(may).T7
4 T4 = cappuccino(may).T8 + coffee(may).T9 + tea(may).T10
5 T5 = pour_sugar.T10
6 T6 = pour_sugar.T9
7 T7 = pour_sugar.T8
8 T8 = pour_milk.T9
9 T9 = pour_espresso(may).T11 + pour_regular(may).T11
10 T10 = pour_tea.T11
11 T11 = take_cup.T1
12
13 net SYS = T1
14
15 Constraints {
16   euro ALT dollar
17   cappuccino OR coffee OR tea
18   dollar EXC tea
19   cappuccino REQ coffee
20   coffee REQ (pour_espresso ALT pour_regular)
21 }
22

```

Figure 13: Specification of the family of coffee machines in VMC

In this example, the system part or process model (i.e., without the constraints) can be seen as the natural encoding of the graph (MTS) of Fig. 2(a), with the process terms corresponding to the nodes of the graph. In general, however, more complex process models can be obtained by parallel composi-

<sup>8</sup>Besides the variability constraints of Def. 11, VMC accepts  $a_j$  IFF  $a_k$  as a shorthand for  $(a_j \text{ REQ } a_k) \wedge (a_k \text{ REQ } a_j)$ .



tion. The validity of properties expressed in v-CTL over a product family are verified directly over such a model, without considering variability constraints.

### 5.2. Generating Valid Products from Product Families

In [5], we defined an algorithm to automatically derive all valid product LTSs of an MTS model of a product family and an associated set of variability constraints. We recently implemented an improved version in VMC, considering that any Boolean expression over actions can be provided through its conjunctive normal form. The underlying idea is to construct the set of valid products by incrementally unfolding the initial MTS according to the modality (may or must) of the transitions into a set of intermediate LTSs, while continuously verifying the coherence of these partial products with respect to the constraints and with respect to the consistency assumption. During unfolding, any intermediate LTS which is incompatible with the constraints or inconsistent is immediately discarded. This approach proved to be rather effective and so far the generation of all valid products never was a bottleneck for verification, but we admit that the scalability to more realistic problems of nearly industrial size was so far not investigated. Clearly, product generation has a worst-case complexity that is exponential with respect to the number of differently labelled optional transitions in the MTS (cf. Section 3.2). However, in practice the theoretical upper bound of  $2^n$  is significantly reduced (sometimes exponentially) by the presence of variability constraints (think, e.g., of  $a$  ALT  $b$  or  $a$  EXC  $b$  for an MTS with optional transitions labelled with  $a$  and with  $b$ ).

**Example 19** (Example 5 continued). *The variability constraints apparently reduce the number of at most  $2^7$  consistent products (cf. Section 3.2) to 13 valid products. This can be concluded from Fig. 14, which shows the result of asking VMC to generate the products. For each product, the action labels of all may transitions that have been preserved (as must transitions) in that product’s LTS are listed. Clicking one of these products, VMC loads it and opens a new window with the product’s process model<sup>9</sup>. The result of doing so for product 6, i.e., the European coffee machine depicted in Fig. 3(b), is shown in Fig. 15.*

### 5.3. Analysing Product Families with VMC

In this section, we illustrate two kinds of analyses that VMC can perform upon loading the specification of the family of coffee machines given in Example 18.

First, the most efficient way to verify a property of a product family with VMC relies on the feasibility of performing the verification directly on the MTS modelling the family (i.e., to perform a family-based analysis). Indeed, we implemented the results presented in Section 4.3 (i.e., Theorems 3 and 4 and

---

<sup>9</sup>At this point, with all variability resolved, the opened window actually shows the model loaded in an instance of the aforementioned KandISTI family’s FMC model checker.

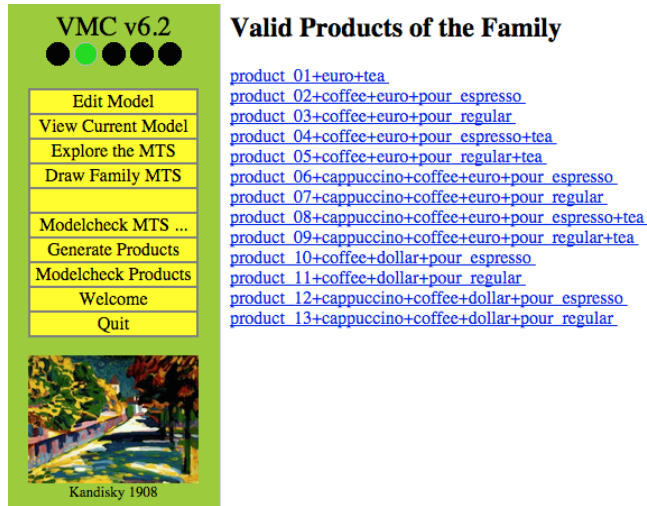


Figure 14: Products of the family of coffee machines generated by VMC

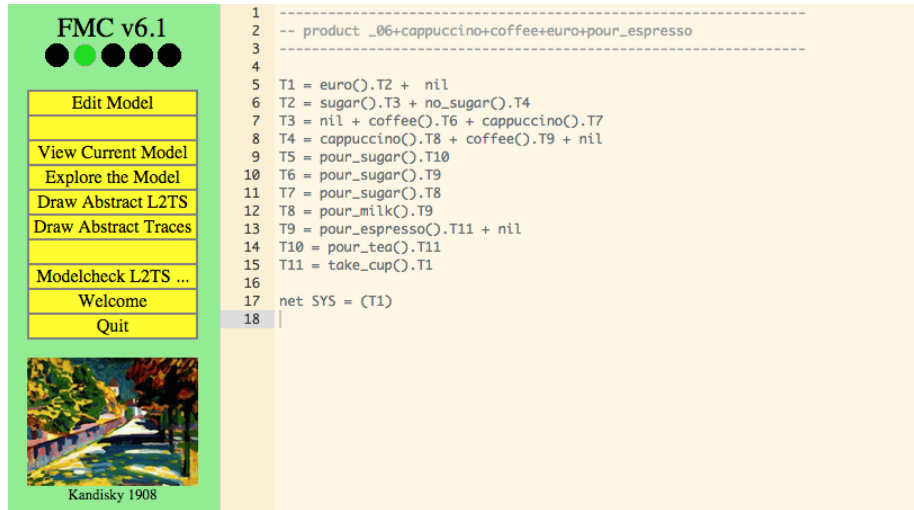


Figure 15: A product of the family of coffee machines generated by VMC

the notion of an MTS being sufficiently live with respect to a formula) in VMC such that VMC v6.2 now automatically notifies the user whenever the result of a formula verified over an MTS is preserved for all its valid products.

Figure 16 shows the result of verifying the  $v\text{-ACTLive}^\square$  formula

$$AG [sugar] AF \{pour\ sugar\} true \tag{3}$$

over the MTS. It expresses the property: *It is always the case that whenever sugar is chosen, eventually sugar is poured.* We see that Formula 3 is true. VMC

moreover reports that this result is preserved by all the valid products that can be generated from the MTS by considering also the variability constraints. This is because, even though the MTS is not live, it is sufficiently live with respect to Formula 3. The reason that the MTS is not live is due to the fact that the source state of the may transitions labelled with pour espresso and pour regular is not live.

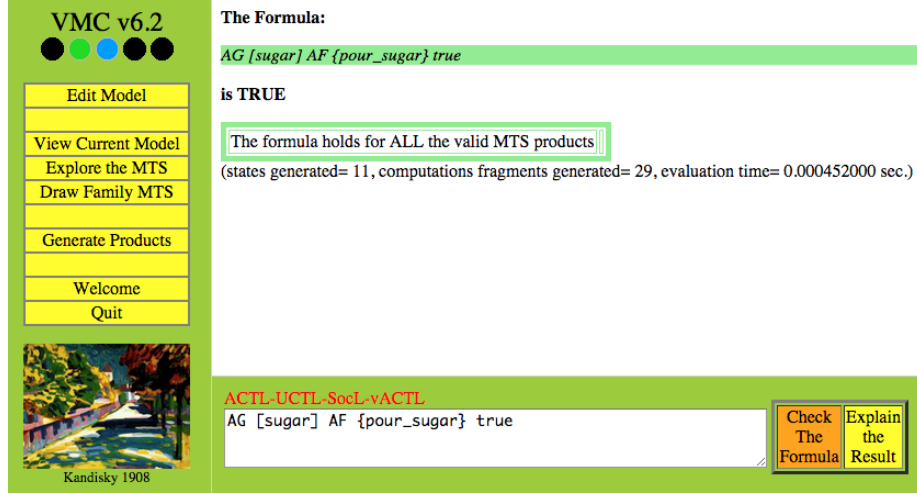


Figure 16: Formula 3 verified by VMC over the family of coffee machines

Figure 17, on the other hand, shows the result of verifying the v-CTL formula

$$AG ((\neg \langle sugar \rangle^{\square} true) \vee (\langle no\ sugar \rangle^{\square} true)) \quad (4)$$

over the MTS<sup>10</sup>. It expresses the property: *It is always the case that whenever sugar can be chosen, also no sugar can be chosen.* To see this, recall that  $(\neg \langle \chi \rangle^{\square} \psi) \vee (\langle \chi' \rangle^{\square} \psi) \equiv (\langle \chi \rangle^{\square} \psi) \implies (\langle \chi' \rangle^{\square} \psi)$ . We see that also Formula 4 is true. However, VMC furthermore reports that this result is not necessarily preserved by all the valid products that can be generated from the MTS by considering also the variability constraints. In fact, Formula 4 is not a v-CTL<sup>□</sup> formula, since  $\neg \langle \chi \rangle^{\square} \phi$  is not part of v-CTL<sup>□</sup>.

It is interesting to note, however, that a similar property to the above can easily be verified by the v-CTL<sup>□</sup> formula

$$AG ((\neg \langle sugar \vee no\ sugar \rangle true) \vee ((\langle sugar \rangle^{\square} true) \wedge (\langle no\ sugar \rangle^{\square} true)))$$

that holds for the MTS and thus also for all its valid products.

<sup>10</sup>As will be explained in Section 5.4, VMC actually translates this v-CTL formula in the CTL formula depicted in Fig. 17 and interprets it on an L<sup>2</sup>TS encoding of the MTS.

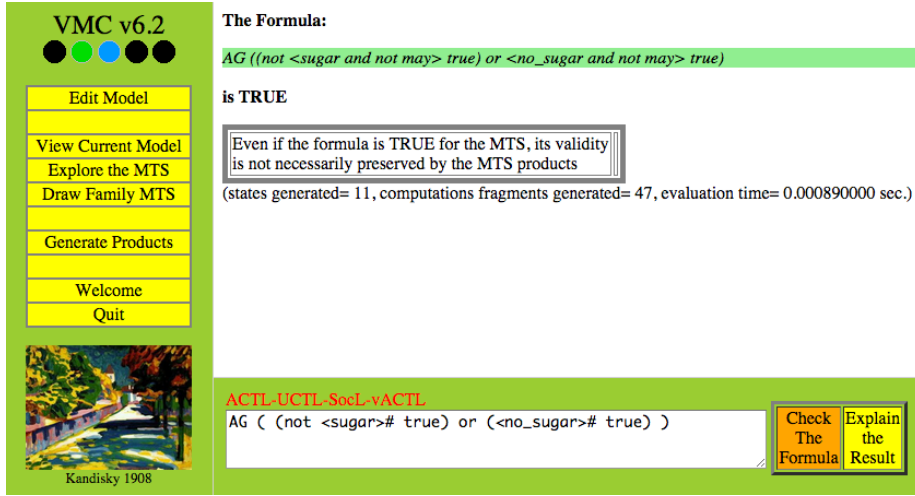


Figure 17: Formula 4 verified by VMC over the family of coffee machines

A limit of our approach is that the kind of verification VMC offers does not take all the additional variability constraints associated with the MTS into account. The model checker is only influenced by the definitions of live sets of actions induced by the constraints, but not by the full semantics of the latter. Therefore, properties still exist (also expressed in  $v\text{-ACTL}_{\text{Live}}^{\square}$ ) that hold for all valid products of an MTS, but which might not hold for the MTS (cf. the examples in Section 4). Conversely, any property which holds for just the valid products (i.e., there is at least one not valid product for which it does not hold), might not hold for the MTS.

Another limit concerns the logic’s applicability, which does not allow one to verify certain interesting properties specific to product families by family-based verification. It would, e.g., be nice to be able to verify for a family that a certain property  $\varphi$  holds for all products that eventually execute an *optional* action  $b$  (i.e., support a certain feature), but this kind of property is not among those that can be expressed in  $v\text{-ACTL}^{\square}$  and evaluated in VMC. Obviously, this property can be expressed in  $(v\text{-})\text{ACTL}$  (viz.  $EF\{b\} \text{ true} \implies \varphi$ ) and evaluated in VMC over an MTS, but its outcome will not allow one to draw any conclusions concerning its validity for the products of the MTS. To do so, one needs to resort to a separate product-by-product verification of this property, which brings us to the second analysis possibility offered by VMC.

The second, in general less efficient, way to verify a property of a product family with VMC exploits the possibility to first generate all valid products of the family (i.e., the finite set of consistent products satisfying all variability constraints) and then verify a property on each of these (i.e., to perform an enumerative product-based analysis). In this case, the logic does not even need to consider variability (since the formula is evaluated on the product LTSs), and thus we can make use of the extended version of  $v\text{-ACTL}$  mentioned in

Section 4, i.e., including various fixed-point and Until operators [15].

As illustrated in Fig. 14, product generation in VMC results in a list of all valid products (13 in this case, cf. Example 19) of the product family (i.e., this time taking also the variability constraints into account). As said before, for each product, VMC also lists the action labels of all may transitions that have been preserved (as must transitions) in that product's LTS.

**VMC v6.2**

- New Model ...
- Edit Current Model
- Explore the MTS
- View Current Model
- Draw Family MTS
- Generate Products
- Welcome
- Quit

**Evaluation of the formula "[dollar] EF {cappuccino} true" on all family products**

product_01+euro+tea	Formula evaluates	TRUE
product_02+coffee+euro+pour_espresso	Formula evaluates	TRUE
product_03+coffee+euro+pour_regular	Formula evaluates	TRUE
product_04+coffee+euro+pour_espresso+tea	Formula evaluates	TRUE
product_05+coffee+euro+pour_regular+tea	Formula evaluates	TRUE
product_06+cappuccino+coffee+euro+pour_espresso	Formula evaluates	TRUE
product_07+cappuccino+coffee+euro+pour_regular	Formula evaluates	TRUE
product_08+cappuccino+coffee+euro+pour_espresso+tea	Formula evaluates	TRUE
product_09+cappuccino+coffee+euro+pour_regular+tea	Formula evaluates	TRUE
product_10+coffee+dollar+pour_espresso	Formula evaluates	FALSE
product_11+coffee+dollar+pour_regular	Formula evaluates	FALSE
product_12+cappuccino+coffee+dollar+pour_espresso	Formula evaluates	TRUE
product_13+cappuccino+coffee+dollar+pour_regular	Formula evaluates	TRUE

Logic Formula for all Products  
[dollar] EF {cappuccino} true

Check The Formula | Explain the Result

Figure 18: Formula 5 verified by VMC over all products of the family of coffee machines

Next to the list of all valid products, Fig. 18 moreover shows the result of verifying the (v-)ACTL formula

$$[dollar] EF \{cappuccino\} true \quad (5)$$

over each of these products. It expresses the property: *Upon the insertion of a dollar, it might be the case that eventually a cappuccino can be chosen.* Note that Formula 5 does not hold for all the products. This is obviously due to the fact that cappuccino is an optional feature in our family of coffee machines. Since Formula 5 is not part of the v-CTL<sup>□</sup> fragment of v-CTL, from an SPLE point of view it makes no sense to verify it over the MTS, but when checked over the set of valid products one can precisely observe those products for which it holds and those for which it does not. In the next section, we will moreover see that verification with VMC is very efficient.

Obviously, we could now use the product-based analysis offered by VMC to verify also Formula 4 over all valid products. If we were to do so, then VMC would actually show that the formula's result is preserved by all valid products that can be generated from the MTS.

#### 5.4. Implementation Details

VMC's core contains a command-line version of the model checker and a product generation procedure, both stand-alone executables written in Ada

(easy to compile for Windows, Linux, Solaris, and Mac OS X) and wrapped with a set of CGI scripts handled by a web server, facilitating a HTML-oriented graphical user interface and integration with other tools for LTS minimisation and graph drawing. Its executables are available upon request. VMC is publicly usable online at <http://fmt.isti.cnr.it/vmc>.

Like the other model checkers of the aforementioned KandISTI family (FMC, UMC, and CMC), the VMC model-checking framework is based on the notion of a Doubly-Labelled Transition System (L<sup>2</sup>TS) [34] as the underlying abstract semantic computational model. An L<sup>2</sup>TS is an extension of an ordinary LTS in which not only edges but also states can be associated with (parametric) labels. Moreover, in the L<sup>2</sup>TSs that we consider, edges can be associated with *sets* of labels. These characteristics allowed us to define and implement the on-the-fly logical verification engine on the abstract L<sup>2</sup>TS in a way that is completely independent from the details of the underlying computational model (either based on UML state machines, as in FMC and UMC, or on a process calculus, as in CMC and VMC). It is the task of the specification language and computational model to define what kind of information is to be mapped from the ground internal structure of the computational model onto the abstract type of labels in the L<sup>2</sup>TS. Since an MTS differs from an LTS only by distinguishing two possible kinds of transitions, viz. admissible (may) or necessary (must) transitions, this aspect can easily be encoded in an L<sup>2</sup>TS by extending the labels on the transitions with information about the modality of the edge (e.g.,  $\xrightarrow{\{b, may\}}$  instead of  $\xrightarrow{b}$ , cf. Fig. 19).

The logical verification engine of the KandISTI model-checking framework has no problems analysing the L<sup>2</sup>TS derived from an MTS, and the additional logical operators specific to v-CTL can easily be defined through translations into the classical CTL operators. In Fig. 19, e.g., we show how a v-CTL formula over the MTS representation of a small process can be interpreted over an L<sup>2</sup>TS in plain CTL. Recall that this ‘boxed’ variant of the classical diamond operator of Hennessy-Milner logic requires that a next state exists, reachable by a necessary (must) transition labelled with action  $a$ , in which *true* holds.

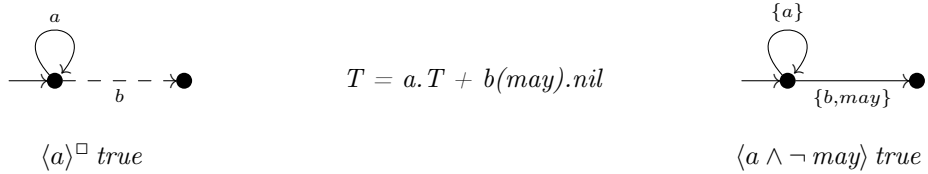


Figure 19: From a v-CTL formula and an MTS representation (left) of a process (middle) to a corresponding CTL formula over the L<sup>2</sup>TS interpretation (right) of the MTS / process

If the MTS contains no optional transitions, which we recall to be admissible (may) transitions that are not necessary (must) transitions, then the encoding of the MTS in VMC becomes precisely the standard encoding of an ordinary LTS (with no need for state labels).

As we have seen in Section 5.3, in VMC the propositional connectives  $\neg$ ,  $\wedge$ ,  $\vee$ , and  $\implies$  need to be written as **not**, **and**, **or**, and **implies**, respectively, while the must operators  $[\ ]^\square$ ,  $\langle \rangle^\square$ ,  $F^\square$ , and  $F^\square\{\}$  may either be written as  $[\ ]\#$ ,  $\langle \rangle\#$ ,  $F\#$ , and  $F\#\{\}$ , respectively, or translated in plain ACTL as mentioned before.

Model checking v-ACTL formulae (without the fixed-point operators) over MTSs is achieved in VMC with a worst-case complexity that is linear with respect to the size of the state space and the size of the formula.

## 6. Related Models and Tools

There are a few other approaches to modelling and analysing (by means of model checking) the behavioural variability in product families, which we describe and compare next.

### 6.1. Featured Transition Systems

By far the most elaborated and closely related approach is the one developed on top of Featured Transition Systems (FTSs) [25–29, 32]. A common trait of FTS and MTS approaches is that the behaviour of an entire product family is modelled in a single compact transition system from which the valid products can be obtained through a notion of refinement. Formally, an FTS is an  $L^2TS$ , with an associated feature diagram, and such that each state is labelled with an atomic proposition, while each transition is labelled with an action and — in the improved definition from [29] — an associated feature expression (a Boolean formula defined over the set of features) that must hold for this specific transition to be part of the executable product behaviour. Hence, an FTS models a family of LTSs, one per product, which can be obtained by projection (i.e., all transitions whose feature expression is not satisfied by the specific product’s set of features are removed, as well as all states and transitions that because of this become unreachable). An MTS, on the other hand, has no associated feature diagram, but it has an associated set of variability constraints (expressed over action labels rather than over features) that each product must satisfy. These constraints, moreover, can express any Boolean function over the action labels, thus including all standard type of constraints that may be modelled by means of a feature diagram (expressed in terms of actions, though).

Comparisons of the FTS and MTS models underling these approaches have appeared throughout the literature. In [5], we demonstrated the need to equip MTSs with an additional set of variability constraints to be able to filter out those products among their refinements that are not valid and be left with all and only their valid products, which in case of FTSs is taken care of by inspecting the associated feature diagram. Recently, in [8], we presented an automatic technique to transform FTSs into MTSs with associated variability constraints, the crux being a transformation from variability constraints expressed in terms of features to variability constraints expressed in terms of actions. Finally, in [19], the expressiveness of FTSs and MTSs (and PL-LTSs, the semantic model

of PL-CCS [45]) in terms of their sets of definable products is investigated, after which testing equivalences are explored. It is shown that MTSs are the least expressive, FTSs the most, and PL-LTSs lie strictly in between the two. However, only MTSs *without* associated variability constraints are considered. Indeed, we know from [8] that once MTSs with additional variability constraints are considered, then they are at least as expressive as FTSs.

## 6.2. Model Checkers for FTSs

We now discuss a number of dedicated FTS model checkers. In [28], an explicit-state model-checking technique, i.e., progressing one state at a time, was defined to verify LTL properties over FTSs. It provides a means to check that whenever a behavioural property is satisfied by an FTS modelling a product family, then it is also satisfied by every product of that family, and whenever a property is violated, then not only a counterexample is provided, but also all products violating the property. Hence, all products of a family are verified at once. Moreover, in [28], classical local model checking, as implemented in VMC, is extended to exhaustive model-checking algorithms that continue their search after a violation was found. This allows one to return a set of violating and a set of satisfying products upon the verification of an LTL property. In [29], this approach was improved by using symbolic model checking, examining sets of states at a time, and a feature-oriented version of CTL. We now discuss the specific tools that came out of these approaches in more detail.

SNIP [25] is a well-maintained model checker for product families modelled as FTSs and specified in fPromela, which is an extension of the Promela input language of the well-known SPIN model checker [47]. Features are declared in the Text-based Variability Language (TVL) and are actually taken into account by SNIP’s model-checking algorithm for the verification of properties expressed in fLTL (feature LTL) interpreted over FTSs, e.g., to verify a property only over a subset of a family’s valid products. Unlike VMC, SNIP is a command-line tool with no graphical user interface and no possibility to generate and explore product behaviour. Moreover, it was built from scratch, while VMC profits from numerous optimisation techniques that were implemented over the years in KandISTI. SNIP, however, treats features as first-class citizens, with built-in support for feature diagrams, and it implements (exhaustive) model-checking algorithms tailored for SPLE.

Furthermore, SNIP has recently been re-engineered and the resulting tool suite ProVeLines [32] supports discrete as well as real-time models, various types of computations, and advanced feature notions. Unlike KandISTI, it is the specific model-checking engine that changes, while all tool variants share the same common input language fPromela.

The symbolic FTS model checking of [29], on the other hand, was implemented as a prototypical extension of the NuSMV model checker [21] with a fully symbolic algorithm for fCTL (feature CTL). Product families are still modelled as FTSs, but this time they must be specified in fSMV, which is a feature-oriented extension of the input language of (Nu)SMV that was independently developed in the context of research on the renown problem of feature



interaction [60]. In contrast with SNIP, a counterexample is produced only for the first violating product found.

### 6.3. Variants of MTSs for SPLE

We now briefly discuss some further variants based on MTSs which have been used for modelling and analysing behavioural SPL models. Again, we focus on analyses by means of model checking.

In [40], an algorithm was defined to check conformance of LTSs against MTSs according to a given branching relation, i.e., to check conformance of the behaviour of a product against that of its product family. It is a fixed-point algorithm that starts with the Cartesian product of the states and iteratively eliminates pairs that are invalid according to the given relation. The algorithm was implemented in the MTS Analyser (MTSA) [35], which is a tool built on top of the LTS Analyser (LTSA). Next to checking whether or not a given LTS conforms to a given MTS according to a number of different branching relations, it supports the modelling and analysis of MTSs specified in an extension of the process algebra FSP (Finite State Processes). It then allows 3-valued FLTL (Fluent LTL) model checking of MTSs, with admissible(may), necessary (must), and optional transitions, by reducing the verification to two FLTL model-checking runs on LTSs. It is known from [20] that model checking any 3-valued temporal logic is no more expensive than model checking the corresponding 2-valued temporal logic, meaning it can be performed using existing model checkers. Hence, like VMC, MTSA is an extension of a tool for LTSs, but unlike VMC, MTSA is a general-purpose MTS analyser that does not provide specific features for the generation and verification of product variability.

In [36], (Generalised) Extended Modal Transition System ((G)EMTS) were introduced to deal with so-called multiple optionality constraints in SPLE, i.e., constraints that impose any valid product implementation to contain at least, at most, or exactly one ( $k$ ) out of  $n$  features. This is done through the use of so-called hypertransitions, which are transitions that still have a single source state, but a set of target states. Each hypertransition thus effectively models a set of transitions. The notion of a GEMTS generalises that of a Disjunctive Modal Transition System (DMTS) [54] as well as that of a 1-selecting Modal Transition System (1MTS) [39]. Their differences lie in the refinement semantics of the hypertransitions. In a DMTS, the refinement semantics requires that at least one of the transitions of a hypertransition must be implemented. In a 1MTS, this disjunctive choice is transformed into an exclusive choice: precisely one of the transitions of a hypertransition must be implemented. Finally, a GEMTS allows two types of hypertransitions, each with its own refinement semantics: those that require at least  $k$ -of- $n$  transitions to be implemented and those that require at most  $k$ -of- $n$  transitions to be implemented. Note that the source states of all hypertransitions with a semantics that requires at least one transition to be implemented, are by definition live. A more detailed comparison, including a hierarchy that compares the expressiveness of each of these variants of MTSs with hypertransitions, is presented in [37].

Variable I/O automata were introduced in [55] to model product families, together with a model-checking approach to the verification of the conformance of products with respect to a family’s variability. This is achieved by using variability information in the model-checking algorithm (while exploring the state space, an associated variability model is consulted continuously). Properties expressed in CTL are verified by explicit-state model checking. Like modal I/O automata [52], variable I/O automata extend I/O automata with a distinction between may and must transitions. While [52] can be viewed as one of the first attempts at describing SPLs in a behavioural fashion, in [52], the focus was on configuring products by configuring an abstract variability model composed of modal I/O automata in such a way that the selected configuration can be refined to configurations of each of the smaller components modelled as modal I/O automata.

#### 6.4. Process-algebraic SPL Approaches

In [12–14, 42], a high-level feature-based modelling language for product families was developed. Like VMC’s input language, it is a process-algebraic language, but process execution is constrained by a store that regulates (dynamic) product configuration; its most recent version supports the specification and analysis of probabilistic models of product families and quantitative constraints. An implementation in the executable modelling language Maude [30] allows formal analyses that range from consistency checking (by means of SAT solving) of product configurations to LTL model checking of product behaviour, whereas a combination with the distributed statistical model checker MultiVeStA [64] allows one to estimate the likelihood of specific configurations and behaviour of a product family by means of statistical model checking [51].

Finally, in [17], the formal specification language and toolset mCRL2 [44] is exploited for the modelling and analysis of product families. Its parametrised data language allows one to model and select valid product configurations in the presence of feature attributes (as in ProVeLines) and quantitative constraints (as it happens with the approach in [12, 13]). Moreover, its industrial-strength and actively maintained toolset allows one to efficiently verify product behaviour by model checking properties in the modal  $\mu$ -calculus with data, in combination with advanced built-in minimisation techniques and a modular verification method developed to exploit feature-based factorisations of product families.

The above approaches all verify an abstract model of a product family. There are, however, also numerous SPL analysis approaches that operate directly on the source code. Also in this case, it often happens that existing tools for software model checking are adapted to deal with variability. Examples include an adaptation of ProMoVer [65] with variability annotations [62] for Java and the SPLverifier [2, 3] tool chain built on Java PathFinder [68] for Java and CPAchecker [7] for C code.

For further details and for other model-checking approaches in SPLE, we refer to the excellent survey [66] which covers not only SPL model checking, but

also type checking, static analysis, and theorem proving and which distinguishes, besides product-based and family-based analyses, also feature-based analyses, which are not relevant to our approach given that features are only implicitly present as actions in our model.

Finally, as a reviewer noted, the generation of valid products from an MTS with variability constraints which, as described in detail in Section 5.2, are obtained by adapting the MTS in a consistent way with respect to the v-CTL formulae expressing the variability constraints, has some commonalities with recent work on maximal synthesis for Hennessy-Milner logic [48], in which L<sup>2</sup>TSs are adapted in the least possible way in order to satisfy a formula in Hennessy-Milner logic. This is done in a so-called maximally permissive way, retaining the most possible behaviour under simulation, by means of unfolding the original model with respect to the depth of the synthesised formula.

## 7. Conclusions

We have presented the theory underlying our framework for the modelling and analysis of behavioural variability in product families, parts of which have been introduced in [4, 5], in full detail. Similar to other approaches based on MTSs [40, 52, 55], FTSs [26–29], and mCRL2 [17], our approach is based on the assumption that the behaviour of a family of products can be specified in a compact behavioural (LTS-based) model of a set of products modelled as LTSs.

In our setting, the family model is specified in a process-algebraic language with a semantic interpretation as MTS, together with an additional set of variability constraints (on actions). Properties to be verified (by means of efficient on-the-fly model checking) are formalised in v-CTL, a special-purpose logic that allows one to specifically take the modality of transitions (labelled with actions) into account. Given an MTS model of a product family, the framework supports both family-based analyses, based on a number of results on the preservation of properties (expressed in the v-CTL<sup>□</sup> and v-CTL<sup>Live</sup><sup>□</sup> fragments of v-CTL) from an MTS to its set of product LTSs, as well as product-based analyses, upon the generation of all valid product LTSs from the MTS. Moreover, all these features are implemented in a tool that also allows MTS/LTS visualisation, minimisation, etc., as we have illustrated in the case of our running example.

For a future version of VMC, we consider the development of a feature that allows a user to specify a property that is supposed to hold on set of products (LTSs) in plain CTL (i.e., without distinguishing may and must transitions), which is then implicitly translated in the corresponding v-CTL<sup>□</sup> formula and verified over the MTS model of these product LTSs.

There is an obvious trade-off between enumerative product-based analysis with highly optimised model checkers originally developed for (single) product engineering, like SPIN, NuSMV, and mCRL2, and family-based analysis with dedicated SPL model checkers, like SNIP and the aforementioned extension of NuSMV. In fact, according to [25], SPIN generally outperforms SNIP due to

SPIN’s many optimisations (e.g., partial order reduction). VMC combines elements of both analysis strategies. For properties expressed in v-CTL<sup>□</sup> or v-CTL<sup>Live</sup><sup>□</sup>, VMC, with its linear worst-time complexity with respect to the number of states of the MTS, clearly outperforms SNIP, which is still linear with respect to the size of the generated state space, which, however, may be exponential with respect to the number of features of the FTS. For general v-CTL properties, this largely depends on the size of the product family (i.e., its number of features), but it should be noted that VMC considers variability constraints expressed in terms of actions of MTSs, while SNIP considers variability expressed by feature models.

In the future, we intend to perform a quantitative evaluation of the expressivity, complexity, and scalability of our approach. The most important challenge in this direction is to experiment VMC on a more realistic case study, preferably of industrial size. To be able to do that, we intend to enrich the input process algebra of VMC with advanced data types (not just integers as in [11], but also tuples, sets, lists, etc.), which calls for a more complex underlying computational model. We also plan to extend the framework with support for other specification notations (such as, for instance, UML state machines, which are already supported by the FMC and UMC members of our KandISTI family of model checkers). A more high-level specification language is a prerequisite to be able to affront industrial case studies. Since fPromela can be used to specify FTSs in SNIP, the model transformation from FTSs to MTSs with variability constraints that we recently presented in [8] might turn out to be useful in this respect.

#### *Acknowledgements*

The first and third author were supported by the Italian MIUR project CINA (PRIN 2010LHT4KM) and the EU project QUANTICOL (600708).

Part of the first author’s work was conducted while he was on sabbatical leave at Leiden University. He gratefully acknowledges the hospitality and support of the Leiden Institute of Advanced Computer Science during his stay in Leiden.

#### **References**

- [1] A. Antonik, M. Huth, K.G. Larsen, U. Nyman, and A. Wařowski. 20 Years of Modal and Mixed Specifications. *Bulletin of the EATCS* 95 (2008), 94–129.
- [2] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. Strategies for Product-Line Verification: Case Studies and Experiments. In *Proceedings of the 35th International Conference on Software Engineering (ICSE’13)*. IEEE, 2013, 482–491.
- [3] S. Apel, H. Speidel, P. Wendler, A. von Rhein, and D. Beyer. Detection of Feature Interactions using Feature-Aware Verification. In *Proceedings of the 26th International Conference on Automated Software Engineering*

- (*ASE'11*) (P. Alexander, C.S. Pasareanu, and J.G. Hosking, eds.). IEEE, 2011, 372–375.
- [4] P. Asirelli, M.H. ter Beek, A. Fantechi, and S. Gnesi. A Logical Framework to Deal with Variability. In *Proceedings of the 8th International Conference on Integrated Formal Methods (IFM'10)* (D. Méry and S. Merz, eds.). *Lecture Notes in Computer Science* 6396, Springer, 2010, 43–58.
- [5] P. Asirelli, M.H. ter Beek, A. Fantechi, and S. Gnesi. Formal Description of Variability in Product Families. In *Proceedings of the 15th International Software Product Line Conference (SPLC'11)* (E.S. de Almeida, T. Kishi, C. Schwanninger, I. John, and K. Schmid, eds.). IEEE, 2011, 130–139.
- [6] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, 2008.
- [7] D. Beyer and M.E. Keremoglu. CPAchecker: A Tool for Configurable Software Verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)* (G. Gopalakrishnan and S. Qadeer, eds.). *Lecture Notes in Computer Science* 6806, Springer, 2011, 184–190.
- [8] M.H. ter Beek, F. Damiani, S. Gnesi, F. Mazzanti, and L. Paolini. From Featured Transition Systems to Modal Transition Systems with Variability Constraints. In *Proceedings of the 13th International Conference on Software Engineering and Formal Methods (SEFM'15)* (R. Calinescu and B. Rumpe, eds.). *Lecture Notes in Computer Science* 9276, Springer, 2015, 344–359.
- [9] M.H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. A state/event-based model-checking approach for the analysis of abstract system properties. *Science of Computer Programming* 76, 2 (2011), 119–135.
- [10] M.H. ter Beek, S. Gnesi, and F. Mazzanti. From EU Projects to a Family of Model Checkers: From Kandinsky to KandISTI. In *Software, Services and Systems: Essays Dedicated to Martin Wirsing on the Occasion of His Emeritation* (R. De Nicola and R. Hennicker, eds.). *Lecture Notes in Computer Science* 8950, Springer, 2015, 315–331.
- [11] M.H. ter Beek, S. Gnesi, and F. Mazzanti. Model Checking Value-Passing Modal Specifications. In *Perspectives of System Informatics: Revised papers of the 9th International Ershov Informatics Conference (PSI'14)* (A. Voronkov and I. Virbitskaite, eds.). *Lecture Notes in Computer Science* 8974, Springer, 2015, 304–319.
- [12] M.H. ter Beek, A. Lluch Lafuente, A. Legay, and A. Vandin. Quantitative Analysis of Probabilistic Models of Software Product Lines with Statistical Model Checking. In *Proceedings of the 6th International Workshop on Formal Methods for Software Product Line Engineering (FMSPLE'15)* (J.M. Atlee and S. Gnesi, eds.). *Electronic Proceedings in Theoretical Computer Science* 182, arXiv:1504.03476v1 [cs.LO], 2015, 56–70.

- [13] M.H. ter Beek, A. Lluch Lafuente, A. Legay, and A. Vandin. Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints. In *Proceedings of the 19th International Software Product Line Conference (SPLC'15)* (D.C. Schmidt, ed.). ACM, 2015, 56–70.
- [14] M.H. ter Beek, A. Lluch Lafuente, and M. Petrocchi. Combining Declarative and Procedural Views in the Specification and Analysis of Product Families. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13), Volume 2* (T. Kishi, S. Cohen, and S. Livengood, eds.). ACM, 2013, 10–17.
- [15] M.H. ter Beek and F. Mazzanti. VMC: Recent Advances and Challenges Ahead. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14), Volume 2* (S. Gnesi, A. Fantechi, M.H. ter Beek, G. Botterweck, and M. Becker, eds.). ACM, 2014, 70–77.
- [16] M.H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A Tool for Product Variability Analysis. In *Proceedings of the 18th International Symposium on Formal Methods (FM'12)* (D. Giannakopoulou and D. Méry, eds.). *Lecture Notes in Computer Science* 7436, Springer, 2012, 450–454.
- [17] M.H. ter Beek and E.P. de Vink. Software Product Line Analysis with mCRL2. In *Proceedings of the 18th International Software Product Line Conference (SPLC'14), Volume 2* (S. Gnesi, A. Fantechi, M.H. ter Beek, G. Botterweck, and M. Becker, eds.). ACM, 2014, 78–85.
- [18] N. Benes, J. Kretínský, Kim G. Larsen, Mikael H. Møller, and J. Srba. Parametric Modal Transition Systems. In *Proceedings of the 9th International Symposium on Automated Technology for Verification and Analysis (ATVA'11)* (T. Bultan and P.-A. Hsiung, eds.). *Lecture Notes in Computer Science* 6996, Springer, 2011, 275–289.
- [19] H. Beohar, M. Varshosaz, and M.R. Mousavi. Basic behavioral models for software product lines: Expressiveness and testing pre-orders. *Science of Computer Programming* (2015). In press.
- [20] G. Bruns and P. Godefroid. Generalized Model Checking: Reasoning about Partial State Spaces. In *Proceedings of the 11th International Conference on Concurrency Theory (CONCUR'00)* (C. Palamidessi, ed.). *Lecture Notes in Computer Science* 1877, Springer, 2000, 168–182.
- [21] A. Cimatti, E.M. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A New Symbolic Model Verifier. In *Proceedings of the 11th International Conference on Computer Aided Verification (CAV'99)* (N. Halbwachs and D. Peled, eds.). *Lecture Notes in Computer Science* 1633, Springer, 1999, 495–499.
- [22] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems* 8, 2 (1986), 244–263.

- [23] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking*. The MIT Press, Cambridge, 1999.
- [24] E.M. Clarke, S. Jha, Y. Lu, and H. Veith. Tree-Like Counterexamples in Model Checking. In *Proceedings of the 17th Annual Symposium on Logic in Computer Science (LICS'02)*. IEEE, 2002, 19–29.
- [25] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model Checking Software Product Lines with SNIP. *International Journal on Software Tools for Technology Transfer* 14, 5 (2012), 589–612.
- [26] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and their Application to LTL Model Checking. *IEEE Transactions on Software Engineering* 39, 8 (2013), 1069–1089.
- [27] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming* 80 (2014), 416–439.
- [28] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*. ACM, 2010, 335–344.
- [29] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. Symbolic Model Checking of Software Product Lines. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. ACM, 2011, 321–330.
- [30] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott (eds.). All About Maude—A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. *Lecture Notes in Computer Science* 4350, Springer, 2007.
- [31] P.C. Clements and L.M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Reading, 2002.
- [32] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: A Product Line of Verifiers for Software Product Lines. In *Proceedings of the 17th International Software Product Line Conference (SPLC'13), Volume 2* (T. Kishi, S. Cohen, and S. Livengood, eds.). ACM, 2013, 141–146.
- [33] R. De Nicola and F.W. Vaandrager. Actions versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes: Proceedings of the Spring School on Theoretical Computer Science* (I. Guessarian, ed.). *Lecture Notes in Computer Science* 469, Springer, 1990, 407–419.
- [34] R. De Nicola and F.W. Vaandrager. Three Logics for Branching Bisimulation. *Journal of the ACM* 42, 2 (1995), 458–487.

- [35] N. D’Ippolito, D. Fischbein, M. Chechik, and S. Uchitel. MTSA: The Modal Transition System Analyser. In *Proceedings of the 23rd International Conference on Automated Software Engineering (ASE’08)*. IEEE, 2008, 475–476.
- [36] A. Fantechi and S. Gnesi. Formal Modelling for Product Families Engineering. In *Proceedings of the 12th Software Product Lines Conference (SPLC’08)* (B. Geppert and K. Pohl, eds.). IEEE, 2008, 193–202.
- [37] A. Fantechi and S. Gnesi. Refinement of Behavioural Models for Variability Description. Chapter 11 in *From Action Systems to Distributed Systems: The Refinement Approach* (L. Petre and E. Sekerinski, eds.). Chapman and Hall/CRC Press, 2016.
- [38] A. Fantechi, S. Gnesi, A. Lapadula, F. Mazzanti, R. Pugliese, and F. Tiezzi. A Logical Verification Methodology for Service-Oriented Computing. *ACM Transactions on Software Engineering and Methodology* 21, 3 (2012), 16:1–16:46.
- [39] H. Fecher and H. Schmidt. Comparing Disjunctive Modal Transition Systems with an One-Selecting Variant. *Journal of Logic and Algebraic Programming* 77, 1–2 (2008), 20–39.
- [40] D. Fischbein, S. Uchitel, and V.A. Braberman. A Foundation for Behavioural Conformance in Software Product Line Architectures. In *Proceedings of the ISSTA’06 Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA’06)* (R.M. Hierons and H. Muccini, eds.). ACM, 2006, 39–48.
- [41] S. Gnesi and F. Mazzanti. On the Fly Verification of Networks of Automata. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’99)*. CSREA, 1999, 1040–1046.
- [42] S. Gnesi and M. Petrocchi. Towards an Executable Algebra for Product Lines. In *Proceedings of the 16th International Software Product Line Conference (SPLC’12), Volume 2* (E. Almeida, C. Schwanninger, and D. Benavides, eds.). ACM, 2012, 66–73.
- [43] M.L. Griss, J. Favaro, and M. d’Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the 5th International Conference on Software Reuse (ICSR’98)*. IEEE, 1998, 76–85.
- [44] J.F. Groote and M.R. Mousavi. Modeling and Analysis of Communicating Systems. The MIT Press, Cambridge, 2014.
- [45] A. Gruler, M. Leucker, and K.D. Scheidemann. Modeling and Model Checking Software Product Lines. In *Proceedings of the 10th IFIP WG 6.1 International Conference on Formal Methods for Open Object-based Distributed Systems (FMODS’08)* (G. Barthe and F.S. de Boer, eds.). *Lecture Notes in Computer Science* 5051, Springer, 2008, 113–131.



- [46] Ø. Haugen and K. Stølen. STAIRS: Steps to Analyze Interactions with Refinement Semantics. In *Proceedings of the 6th International Conference on The Unified Modeling Language (UML'03)* (P. Stevens, J. Whittle, and G. Booch, eds.). *Lecture Notes in Computer Science* 2863, Springer, 2003, 388–402.
- [47] G.J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Reading, 2004.
- [48] A.C. van Hulst, M.A. Reniers, and W.J. Fokkink. Maximal Synthesis for Hennessy-Milner Logic. *ACM Transactions on Embedded Computing Systems* 14, 1 (2015), 10:1–10:21.
- [49] K.C. Kang, S.G. Cohen, J.A. Hess, W.E. Novak, and A.S. Peterson. Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report SEI-90-TR-21. Carnegie Mellon University, 1990.
- [50] D. Kozen. Results on the propositional  $\mu$ -calculus. *Theoretical Computer Science* 27 (1983), 333–354.
- [51] K.G. Larsen and A. Legay. Statistical Model Checking: Past, Present, and Future. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA'14)* (T. Margaria and B. Steffen, eds.). *Lecture Notes in Computer Science* 8802, Springer, 2014, 135–142.
- [52] K.G. Larsen, U. Nyman, and A. Wasowski. Modal I/O Automata for Interface and Product Line Theories. In *Proceedings of the 16th European Symposium on Programming (ESOP'07)* (R. De Nicola, ed.). *Lecture Notes in Computer Science* 4421, Springer, 2007, 64–79.
- [53] K.G. Larsen and B. Thomsen. A Modal Process Logic. In *Proceedings of the 3rd Annual Symposium on Logic In Computer Science (LICS'88)*. IEEE, 1988, 203–210.
- [54] K.G. Larsen and L. Xinxin. Equation Solving Using Modal Transition Systems. In *Proceedings of the 5th Annual Symposium on Logic in Computer Science (LICS'90)*. IEEE, 1990, 108–117.
- [55] K. Lauenroth, K. Pohl, and S. Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proceedings of the 24th International Conference on Automated Software Engineering (ASE'09)*. IEEE, 2009, 269–280.
- [56] M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. DeltaCCS: A Core Calculus for Behavioral Change. In *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation: Technologies for Mastering Change (ISoLA'14)* (T. Margaria and B. Steffen, eds.). *Lecture Notes in Computer Science* 8802, Springer, 2012, 320–335.

- [57] J.-V. Millo, S. Ramesh, S.N. Krishna, and G.K. Narwane. Compositional Verification of Software Product Lines. In *Proceedings of the 10th International Conference on Integrated Formal Methods (IFM'13)* (E. Broch Johnsen, L. Petre, eds.). *Lecture Notes in Computer Science* 7940, Springer, 2013, 109–123.
- [58] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
- [59] R. Muschevici, J. Proença, and D. Clarke. Modular Modelling of Software Product Lines with Feature Nets. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods (SEFM'11)* (G. Barthe, A. Pardo, and G. Schneider, eds.). *Lecture Notes in Computer Science* 7041, Springer, 2011, 318–333.
- [60] M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming* 41, 1 (2001), 53–84.
- [61] K. Pohl, G. Böckle, and F.J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
- [62] I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional Algorithmic Verification of Software Product Lines. In *Proceedings of the 9th International Symposium on Formal Methods for Components and Objects (FMCO'10)* (B.K. Aichernig, F.S. de Boer, and M.M. Bonsangue, eds.). *Lecture Notes in Computer Science* 6957, Springer, 2012, 184–203.
- [63] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature Diagrams: A Survey and a Formal Semantics In *Proceedings of the 14th International Requirements Engineering Conference (RE'06)*. IEEE, 2006, 136–145.
- [64] S. Sebastio and A. Vandin. MultiVeStA: Statistical Model Checking for Discrete Event Simulators. In *Proceedings of the 7th International Conference on Performance Evaluation Methodologies and Tools (ValueTools'13)*, (A. Horvath, P. Buchholz, V. Cortellessa, L. Muscariello, and M.S. Squillante, eds.). ACM, 2013, 310–315.
- [65] S. Soleimanifard, D. Gurov, and M. Huisman. ProMoVer: Modular Verification of Temporal Safety Properties. In *Proceedings of the 9th International Conference on Software Engineering and Formal Methods (SEFM'11)* (G. Barthe, A. Pardo, and G. Schneider, eds.). *Lecture Notes in Computer Science* 7041, Springer, 2011, 366–381.
- [66] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys* 47, 1 (2014), 6:1–6:45.
- [67] M. Tribastone. Behavioral Relations in a Process Algebra for Variants. In *Proceedings of the 18th International Software Product Line Conference*

- (*SPLC'14*) (S. Gnesi, A. Fantechi, P. Heymans, J. Rubin, K. Czarnecki, and D. Dhungana, eds.). ACM, 2014, 82–91.
- [68] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model Checking Programs. *Automated Software Engineering* 10 (2003), 203–232.
- [69] T. Ziadi and J.M. Jézéquel. Software Product Line Engineering with the UML: Deriving Products. In *Software Product Lines: Research Issues in Engineering and Management*. Springer, 2006, 557–588.