A Compositional Framework to Derive Product Line Behavioural Descriptions*

Patrizia Asirelli[1], Maurice H. ter Beek[1],
Alessandro Fantechi[1,2], and Stefania Gnesi[1]

[1] ISTI–CNR, Pisa, Italy
[2] DSI, University of Florence, Italy

Abstract. Modelling variability in product families has been the subject of extensive study in the literature on Software Product Lines, especially that concerning Feature Modelling. In recent years, we have laid the basis for the study of the application of temporal logics to the formal modelling of behavioural variability in product family definitions. A critical point in this formalization is to give an adequate representation of the elements of the feature model and their relation with the behaviour of the many products that are to be derived from the family. To this aim, we propose a methodology to systematize this step as much as possible, in order to allow the derivation of behavioural models that are general enough to capture the behaviour of all consistent products belonging to the family.

1 Introduction

Product Line Engineering (PLE) is a paradigm for the development of a variety of products from a common product platform [23]. Its aim is to lower the production costs of individual products by letting them share an overall reference model of a product family, while allowing them to differ w.r.t. particular features in order to serve, e.g., different markets. Commonality and variability are often defined in terms of *features* and managing variability consists of identifying variation points in a family design as those places where a choice must be made among (optional, mandatory or alternative) features and deciding which combinations of features define valid products. Software Product Line Engineering (SPLE) is a discipline for developing a diversity of software products and software-intensive systems based on the underlying architecture of the product platform [25]. Variability management is what distinguishes SPLE from 'conventional' software engineering.

Since many variability-intensive systems are safety-critical, there is a strong need for rigour and formal modelling and verification (tools). Our contribution in making the development of product families more rigorous consists in an ongoing research effort to investigate upon a suitable formal modelling structure for describing behavioural product variability and a temporal logic than can be

* Research supported by the TRACE-IT project funded by the Tuscany Region under the programme PAR FAS 2007–2013.

T. Margaria and B. Steffen (Eds.): ISoLA 2012, Part I, LNCS 7609, pp. 146–161, 2012.

interpreted over that structure [13,6,7,9]. We opted for Modal Transition Systems (MTSs) [4], which were recognized in [15,20,21] as a useful formal method for describing in a compact way the possible operational behaviour of all products of a product family. We defined a suitable action-based branching-time temporal CTL-like logic over MTSs and developed efficient algorithms to derive valid products from families and to verify properties over products and families alike. We moreover implemented these algorithms in an experimental tool [9].

In this paper, we propose a methodology to systematize the step of formally representing the features from a feature model in relation with the behaviour of the products that can be derived from the family feature model. We also discuss several strategies for the compositional refinement of such behavioural models. We illustrate our approach by means of a simple and intuitive running example.

2 Running Example: A Family of Coffee Machines

For easy comparison, we use the running example from [6,7,11]. It describes a family of (simplified) coffee machines through the following list of requirements:

1. Initially, a coin must be inserted: either a euro, exclusively for European products, or a dollar, exclusively for Canadian products;
2. After inserting a coin, the user has to choose whether (s)he wants sugar, after which (s)he may select a beverage;
3. The choice of beverage (coffee, tea, cappuccino) varies, but coffee must be offered by all products of the family, while cappuccino may be offered solely by European products;
4. Optionally, a ringtone may be rung after delivering a beverage. However, a ringtone must be present in all products offering cappuccino;
5. After the beverage is taken, the machine returns idle.

This list contains a mix of a kind of static constraints defining the differences in configuration (features) between products and more operational constraints defining the behaviour of products through admitted sequences (temporal orderings) of actions/operations implementing features.

The de facto standard variability model in SPLE are *feature diagrams* or *feature models* [19,5] providing compact representations of all products of a product family in terms of their features, and additional constraints among them. Graphically, features are represented as the nodes of a tree, with the family as its root and relations between these features representing constraints. The first three relations below form the tree, the latter two model additional constraints:

Optional features may (but need not) be present only if their parent is present;
Mandatory features are (have to be) present iff their parent is present;
Alternative features are such that only one is present if their parent is present;
Requires is a partial preorder relation indicating that the presence of one feature requires that of the other;
Excludes is a partial symmetric relation indicating the presence of two features to be mutually exclusive.

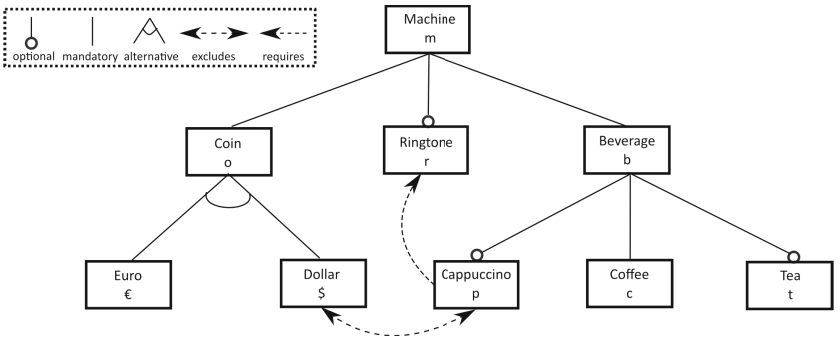


Fig. 1. Feature model of the family of coffee machines (with shorthand names)

Fig. 1 shows a feature model for the coffee machine family, obtained from the list of requirements by considering only the static requirements (1, 3, and part of 4). The behavioural requirements are ignored, since feature models only represent static variability. The feature model defines the following family of 10 products (coffee machines defined by their features):

$\{m, o, b, c, \in\}$,$\{m, o, b, c, \in, r\}$,$\{m, o, b, c, \in, t\}$,$\{m, o, b, c, \in, t, r\}$, $\{m, o, b, c, \in, p, r\}$, $\{m, o, b, c, \$\}$,$\{m, o, b, c, \$, r\}$,$\{m, o, b, c, \$, t\}$,$\{m, o, b, c, \$, t, r\}$,$\{m, o, b, c, \in, p, r, t\}$

A feature model can be characterized by a propositional logic formula [5,26]:

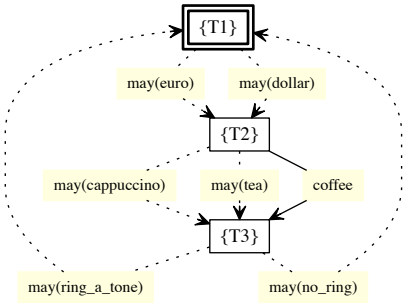
$$(m \iff \mathit{true}) \wedge (o \iff m) \wedge ((\in \iff (\neg\$ \wedge o)) \wedge (\$ \iff (\neg \in \wedge o))) \wedge (r \implies m)$$
$$\wedge (b \iff m) \wedge ((p \implies b) \wedge (c \iff b) \wedge (t \implies b)) \wedge (p \implies r) \wedge (\neg(\$ \wedge p))$$

Suppose we have defined two coffee machines with the following sets of features:

$$\text{CM1} = \{m, o, b, c, \in\} \quad \text{and} \quad \text{CM2} = \{m, o, b, c, \in, p\}$$

Without generating all products, it is easy to see that coffee machine CM1 belongs to the product family since it satisfies the characteristic formula of the feature model, whereas CM2 obviously does not: it falsifies the constraint that a cappuccino requires a ringtone ($p \implies r$). This can be formally verified by interpreting its set of features as a conjunction of axioms ($m \wedge o \wedge b \wedge c \wedge \in \wedge p$) that when added to the characteristic formula makes it either true or false, according to whether or not the product belongs to the family. In general, the problem of finding a product that satisfies the characterization of a feature model is reduced to the problem of finding a satisfying assignment to a set of boolean variables. Efficient SAT solvers can therefore be used to address this kind of problems [5].

In the next sections, we present the way MTSs can provide behavioural descriptions of product families (based on a combination of their initial lists of requirements and their feature models) over which we can then verify temporal properties. To this end, we first provide their features with a temporal ordering and then step-by-step refine the resulting behavioural descriptions.




```
T1 = may(euro).T2 + may(dollar).T2
T2 = coffee.T3 + may(cappuccino).T3
   + may(tea).T3
T3 = may(ring_a_tone).T1 + may(no_ring).T1
net SYS = T1

Constraints {
   euro ALT dollar
   dollar EXC cappuccino
   cappuccino REQ ring_a_tone
   ring_a_tone ALT no_ring }
```

Fig. 2. Coffee machine family: MTS (ℓ) and its textual encoding with constraints (r)

3 Modelling Product Family Behaviour with MTSs

Before defining MTSs, we define their underlying Labelled Transition Systems.

Definition 1. *A* Labelled Transition System *(LTS) is a 4-tuple* $(Q, A, \overline{q}, \delta)$*, with set* Q *of states, set* A *of actions, initial state* $\overline{q} \in Q$*, and transition relation* $\delta \subseteq Q \times A \times Q$*; we may write* $q \xrightarrow{a} q'$ *if* $(q, a, q') \in \delta$*.*

An MTS is an LTS which distinguishes between *may* and *must* transitions.

Definition 2. *A* Modal Transition System *(MTS) is a 5-tuple* $(Q, A, \overline{q}, \delta^{\Diamond}, \delta^{\Box})$ *such that* $(Q, A, \overline{q}, \delta^{\Diamond} \cup \delta^{\Box})$ *is an LTS and* $\delta^{\Box} \subseteq \delta^{\Diamond}$*. An MTS distinguishes the* may *transition relation* $\delta^{\Diamond}$*, expressing* admissible *transitions, and the* must *transition relation* $\delta^{\Box}$*, expressing* necessary *transitions; we may write* $q \xrightarrow{a}_{\Diamond} q'$ *for* $(q, a, q') \in \delta^{\Diamond}$ *and* $q \xrightarrow{a}_{\Box} q'$ *for* $(q, a, q') \in \delta^{\Box}$*.*

The inclusion $\delta^{\Box} \subseteq \delta^{\Diamond}$ formalises that necessary transitions are also admissible. Reasoning on the existence of transitions is like reasoning with a 3-valued logic with truth values *true*, *false*, and *unknown* [16]: necessary transitions ($\delta^{\Box}$) are *true*, admissible but not necessary transitions ($\delta^{\Diamond} \setminus \delta^{\Box}$) are *unknown*, and impossible transitions ($(q, a, q') \notin \delta^{\Box} \cup \delta^{\Diamond}$) are *false*. Graphically, an MTS is a directed edge-labelled graph where nodes model states and edges model transitions: solid edges are necessary ones and dotted edges are admissible but not necessary ones. Edges are labelled with actions executed as the result of state changes. A sequence of state changes is called a path.

An MTS can provide an abstract description of the set of (valid) products of a product family, defining both the behaviour that is common to all products and the behaviour that varies among different products. This requires an interpretation of the requirements of a product family and its constraints w.r.t. certain features as may and must transitions labelled with actions/operations, and a temporal ordering among these transitions.

The methodology we propose in this paper foresees a step-by-step approach, initiated by ordering the features of the feature model. For our running example, the first step of this methodology results in the MTS depicted in Fig. 2(ℓ).

The standard derivation methodology for obtaining a product (which becomes an LTS) from an MTS modelling a product family is defined as including all its (reachable) must transitions and a subset of its (reachable) may transitions; each selection is a product. Unfortunately, MTSs alone are incapable of modelling all common variability constraints. While an MTS is apparently able to model the constraints concerning optional and mandatory features, by means of may and must transitions, no MTS is able to model the constraints regarding alternative features nor those regarding the requires and excludes inter-feature relations. The solution elaborated in [6,7] is to enrich the MTS description with a set of constraints that allow one to define which of the standardly derivable products should be considered as acceptable valid products. In particular, an appropriate variability and action-based temporal logic to formalize these constraints is defined in [6] and an algorithm to derive all and only LTSs describing valid products in [7]. For now, we consider three kinds of (binary[1]) constraints:

F1 ALT F2 Features F1 and F2 are *alternative*;
F1 EXC F2 Feature F1 *excludes* feature F2;
F1 REQ F2 Feature F1 *requires* feature F2.

Their intuitive meaning is as expected: if F1 and F2 are alternative, then all valid products must contain either F1 or F2, but not both; if F1 requires (excludes) F2, then a product which contains F1 must (may not) contain F2. These constraints allow us to define in more detail the set of valid products derivable from the MTS of Fig. 2(ℓ), namely those satisfying each of the constraints specified in Fig. 2(r). Note, however, that these constraints do not imply a temporal ordering among the involved features: a coffee machine that rings a tone before delivering a cappuccino cannot be excluded as a product of the family based on the constraint `cappuccino REQ ring_a_tone`. Such orderings are imposed by the associated behavioural description of a product (family) in the form of an LTS (MTS).

4 Generating and Analyzing Valid Products with VMC

We implemented the above solution in an experimental tool for the modelling and analysis of variability in product lines: the *Variability Model Checker* VMC [9]. Given a product family specified as an MTS, possibly with additional variability constraints, it can automatically generate all the valid products of a family (according to the given constraints), visualize the family/products as MTS/LTSs, and efficiently model check properties expressed in an action- and state-based branching-time temporal CTL-like logic over products and families alike.

VMC takes as input the textual encoding of an MTS in the form of a simple process algebra and an additional set of constraints of the form ALT, EXC, REQ, and IFF (a shorthand for bilateral REQs). The distinction among may and must transitions is encoded in the resulting LTS by structuring action labels of may transitions as `may(·)` (i.e., typed actions). The LTS modelling an MTS through

[1] We plan to extend our approach and tool to deal also with n-ary constraints.

typed actions shown in Fig. 2(ℓ) is in fact generated by VMC taking as input the textual representation (without the constraints) depicted in Fig. 2(r).

VMC implements the algorithm in [7] to generate all valid products derivable from an MTS when an associated set of constraints is taken into account. Beyond generating all valid products (LTSs), VMC allows browsing them, verifying whether they satisfy a certain property (a logic formula) and investigating why a specific valid product does (not) satisfy the verified property. To do so, for each product a new window with its textual encoding can be opened.

From the MTS defined in Fig. 2(r) VMC indeed generates the 10 products of the family defined by the feature model depicted in Fig. 1, listing for each product moreover which admitted but not necessary (may) transitions it contains:

```
product14-dollar-ring_a_tone            product23-euro-cappuccino-tea-ring_a_tone
product15-dollar-no_ring                product26-dollar-tea-ring_a_tone
product17-euro-cappuccino-ring_a_tone   product27-dollar-tea-no_ring
product20-euro-tea-ring_a_tone          product8-euro-ring_a_tone
product21-euro-tea-no_ring              product9-euro-no_ring
```

Clicking on a product, its specification appears in a new window. For instance:

```
-----------------------------------     ------------------------------------------------
-- product14-dollar-ring_a_tone         -- product23-euro-cappuccino-tea-ring_a_tone
-----------------------------------     ------------------------------------------------
T1 = dollar.T2                          T1 = euro.T2
T2 = coffee.T3                          T2 = coffee.T3 + tea.T3 + cappuccino.T3
T3 = ring_a_tone.T1                     T3 = ring_a_tone.T1
net SYS = T1                            net SYS = T1
```

After having provided an initial temporal ordering of the features, the next step of our methodology is to start refining the behavioural description of a family through the addition of more detailed (operational) information. In our running example, we consider the possibility of distinguishing beverages with and without sugar, as well as different ways of actually mixing the ingredients of beverages:

```
T1 = may(euro).T2 + may(dollar).T2
T2 = coffee.T3 + may(cappuccino).T4 + may(tea).T5
T3 = may(pour_sugar).T6 + pour_coffee.T11
T4 = may(pour_sugar).T7 + pour_coffee.T9 + pour_milk.T10
T5 = may(pour_sugar).T8 + pour_tea.T11
T6 = pour_coffee.T11
T7 = pour_coffee.T9 + pour_milk.T10
T8 = pour_tea.T11
T9 = pour_milk.T11
T10 = pour_coffee.T11
T11 = may(ring_a_tone).T12 + may(no_ring).T12
T12 = take_cup.T1
net SYS = T1
```

```
Constraints {
   euro ALT dollar
   dollar EXC cappuccino
   cappuccino REQ ring_a_tone
   ring_a_tone ALT no_ring
}
```

Given this refined family, VMC generates a total of 36 products. This explosion is due to the way we introduced the possibility of pouring sugar into a beverage, allowing coffee machines in which certain beverages are offered only sugared and others only unsugared. For instance, it allows the two products below: in one only coffee can be sugared while in the other only tea can be sugared. The MTSs of these two products as generated by VMC are shown in Fig. 3; they are obtained by taking as input the following textual representations:

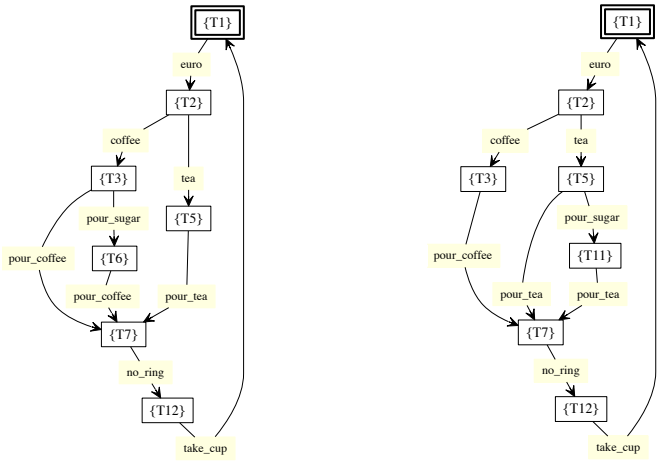


Fig. 3. Valid products of the refined family model as generated by VMC

```
-------------------------------------------
-- product55-euro-tea-pour_sugar-no_ring
-------------------------------------------
T1 = euro.T2
T2 = coffee.T3 + tea.T5
T3 = pour_coffee.T7 + pour_sugar.T6
T5 = pour_tea.T7
T6 = pour_coffee.T7
T7 = no_ring.T12
T12 = take_cup.T1
net SYS = T1
```

```
-------------------------------------------
-- product58-euro-tea-pour_sugar-no_ring
-------------------------------------------
T1 = euro.T2
T2 = coffee.T3 + tea.T5
T3 = pour_coffee.T7
T5 = pour_tea.T7 + pour_sugar.T11
T7 = no_ring.T12
T11 = pour_tea.T7
T12 = take_cup.T1
net SYS = T1
```

There are several ways of resolving this, assuming that coffee machines that offer sugared versions of only some of the available beverages is not what we want. One way, leading to the model of [6,7] depicted in Fig. 4, is to further refine the family by explicitly modelling the choice for sugar upfront (in line with the requirements), so distinguishing the beverages being sugared, and to extend the constraints enforcing that all available beverages may but need not be sugared:

```
T1 = may(euro).T2 + may(dollar).T2
T2 = sugar.T3 + no_sugar.T4
T3 = sugared_coffee.T5
   + may(sugared_cappuccino).T6
   + may(sugared_tea).T7
T4 = unsugared_coffee.T8
   + may(unsugared_cappuccino).T9
   + may(unsugared_tea).T10
T5 = pour_sugar.T8
T6 = pour_sugar.T9
T7 = pour_sugar.T10
T8 = pour_coffee.T13
T9 = pour_coffee.T11 + pour_milk.T12
T10 = pour_tea.T13
T11 = pour_milk.T13
T12 = pour_coffee.T13
T13 = may(ring_a_tone).T14 + may(no_ring).T14
T14 = take_cup.T1
net SYS = T1
```

```
Constraints {
   euro ALT dollar
   sugared_tea IFF unsugared_tea
   sugared_cappuccino IFF unsugared_cappuccino
   dollar EXC sugared_cappuccino
   sugared_cappuccino REQ ring_a_tone
   ring_a_tone ALT no_ring
}
```

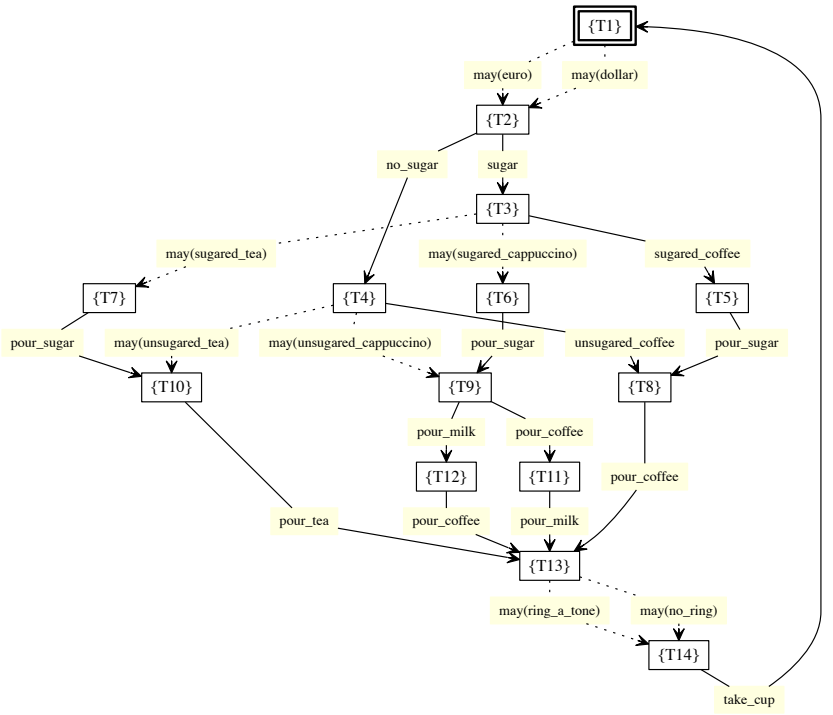


Fig. 4. MTS of refined coffee machine family as generated by VMC

From this refined family MTS, VMC generates the following 10 product LTSs:

```
product11-euro-ring_a_tone
product12-euro-no_ring
product20-dollar-ring_a_tone
product21-dollar-no_ring
product65-euro-sugared_cappuccino-unsugared_cappuccino-ring_a_tone
product77-euro-sugared_tea-unsugared_tea-ring_a_tone
product78-euro-sugared_tea-unsugared_tea-no_ring
product89-euro-sugared_cappuccino-sugared_tea-unsugared_cappuccino-unsugared_tea-ring_a_tone
product95-dollar-sugared_tea-unsugared_tea-ring_a_tone
product96-dollar-sugared_tea-unsugared_tea-no_ring
```

Rather than clicking on the products to analyze them, we can use the model-checking features of VMC to verify whether all valid European products offer both sugared and unsugared cappuccino by checking the following logic formula:[2]

$$[euro]\,((EF\,\langle sugared\_cappuccino\rangle\,true)\;and\;EF\,\langle unsugared\_cappuccino\rangle\,true)$$

VMC produces a table of the above 10 products listing whether or not they satisfy this formula (recall that cappuccino is optional even for European products):

[2] Operators $\langle\,\rangle$ ("possibly") and $[\,]$ ("necessarily") are the classic diamond and box modalities, while EF ("eventually") is a combination of the classic existential path operator E ("exists") and the classic state operator F ("future").

```
product11-euro-ring_a_tone                                          Formula is FALSE
product12-euro-no_ring                                              Formula is FALSE
product20-dollar-ring_a_tone                                        Formula is TRUE
product21-dollar-no_ring                                            Formula is TRUE
product65-euro-(un)sugared_cappuccino-ring_a_tone                   Formula is TRUE
product77-euro-sugared_tea-unsugared_tea-ring_a_tone                Formula is FALSE
product78-euro-sugared_tea-unsugared_tea-no_ring                    Formula is FALSE
product89-euro-(un)sugared_cappuccino-(un)sugared_tea-ring_a_tone   Formula is TRUE
product95-dollar-sugared_tea-unsugared_tea-ring_a_tone              Formula is TRUE
product96-dollar-sugared_tea-unsugared_tea-no_ring                  Formula is TRUE
```

Likewise we can verify whether all valid products offer both sugared and unsugared coffee, in which case VMC reports that this property holds for all 10 products:

$$[euro]\,((EF\,\langle sugared\_coffee\rangle\,true)\;and\;EF\,\langle unsugared\_coffee\rangle\,true)$$

5 Compositional Modelling of Feature Models and MTSs

The step-by-step refinement described in the previous sections is one way to build a large behavioural model of a product family in a bottom-up fashion: a minimal feature model leads to an initial MTS which is subsequently refined by repeatedly adding functionality. Two alternative strategies are based on composition: several minimal feature models, representing different functionalities, are first composed and then interpreted as MTSs or first interpreted and then composed. These strategies require two compositional operators, namely one for feature models and one for MTSs. For feature models, we can use the feature model composition operators described in [1], while for MTSs we can use the classic process-algebraic choice operator (+) and apply it to their textual encodings. Ideally, all strategies should lead to the same refined model. We show that this is not yet the case by applying also the latter two strategies to our running example.

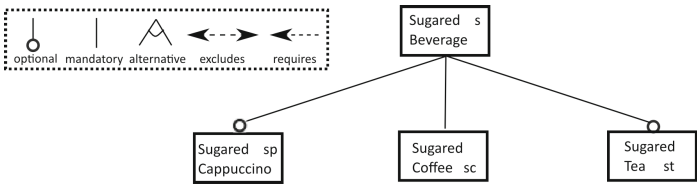


Fig. 5. Sugared Beverage aspect feature model for sugared beverages

Consider the feature model of Fig. 1 and suppose that we want to explicitly add to this configuration the aspect (taken from the list of requirements) that coffee machines come equipped with a functionality that allows the user to choose between sugared and unsugared beverages. Following [1], we can do this by defining the so-called aspect feature model depicted in Fig. 5 and compose it with the feature model depicted in Fig. 1 through the so-called insert operator:

$$\texttt{insert}(\texttt{Sugared Beverage}, \texttt{Beverage}, \texttt{And-Mandatory})$$

Sugared Beverage is the feature to insert (from the aspect feature model), Beverage the target feature (in the base feature model), and And-Mandatory the operator (i.e., the relation defining how the feature is to be included in the tree).

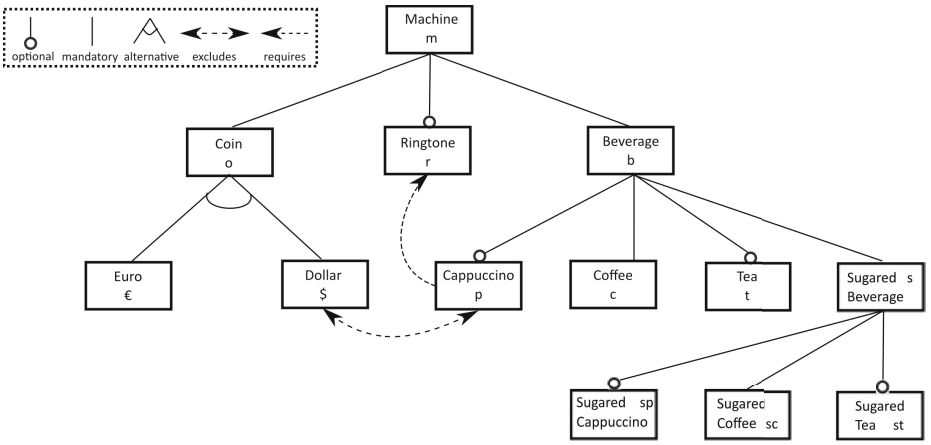


Fig. 6. Aspect inserted into feature model of the family of coffee machines

This composition by insertion results in the feature model depicted in Fig. 6. As before, our methodology now advocates a step-by-step interpretation of the feature model, initially providing a mere ordering of the features. This may result in the MTS depicted in Fig. 7 and its following process-algebraic representation:

```
T1 = may(euro).T2 + may(dollar).T2
T2 = sugar.T3 + no_sugar.T4
T3 = sugared_coffee.T5 + may(sugared_cappuccino).T5
   + may(sugared_tea).T5
T4 = unsugared_coffee.T5 + may(unsugared_cappuccino).T5
   + may(unsugared_tea).T5
T5 = may(ring_a_tone).T1 + may(no_ring).T1
net SYS = T1
```

```
Constraints {
   euro ALT dollar
   dollar EXC cappuccino
   cappuccino REQ ring_a_tone
   ring_a_tone ALT no_ring
}
```

Now we run into a problem that is mentioned as future work in [1] and also in later work by the same authors and which — to the best of our knowledge — has not yet been solved: the current composition operators for feature models do not consider inter- and intra-feature constraints between features (such as, e.g., the requires and excludes relations). Translated into our example: composing the feature models depicted in Figs. 5 and 6 should ideally result in a composition (feature model) that incorporates the following constraints:

```
sugared_tea IFF unsugared_tea          sugared_cappuccino IFF unsugared_cappuccino
```

These constraints serve to guarantee that whenever a specific beverage is offered in a valid coffee machine, it can always be obtained sugared as well as unsugared. While this hopefully can be done automatically in the future, for now we have no other choice than to add these constraints by hand.

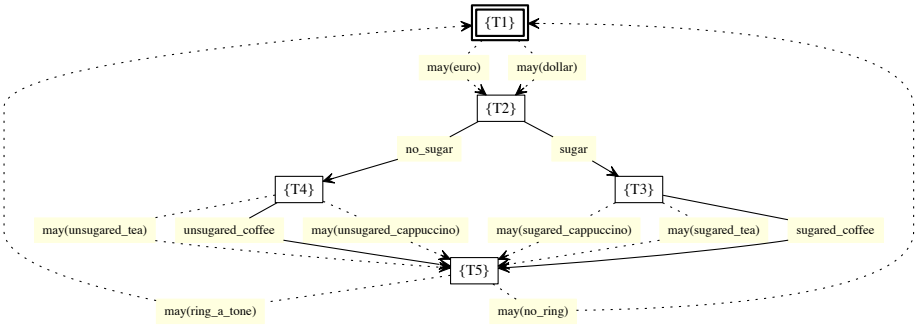


Fig. 7. Coffee machine family interpretation of the feature model of Fig. 6

The next step of our methodology is to refine this behavioural description of the family by adding more detailed (operational) information. If we consider, as before, the different ways of actually mixing the ingredients of the beverages, then this may lead us — once more — to the model of [6,7] depicted in Fig. 4.

This concludes one of the two alternative bottom-up strategies based on composition mentioned in the beginning of this section, namely first composing minimal feature models, representing different functionalities, and then interpreting the resulting composition as an MTS.

The remaining strategy (first interpreting the minimal feature models as MTSs, and then composing these) requires a compositional operator for MTSs.

Consider the MTS and its textual encoding depicted in Fig. 2 (interpreting the feature model depicted in Fig. 1). As before, we want to add the aspect that coffee machines come equipped with a functionality that allows the user to choose between sugared and unsugared beverages as represented by the aspect feature model depicted in Fig. 5. The latter can be interpreted as follows:

```
T = sugared_coffee.Tx + may(sugared_cappuccino).Ty + may(sugared_tea).Tz
```

To compose this with the textual encoding, depicted in Fig. 2(r), of the MTS depicted in Fig. 2(ℓ), we use the well-known process-algebraic choice operator:[3]

```
T1 = may(euro).T2 + may(dollar).T2
T2 = sugar.T3 + no_sugar.T4
T3 = sugared_coffee.T5 + may(sugared_cappuccino).T5
   + may(sugared_tea).T5
T4 = coffee.T5 + may(cappuccino).T5 + may(tea).T5
T5 = may(ring_a_tone).T1 + may(no_ring).T1
net SYS = T1
```

```
Constraints {
   euro ALT dollar
   dollar EXC cappuccino
   cappuccino REQ ring_a_tone
   ring_a_tone ALT no_ring
}
```

In this way, we interpret properly the functionality (originating from the list of requirements) that allows the user to choose between sugared and unsugared beverages after having inserted a coin. However, as before, we need to add constraints to guarantee that whenever a specific beverage is offered in a valid coffee machine, it can always be obtained sugared as well as unsugared.

[3] `a.Tx + b.Ty` represents a system which may behave either as `a.Tx` or as `b.Ty`.

After having done so, the next step of our methodology would be — once again — to refine this behavioural description of the family by adding more detailed (operational) information. If we consider, as before, the different ways of actually mixing the ingredients of the beverages, then this may lead us — once again — to the model of [6,7] depicted in Fig. 4.

This concludes the second of two alternative bottom-up strategies based on composition mentioned in the beginning of this section, namely first interpreting minimal feature models, representing different functionalities, as textual encodings of MTSs and then composing the resulting textual encodings into one textual encoding of an MTS.

6 Getting Acquainted with VMC

The core of VMC consists of a command-line-oriented version of the model checker and by a product generation procedure. These programs are stand-alone executables written in Ada and can easily be compiled for the Windows / Linux / Solaris / MacOSX platforms. These core executables are wrapped with a set of CGI scripts handled by a web server; in this way, a graphical html-oriented GUI can easily be built, and the integration with other tools for LTS minimization and graph drawing is easily achieved. (Cf. [9] for further details and references.)

The development of VMC is still in progress, but a prototypical version of the tool is being used at ISTI–CNR for academic and experimental purposes. VMC is publicly usable online (`http://fmtlab.isti.cnr.it/vmc/`) and its executables are available upon request. The reader is warmly invited to experiment with VMC. The definition of the refined model of the running example used in this paper is available as `coffeemodel2.txt` from one of the examples.

The current version of VMC is not targeted to the verification of very large systems. Its main limitation, however, lies in generating the model from its process-algebraic input language, while its on-the-fly verification engine and advanced explanation techniques are those of the highly optimized family of on-the-fly model checkers developed during the last decades at ISTI–CNR [8,14]. The on-the-fly nature of their underlying model-checking algorithms means that in general not the whole state space needs to be generated and explored. This feature improves performance and allows to deal with infinite-state systems.

7 Related Work

We first discuss work related to our behavioural modelling and analysis framework based on MTSs and temporal logic, after which we discuss work related to the compositional modelling approach of Archer et alii [1,2,3] that we adopted in the compositional framework presented in this paper.

Behavioural framework. The approach closest to ours is that based on Featured Transition Systems (FTSs) [11,12]. An FTS is a (doubly-labelled) transition system with an associated feature diagram and a specific distinction among its

transitions by means of a labelling indicating which transitions correspond to which features. This approach, like ours, thus models product families in terms of specific transition systems that define family behaviour in terms of actions (features). Likewise, both approaches require the addition of further structural relationships between actions to manage (advanced) variability constraints.

In [11], an explicit-state model-checking technique, progressing one state at a time, to verify Linear Temporal Logic (LTL) properties over FTSs is defined. This results in 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 so are the products violating the property. In [12], this approach is improved by using symbolic model checking, examining sets of states at a time, and a feature-oriented version of classic CTL (Computation Tree Logic).

SNIP [10] is a model checker for product families modelled as FTSs specified in a language based on that of the well-known SPIN (`http://spinroot.com/`) model checker. Features are declared in the Text-based Variability Language TVL and are taken into account by the explicit-state model-checking algorithm of SPIN for verifying properties expressed in fLTL (feature LTL) interpreted over FTSs (e.g., to verify a property only over a subset of valid products). Exhaustive model-checking algorithms (continuing their search after a violation was found) moreover allow to verify all the products of a family at once and to output all the products that violate a property. Unlike VMC, SNIP is a command-line tool with no graphical interface. Moreover, it was built from scratch, while VMC profits from numerous optimization techniques that were implemented over the years in the family of on-the-fly model checkers to which it belongs (cf. [8]). SNIP, however, treats features as first-class citizens, with built-in support for feature diagrams, and implements model-checking algorithms tailored for product families.

As said before, MTSs were recognized as a suitable behavioural model to describe product families in [15,20,21]. In [15], a fixed-point algorithm, implemented in a tool, is defined to check whether an LTS conforms to an MTS w.r.t. several different branching relations. In the context of SPLE, it allows to check the conformance of the behaviour of a product against that of its product family.

In [21], variable I/O automata are introduced to model product families. Like modal I/O automata [20], they extend I/O automata with a distinction among may and must transitions. A model-checking approach to verify conformance of products w.r.t. the variability of a family is also defined. 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 can be verified through explicit-state model checking.

Finally, in [22] an algebraic approach to behavioural modelling and analysis of product families is described, while feature Petri nets are introduced in [24] to model behaviour of product families with a high degree of variability.

Compositional framework. The compositional framework presented in this paper is based on the compositional feature modelling approach of [1]. Archer et alii define a number of operators to separate, relate and compose feature models and

semantic properties that must be preserved during such (de)compositions. These operators include the `insert` operator used in Sect. 5 and the more generic `merge` operator built on top of it, as well as the `slice` decomposition operator. To support the manipulation of feature models, they moreover developed the domain-specific language FAMILIAR [3]. For a systematic overview and comparison of their approach with a number of related approaches, we refer to [2]. One of its conclusions is that generic model composition frameworks are outperformed by domain-specific approaches, which convinces us to pursue the development of the compositional framework proposed in this paper.

Two more related domain-specific component-based development approaches are constraint-oriented variability modelling [28], in which behavioural models are constructed by iteratively refining the constraints to determine the admissible solutions, and hierarchical variability modelling [18], which integrates component variability and component hierarchy and is equipped with compositional LTL-based verification techniques for SPL behaviour implemented in a tool set [27]. Compared to our approach, the former uses a top-down rather than a bottom-up approach, while the latter is an architectural rather than behavioural approach to variability modelling.

Finally, in [17] a feature-oriented approach to modelling product families in Event-B by means of a chain of refinements is explored by applying existing Event-B (de)composition techniques to two case studies, using a prototypical feature composition tool. Behavioural variability is not considered, but it would be interesting to explore the feasibility of using this *Feature Event-B* as a high-level specification language on top of the semantic model of our approach.

8 Conclusions and Future Work

We have proposed and illustrated three different methodologies for the derivation of product line behavioural descriptions from feature models, one through refinement and two by means of composition. To complete the variant based on composing MTSs, we are currently working out the details of domain-specific composition operators for MTSs. Subsequently, we intend to implement the resulting compositional framework in VMC.

References

1. Acher, M., Collet, P., Lahire, P., France, R.: Composing Feature Models. In: van den Brand, M., Gašević, D., Gray, J. (eds.) SLE 2009. LNCS, vol. 5969, pp. 62–81. Springer, Heidelberg (2010)
2. Acher, M., Collet, P., Lahire, P., France, R.: Comparing Approaches to Implement Feature Model Composition. In: Kühne, T., Selic, B., Gervais, M.-P., Terrier, F. (eds.) ECMFA 2010. LNCS, vol. 6138, pp. 3–19. Springer, Heidelberg (2010)
3. Acher, M., Collet, P., Lahire, P., France, R.B.: Managing Feature Models with FAMILIAR: a Demonstration of the Language and its Tool Support. In: Heymans, P., Czarnecki, K., Eisenecker, U.W. (eds.) Proceedings 5th Workshop on Variability Modelling of Software-intensive Systems (VaMoS 2011), pp. 91–96. ACM (2011)

4. Antonik, A., Huth, M., Larsen, K.G., Nyman, U., Wąsowski, A.: 20 Years of Modal and Mixed Specifications. Bulletin of the EATCS 95, 94–129 (2008)
5. Batory, D.: Feature Models, Grammars, and Propositional Formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, pp. 7–20. Springer, Heidelberg (2005)
6. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: A Logical Framework to Deal with Variability. In: Méry, D., Merz, S. (eds.) IFM 2010. LNCS, vol. 6396, pp. 43–58. Springer, Heidelberg (2010)
7. Asirelli, P., ter Beek, M.H., Fantechi, A., Gnesi, S.: Formal Description of Variability in Product Families. In: Proceedings 15th International Software Product Line Conference (SPLC 2011), pp. 130–139. IEEE (2011)
8. ter Beek, M.H., Fantechi, A., Gnesi, S., Mazzanti, F.: A state/event-based model-checking approach for the analysis of abstract system properties. Science of Computer Programming 76(2), 119–135 (2011)
9. ter Beek, M.H., Mazzanti, F., Sulova, A.: VMC: A Tool for Product Variability Analysis. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 450–454. Springer, Heidelberg (2012)
10. Classen, A., Cordy, M., Heymans, P., Schobbens, P.-Y., Legay, A.: SNIP: An Efficient Model Checker for Software Product Lines. Technical Report P-CS-TR SPLMC-00000003, PReCISE Research Center, University of Namur (2011)
11. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., Raskin, J.-F.: Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines. In: Proceedings 32nd International Conference on Software Engineering (ICSE 2010), pp. 335–344. ACM (2010)
12. Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A.: Symbolic Model Checking of Software Product Lines. In: Proceedings 33rd International Conference on Software Engineering (ICSE 2011), pp. 321–330. ACM (2011)
13. Fantechi, A., Gnesi, S.: Formal Modelling for Product Families Engineering. In: Proceedings 12th Software Product Lines Conference (SPLC 2008), pp. 193–202. IEEE (2008)
14. Fantechi, A., Gnesi, S., Lapadula, A., Mazzanti, F., Pugliese, R., Tiezzi, F.: A Logical Verification Methodology for Service-Oriented Computing. ACM Transactions on Software Engineering and Methodology 21(3), article 16, 1–46 (2012)
15. Fischbein, D., Uchitel, S., Braberman, V.A.: A Foundation for Behavioural Conformance in Software Product Line Architectures. In: Hierons, R.M., Muccini, H. (eds.) Proceedings ISSTA 2006 Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA 2006), pp. 39–48. ACM (2006)
16. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-Based Model Checking Using Modal Transition Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, pp. 426–440. Springer, Heidelberg (2001)
17. Gondal, A., Poppleton, M., Butler, M.: Composing Event-B Specifications - Case-Study Experience. In: Apel, S., Jackson, E. (eds.) SC 2011. LNCS, vol. 6708, pp. 100–115. Springer, Heidelberg (2011)
18. Haber, A., Rendel, H., Rumpe, B., Schaefer, I., van der Linden, F.: Hierarchical Variability Modeling for Software Architectures. In: Proceedings 15th International Software Product Line Conference (SPLC 2011), pp. 150–159. IEEE (2011)
19. Kang, K., Choen, S., Hess, J., Novak, W., Peterson, S.: Feature Oriented Domain Analysis (FODA) Feasibility Study. Technical Report SEI-90-TR-21. Carnegie Mellon University (1990)

20. Larsen, K.G., Nyman, U., Wąsowski, A.: Modal I/O Automata for Interface and Product Line Theories. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 64–79. Springer, Heidelberg (2007)
21. Lauenroth, K., Pohl, K., Töhning, S.: Model Checking of Domain Artifacts in Product Line Engineering. In: Proceedings 24th International Conference on Automated Software Engineering (ASE 2009), pp. 269–280. IEEE (2009)
22. Leucker, M., Thoma, D.: A Formal Approach to Software Product Families. In: Margaria, T., Steffen, B.(eds.) ISoLA 2012, Part I. LNCS, vol. 7609, pp. 131–145. Springer, Heidelberg (2012)
23. Meyer, M.H., Lehnerd, A.P.: The Power of Product Platforms: Building Value and Cost Leadership. The Free Press (1997)
24. Muschevici, R., Clarke, D., Proença, J.: Feature Petri Nets. Schaefer, I., Carbon, R., (eds): Proceedings 1st Workshop on Formal Methods in Software Product Line Engineering (FMSPLE 2010). Technical Report, University of Lancaster (2010)
25. Pohl, K., Böckle, G., van der Linden, F.: Software Product Line Engineering: Foundations, Principles, and Techniques. Springer (2005)
26. Roos-Frantz, F.: Automated Analysis of Software Product Lines with Orthogonal Variability Models: Extending the FaMa Ecosystem. Ph.D. Thesis, University of Seville (2012)
27. Schaefer, I., Gurov, D., Soleimanifard, S.: Compositional Algorithmic Verification of Software Product Lines. In: Aichernig, B.K., de Boer, F.S., Bonsangue, M.M. (eds.) FMCO 2011. LNCS, vol. 6957, pp. 184–203. Springer, Heidelberg (2011)
28. Schaefer, I., Lamprecht, A.-L., Margaria, T.: Constraint-oriented Variability Modeling. In: Proceedings 34th Annual IEEE Software Engineering Workshop (SEW 2011), pp. 77–83. IEEE (2012)