Using mCRL2 for the Analysis of Software Product Lines

(extended version)

Maurice H. ter Beek
ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it

Erik P. de Vink
Eindhoven University of Technology &
CWI, Amsterdam, The Netherlands
evink@win.tue.nl

ABSTRACT

We show how the formal specification language mCRL2 and its state-of-the-art toolset can be used successfully to model and analyze variability in software product lines. The mCRL2 toolset supports parametrized modeling, model reduction and quality assurance techniques like model checking. We present a proof-of-concept, which moreover illustrates the use of data in mCRL2 and also how to exploit its data language to manage feature attributes of software product lines and quantitative constraints between attributes and features.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

General Terms

Experimentation,Verification

Keywords

Model checking, product lines, variability analysis, mCRL2

1. INTRODUCTION

The last decades have witnessed a paradigm shift from mass production to mass customization in order to serve as many individual customer needs as possible. This has led to the emergence of Software Product Line Engineering (SPLE) [22], a software engineering approach aimed at cost-effectively developing a variety of software-intensive products that share a reference model, i.e. that together form a (software) product line. In SPLE, commonality and variability are defined in terms of features, which may be mandatory, optional or alternative, and managing variability means identifying variation points in a shared family model to encode exactly those combinations of features concerning valid products. Actual product configuration during application engineering is then reduced to selecting desired options in the variability model.

The most prominent variability model, a feature diagram, is a compact representation of all products of a product line in terms of features [24]. Graphically, features are nodes of a rooted tree and relations between them model constraints (mandatory, optional or alternative, but also, e.g., mutually exclusive). However, there may be hundreds of features, requiring models with hundreds of options, which easily leads to anomalies like superfluous or—worse—contradictory variability information (e.g. false optional or dead features). There is a large body of literature on computer-aided analyses of variability models to extract valid products and detect anomalies [5]. None of these analyses consider behavioral variability, though, meaning that only the presence of features is measured, not their causality or ordering in time.

Formal methods have successfully been applied in single product engineering for decades now with the aim of rigorously establishing critical system requirements. In SPLE, on the contrary, formal methods are not exploited that broadly, despite their potential to detect anomalies and to improve product quality. One reason is that mainstream formal methods do not consider variability directly. Still, in SPLE the correctness of artifacts intended for reuse as well as the correctness of the developed products is of crucial importance since many of them concern massively produced and safety- or business-critical applications. Remarkably, formal methods that have been applied in SPLE mainly focus on structural rather than behavioral properties.

To lay a basis for a formal analysis of product lines that does take behavior into account, it is important to formally model behavioral variability. After this was first recognized in the context of UML [17, 27], it has caused a growing interest in behavioral variability and the tailoring of several behavioral models for SPLE, which has given rise to variants of transition systems [11, 18, 19, 1, 7], process algebras [16, 12, 3], Petri nets [21], Event-B models [15] and state machines [20]. As a result, behavioral analysis techniques like model checking are recently being deployed for the verification of temporal properties over product lines [6, 4, 20].

In this paper, in line with the analysis recommendations of [2], we report on the feasibility of using the mCRL2 toolset for the analysis of software product lines. Whenever appropriate, connections with related approaches are mentioned, but a detailed comparison is left for future work. mCRL2 [10] is a formal, process-algebraic specification language with an associated industrial-strength toolset [13], specifically designed to reason on distributed and concurrent systems. It has been developed, improved and actively maintained for several years now and it strives for maximal user influence at every step during modeling and verification. System analysis with mCRL2 has been successfully applied in a wide range of academic and industrial case studies. Furthermore, mCRL2's modeling language supports user-defined abstract datatypes that can be exploited to deal with feature attributes and associated quantitative constraints.

The body of this technical report appears in the Proceedings of the 2nd FME Workshop on Formal Methods in Software Engineering (FormaliSE'14), Hyderabad, India, IEEE. This extended version contains the full mCRL2 code in an appendix.

In this paper, we present our ideas through a toy example, but we have started to work on a larger industrial case study that we hope to present in the near future. The contribution of this paper is a proof-of-concept for variability analysis using the mCRL2 toolset. Sect. 2 introduces basic notions of variability modeling and our flavor of coffee machine used as a running example. Sect. 3 provides some details on mCRL2 and prepares the ground for Sect. 4, where the basic setup of the approach is sketched. In particular, we discuss mCRL2 modeling of feature selection and analysis for the example product line using model checking of properties in the modal μ-calculus. Having added pointers to related work on-the-fly mostly, Sect. 5 wraps up with concluding remarks.

2. RUNNING EXAMPLE PRODUCT LINE

Our running example is an extension of the family of coffee machines from [1]. It has the following list of requirements:

- Initially, money must be inserted: either at least one euro's worth in coins, exclusively for European products, or at least one dollar's worth in coins, exclusively for Canadian products.
- Input of money can be canceled via a cancel button. Optionally, the machine returns change after more than one euro or one dollar was inserted.
- Once the machine contains at least one euro or one dollar, the user has to choose whether (s)he wants sugar, by pressing one of two buttons, after which (s)he can select a beverage.
- The choice of beverage (coffee, tea, cappuccino) varies, but coffee must be offered by all products whereas cappuccino may be offered solely by European products.
- Optionally, a ringtone may be rung after delivering a beverage. However, a ringtone must be rung by all products offering cappuccino.
- After the beverage is taken, the machine returns idle.

A *feature diagram* is an and/or-hierarchy of features of a product line, which regulates their presence in products: *optional* features may be present provided their parent is, *mandatory* features must be present provided their parent is, and exactly one *alternative* feature must be present provided their parent is. We speak of a *feature model* when a feature diagram is moreover equipped with *cross-tree constraints*: a *requires* constraint indicates that the presence of a feature requires that of another, and an *excludes* constraint indicates that two features are mutually exclusive. Finally, by adding (non-functional) attributes (e.g. $cost(tea) = 3$) to features and quantitative constraints (e.g. $cost(Machine) \leqslant 30$) we obtain an *attributed feature model*.

Figure 1 depicts the attributed feature model of the product line for our example family of coffee machines, involving the root feature M and the set $Feature$ consisting of the 10 non-trivial features S, O, R, B, X, E, D, P, C, and T. As usual, we identify a product from the product line with a non-empty subset of $Feature$ united with the root feature. The cost function $cost : Feature \to \mathbb{N}$, associated to the attribute *cost*, extends to products straightforwardly: $cost(product) = \sum\{\, cost(feature) \mid feature \in product \,\}$.

The example only involves binary cross-tree constraints, non-interacting feature-wise quantifiable attributes and a single optimization objective, viz. $cost(Machine) \leqslant 30$. More general and complex constraints, properties and objectives can however be treated as well [5, 25]. The feature diagram, i.e. ignoring the cross-tree constraints, gives rise to 2^5 valid products out of the $2^{10} - 1$ possible non-empty feature sets. The feature model reduces this number to 20, while it is further reduced to 16 valid products if the attributed feature model is considered (e.g. $cost(\{M, S, O, R, B, X, E, C, T\}) = 33$ exceeds the limit of 30).

3. THE mCRL2 LANGUAGE AND TOOLSET

mCRL2 is a formal specification language together with a toolset for the specification and analysis of the behavior of distributed systems and protocols [13]. Starting from its development almost a decade ago, mCRL2 is actively maintained and targets industrial-size applications. The specification language originates from the process algebra ACP. Abstract datatypes can be used to parametrize actions. mCRL2 aims to provide the user maximal access to artifacts constructed during analysis for tailored manipulation. As a consequence, the toolset consists of a wide range of tools and supports simulation, visualization, behavioral reduction and model checking, as well as dedicated optimization techniques and back-ends to other software.

The mCRL2 toolset was successfully applied in various settings. One of them concerns the massive data collection system used for the high-energy experiments conducted at the large hadron collider of CERN [23]. Parts of the system occasionally entered inconsistent states, leading to a loss of efficiency and a potential loss of data. Critical subsystems were modeled in mCRL2 and safety and liveness requirements were verified using model checking. These requirements stated, e.g., that jobs are always processed once submitted, and that jobs never enter an inconsistent state. Violations of these requirements revealed livelocks and race conditions, explaining phenomena observed in the actual system.

FlexRay is a widely adopted communication protocol in the automotive industry. It aims to provide a reliable, high-bandwidth communication channel between car components. The protocol is *time-triggered*, i.e., it relies on components to have synchronized clocks, and operates by allocating bandwidth based on a global, cyclic schedule. The FlexRay start-up procedure, which ensures that activated components will find each other and will correctly initialize their local view on the global schedule, was modeled and checked for correctness using mCRL2 [9]. The expressivity of the mCRL2 language allowed the author of [9] to specify the protocol faithfully. The robustness of FlexRay was analyzed by injecting faults that may occur in the system. This was implemented by making small, local changes to the fault-free model.

Documentation and binaries of the mCRL2 toolset can be downloaded at www.mcrl2.org. The toolset is open source; the associated boost license allows anyone free use for any purpose. For our approach to variability analysis, mCRL2's full expressivity is not needed. We can resort to relatively simple structured models, extending the range of the tools. A trivial example is the labeled transition system (LTS) in Figure 2, which can be modeled by an mCRL2 process Foo with integer st as a parameter holding the state and with actions a - e. Part of the code for the Foo process looks like:

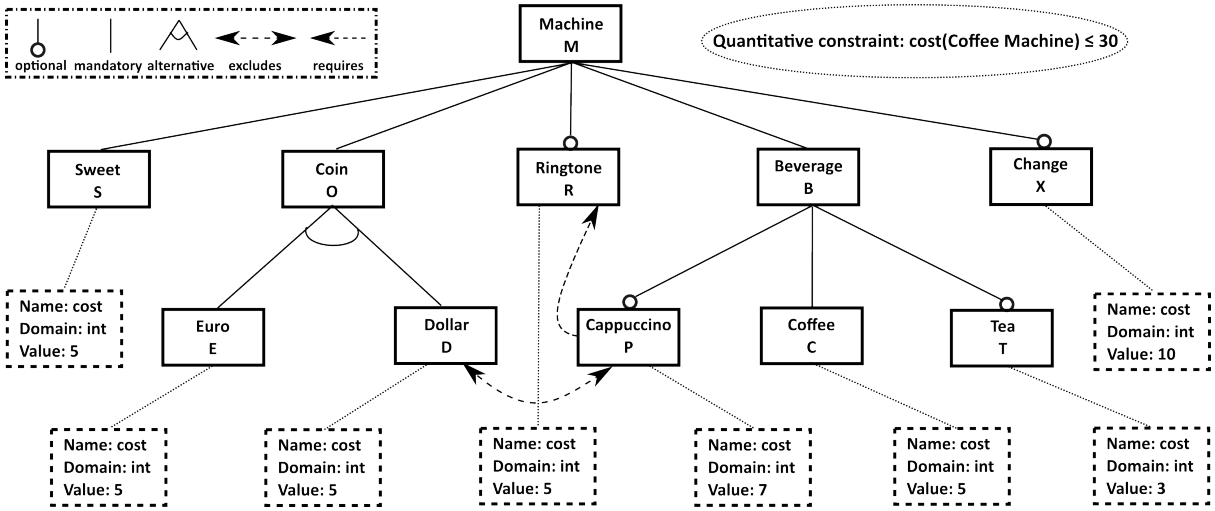

Figure 1: Attributed feature model of the family of coffee machines (with shorthand names)

```
proc Foo(st:Int) =
  ( st==0 ) -> ( b.Foo(1) + a.Foo(2) ) +
  ( st==1 ) -> ( c.Foo(3) ) +
  ( st==2 ) -> ( b.Foo(1) + b.Foo(3) + a.Foo(4) ) +
    ...
```

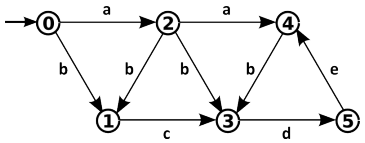

Figure 2: Example LTS

With `mCRL2`, a system property can be expressed as a formula in a variant of the modal μ-calculus [14]. Subsequently, the property can be verified against the specification using the model checking facilities of the `mCRL2` toolset. Among the properties that hold for the specification above we mention:

- `[true*]<true>true`: absence of deadlock.
- `[true*.b.true*.a]false`: after any sequence where the action `b` precedes the action `a`, *false* will hold. As the latter never holds the formula can be reformulated: no `a`-action is possible after a `b`-action has happened.
- `mu Y.(<c>||[true]Y)`: a least-fixed-point construction. Always, after a finite amount of steps a `c`-action can be done (or deadlock occurs earlier). The smallest set of states `Y` that can do a `c`-action or cannot step outside of `Y`, can be computed by iteration.
- `mu Y.((nu Z.(<b.d.e>Z))||[true]Y)`: a nesting of a least-fixed-point and a greatest-fixed-point construction. Always after a finite amount of steps an infinite repetition of `b`, `d`, and `e` is possible.

The modal μ-calculus, occasionally also dubbed the 'Logic of Everything', is known to be very expressive and to subsume other well-known temporal logics like LTL and (A)CTL. The process-algebraic approaches of [16, 3] propose multi-valued modal μ-calculus and LTL model checking, respectively, while that of [19] proposes CTL model checking. Only the approach of [3] is implemented, viz. in the Maude toolset (`maude.cs.uiuc.edu`). The appeal of the modal μ-calculus variant used in `mCRL2` [14] is the possibility to quantify over data, as we shall see in the example product line below.

The general workflow for model checking with `mCRL2` consists of (i) translating a specification `foo.mcrl2` into a standard format called a linear process specification `foo.lps`, (ii) transforming the linear process together with the modal formula `bar.mf` to a so-called parametrized Boolean equation system `foo.pbes` and then solving it, yielding *true* or *false* for the formula `bar.mf` with respect to the specification `foo.mcrl2`. Alternatively, one can generate the underlying state space `foo.lts` of the specification to visually inspect it. The hiding of well-chosen actions and minimization with respect to one of the process equivalences offered by the toolset (like trace equivalence, weak and branching bisimulation) allows one to transform `foo.lts` and to focus on specific behavioral aspects of the specification. Using the latter technique, as will be illustrated in the next section, a state space with millions of states can be reduced substantially, bringing it in scope for human examination.

4. VERIFYING THE RUNNING EXAMPLE

In our approach to variability analysis we model a product line as an `mCRL2` process. A product line can be represented as a combination of two finite state machines (FSMs). An initial FSM, whose behavior is transient, deals with feature selection. Its successful end states are in one-to-one correspondence with the consistent and complete configurations of the product line, and are the starting point for the FSM describing the actual behavior of a product. Therefore, the FSM takes an eligible set of features as an argument to model the particular instance of the product line with respect to the selected set of features. Only actions that are in agreement with these selected features will be executed.

We model the configuring of a product separate from its actual behavior (following [3]). The breadth-first node traversal of a feature diagram is directly translated into an FSM leading from the initial state to an end state. At this stage an end state is successful (but temporary) if the selected features meet the constraints of the feature model; other-

wise the end state is a deadlock state (or, alternatively, the starting point of a process that catches the error). To start with, the root feature is selected in the initial state. For our example the nodes of the feature diagram of Figure 1 are visited in the order S, O up to T to include all mandatory features and to select a number of optional features.

The full mCRL2 code can be found in the Appendix. We define a process `Sel` with parameters `st` and `fs` that hold, respectively, a state represented by an integer and a set of features included so far. The initial call to the process is `Sel(0,[M])`, i.e. the selection process starts in state 0 with only the root feature M selected. The mCRL2 code snippet below shows the inclusion of the mandatory feature O in state 1. The parameter `st` is incremented and the parameter `fs` is updated via the construct `insert(O,fs)`, adding the feature O to the feature set under construction (with the mandatory feature S added in state 0, yielding `[M,S,O]`). In state 2 there is a choice since the R-feature is optional. If the feature R is not selected the silent action `tau` is taken, the state parameter `st` is incremented but the parameter `fs` is left unchanged. If, on the contrary, the feature R is selected, then the action `setR` is taken, and the parameters are updated accordingly.

```
proc Sel(st:Int,fs:FSet) =
  ...
( st == 1 ) -> (
  ( M in fs ) -> (
    setO . Sel(2,insert(O,fs) )
) ) +
( st == 2 ) -> (
  ( M in fs ) -> (
    tau . Sel(3,fs) +
    setR . Sel(3,insert(R,fs) )
) ) +
  ...
```

This leads to a number of deterministic actions for mandatory features, like `setO`, and non-deterministic actions for optional features, like `tau` and `setR`. After all nodes (leaves included) in the feature diagram have been visited, a resulting feature set `fs` still may or may not satisfy the cross-tree and attribute constraints of the (attributed) feature model.

The cross-tree constraints are considered next. In the code snippet below, the action `wrong_set` is taken to reject the selected feature set. If this action is executed, the process enters a deadlock state; there is no transition beyond `delta`. The first condition `(D in fs) && (P in fs)` captures that the D-feature and P-feature exclude each other, i.e. they cannot both be in an admissible feature set. The second condition `!(R in fs) && (P in fs)` captures that the P-feature requires the R-feature, i.e. `P` cannot be in an admissible feature set if `R` is not. If the two tests fail, i.e. both additional constraints are met, then the action `ctc_ok` with the selected feature set `fs` as an argument is taken and the quantitative constraints on attributes are handled similarly.

```
  ( st == 8 ) -> (
  ( ( D in fs ) && ( P in fs ) ) ->
    wrong_set . delta <>
  ( !( R in fs ) && ( P in fs ) ) ->
    wrong_set . delta <>
  ctc_ok . Sel(9,fs)
) +
( st == 9 ) -> (
  ( tcost(fs) <= 30 ) ->
    set_ok(fs) . cost( tcost(fs) ) . Prod(0,fs) <>
  wrong_set . delta );
```

In our example, a function `tcosts` calculates the total costs over the set of features `fs`. If this exceeds the threshold of 30, as expressed in the attributed feature model of Figure 1, the `wrong_set` action is taken to the deadlock state. If the total costs remain within bounds, the `set_ok` is taken and control is transferred from the feature selecting process `Sel` to the process `Prod` modeling the product behavior. Note that the feature set `fs` is passed as an argument of the action `set_ok`. This will prove useful for model checking purposes later.

Thus, the modeling of feature selection is guided by the feature diagram, while afterwards fulfillment of the cross-tree and of the attribute constraints is checked, in that order.

The actual behavior of the product is launched by the call `Prod(0,fs)` for a specific consistent and complete feature set `fs`, after execution of actions `set_ok` and `cost` by the selection process `Sel`. To model the behavior of our example we follow the LTS in Figure 3. This is the LTS from [1] extended with detailed money insertion handling. For conciseness, the box labelled `Insert/Return` abbreviates the sub-LTS concerning the latter. The process `Prod` thus starts with a call to the process `Insert` to enable money insertion.

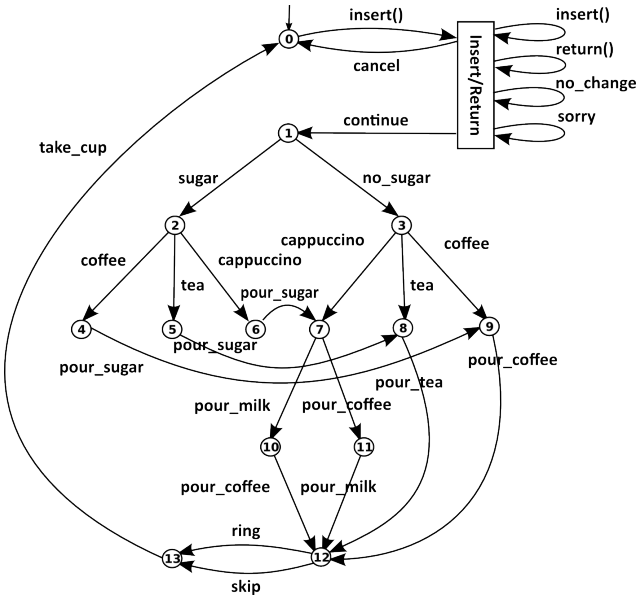

Figure 3: LTS modeling family behavior

To reflect product configuration, we only allow an action if the feature it belongs to is part of `fs` (i.e. reminiscent of the way this is done in [7, 21, 3], we implicitly assume actions to be tagged with a feature). In the mCRL2 code we have, e.g.,

```
proc Prod(st:Int,fs:FSet) =
( st == 0 ) -> ( Insert(0,fs) ) +
  ...
( st == 2 ) -> (
  ( C in fs ) -> coffee . Prod(4,fs) +
  ( T in fs ) -> tea . Prod(5,fs) +
  ( P in fs ) -> cappuccino . Prod(6,fs)
) +
  ...
```

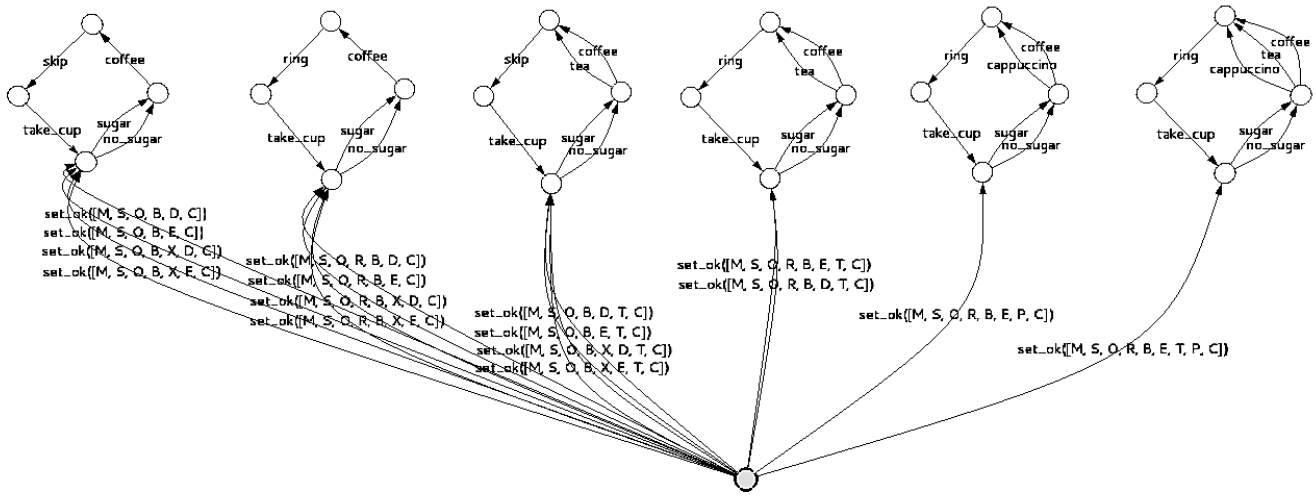

Figure 4: Product behavior with configuration and payment abstracted away

expressing that a `coffee` action is possible provided the (mandatory) C-feature is selected. Likewise for the `tea` and `cappuccino` actions (which do stem from optional features).

Handling coins is more involved and illustrates the use of data in `mCRL2`. In our example, money is to be inserted until the balance is 100 cents or more, unless the cancel button is pressed earlier. In the latter case, the balance is returned. In the former case, control continues to the handling of sugar and beverage. However, with the change feature X selected, change may be returned. Otherwise no change will be given, as notified by the `no_change` action. Below, the code for the process `Insert` is given. The process has two parameters: `bal` holding the balance and `fs` holding the selected features.

```
proc Insert(bal:Nat,fs:FSet) =
( bal < 100 ) -> (
  ( D in fs ) -> (
    insert(dime) . Insert(bal+10,fs) +
    insert(quarter) . Insert(bal+25,fs) +
    insert(half) . Insert(bal+50,fs) +
    insert(dollar) . Insert(bal+100,fs) ) +
  ( E in fs ) -> (
    insert(ct10) . Insert(bal+10,fs) +
    insert(ct20) . Insert(bal+20,fs) +
    insert(ct50) . Insert(bal+50,fs) +
    insert(euro) . Insert(bal+100,fs) ) ) +
( ( bal > 0) && ( bal < 100 ) ) ->
  Return(bal,fs) . cancel . Prod(0,fs) +
( bal >= 100 ) -> (
  ( ( !(X in fs) ) ->
    no_change . continue . Prod(1,fs) <>
  Return(Int2Nat(bal-100),fs) .
    continue . Prod(1,fs) ) );
```

For a balance less than 100 cents, insertion of a specific coin is coupled to an update of the balance. Only coins of the right currency are accepted. Note that this assumes that an eligible feature set `fs`, which was passed by the `Sel` process, contains either `D` or `E`. For the action `cancel`, the balance is returned via a call to `Return` and control returns to the initial state `Prod(0,fs)`. If, on the other hand, the balance has grown sufficiently, control proceeds to the state `Prod(1,fs)` via the `continue` action. Furthermore, in case change is returned, 100 cents are subtracted from the balance first.

Characteristic for process-algebraic approaches, including `mCRL2`, is the availability of hiding. One can abstract away a subset of actions to focus on the behavior built from the remaining ones. Moreover, as with `mCRL2` all intermediate artifacts are available to the user, these can be manipulated to the user's liking, possibly with the help of the many tools available in the toolset. For instance, when the actions of configuring the product and those related to payment are abstracted away, and the result is minimized with respect to a process equivalence like weak trace equivalence, then the state space depicted in Figure 4 is obtained. The actual behavior of such an abstracted product consists of a sequence of actions starting with a `sugar` or `no_sugar` action and ending with a `take_cup` action.

Starting from the initial state, one out of six cyclic behaviors is reached: four products with coffee only and no ringtone (varying in dollar or euro currency and the availability of change) and four products with coffee only but with a ringtone (and similar variations for the other features); four products with coffee and tea and no ringtone and, because of the cost constraint, two products with coffee and tea but with a ringtone; one product with coffee, cappuccino and a ringtone, and one product with coffee, tea, cappuccino and a ringtone. This way, by visual inspection of part of the state space, viz. the part remaining after hiding and minimization, one may verify that a product with cappuccino indeed provides a ringtone as well. In general, by these reduction techniques large state spaces can be tuned to inspect specific, not necessarily local, behavior.

The property language of `mCRL2` is first-order and thus allows to incorporate data. To show what can be considered for variability analysis using the modal μ-calculus, we discuss the following properties of our example product line:

- `[(!continue)*.take_cup] false`: if payment is not settled by action `continue`, no beverage is delivered.
- `[true*.setX.true*.no_change] false`: once the X-feature is selected, action `no_change` will not occur.
- `forall fs:FSet.val(isSet(fs)) && [true*.set_ok(fs)]true => val((D in fs) => !(P in fs))`: if a product is configured successfully as indicated by the `set_ok` action, then it cannot be a product that accepts dollars and also provides cappuccino.

- `mu Y.(<exists fs:FSet.set_ok(fs)>true || <wrong_set>true || [true]Y)`: from the initial state, after a finite number of steps, either action `set_ok` (with some parameter `fs`) or action `wrong_set` occurs.

- `forall c:Coin.[true*.insert(c)] mu Y.(<cancel || take_cup> true || [true]Y )`: after money has been inserted, in a finite number of steps, a beverage can be taken unless the transaction was canceled.

The first property only concerns the actual behavior of any eligible product. The second relates an action in the configuration phase, `setX`, and an action in the behavior phase, `no_change`. The third property involves a quantification over all possible configurable feature sets, which shows the aforementioned usefulness of passing the feature set `fs` as an argument of `set_ok`. The casts `val` are needed to yield a Boolean value. The last two properties involve a minimal fixed point over the formal variable `Y`. In the fourth property the existential quantification is framed within the modality as no further reference to the actual feature set is needed. The last property involves a universal quantification, viz. over the set of coins. In such a situation, as occurs also in property 3, it is essential that the range of the quantifier is bounded. For the relative small example coffee machine product line that we consider here, properties are model checked on a standard PC within seconds (e.g. property 3), and often much quicker (like the other properties above).

The above describes the basic setup for our approach of using `mCRL2` for variability analysis; several extensions of the basic scheme are possible. (i) For instance, similar to the way this is done in [3], dynamic feature management can be arranged for by adding transitions from the `Prod` process to the `Sel` process and back again to `Prod` to deal with feature selection on-the-fly. (ii) Here, we followed a strict order for feature selection: first the feature diagram was taken into account, then cross-tree constraints were considered and finally quantitative (attribute) constraints were checked. When dealing with many more features, it is likely advantageous to do this in a more flexible way, so as to rule out inconsistent feature sets at an early stage during selection. Clearly, any of the additional constraints can be checked as soon as all features involved have been selected or not. Even more refined selection schemes traversing the tree associated with the feature diagram can be modeled as well. (iii) As a final variation we mention the possibility to tweak the feature selection process and have it enumerate subfamilies, e.g. containing the products satisfying all requirements or those violating a specific family requirement, to gain insight in the relation between specific constraints and subsets of features. This can be achieved by stealing control at the corresponding point of the selection process and to transfer to a proper signaling process.

On a final note, because of the open workflow with `mCRL2`, any output of the toolset can be exported to other tools, e.g. SAT/SMT-solvers. Reversely, specific feature settings, e.g. resulting from other means of analysis, can be used as a starting point by jumping to the right state of the `Sel` or `Prod` process. The crucial point is that we perform product line analysis within the framework of a full-fledged verification toolset while maintaining control of the design choices to be made during modeling and of the properties to verify. Still, larger case studies need to be done to assess the scalability of our setup, in particular regarding model checking.

Several of the behavioral variability models mentioned in Sect. 1 have an associated tool for behavioral variability analysis with verification techniques like model checking. `SNIP` [6] is a model checker for product lines modeled as featured transition systems (FTSs) specified in a language based on that of the `SPIN` model checker (`spinroot.com`). The feature diagram is coded in the textual variability language TVL to be consulted by the explicit-state on-the-fly model-checking algorithm of `SNIP` to verify properties expressed in so-called feature LTL interpreted over FTSs (e.g. to verify a property over only a subset of the valid products).

Symbolic FTS model checking was implemented as an extension of the `NuSMV` model checker (`nusmv.fbk.eu`) with a fully symbolic algorithm for feature CTL, as it is called. In `SNIP`, special-purpose exhaustive model checking algorithms (continuing a search also after a violation was found) allow the user to verify all products of a product line at once and to output counterexamples for all products that violate a property (in contrast with the `NuSMV` extension that only produces a counterexample for the first violating product found). `SNIP` has recently been re-engineered and the resulting tool suite ProVeLines can handle feature attributes [8].

`VMC` [4] (`fmt.isti.cnr.it/vmc`) is a model checker for product lines modeled as modal transition systems (MTSs) with added variability constraints, but with no specific reference to feature diagrams. `VMC` offers automatic generation of one/some/all valid products (modeled as LTSs) of a product line. The user can simulate, visualize or model check either the full product line or a set of its valid products. `VMC`'s explicit-state on-the-fly model-checking algorithm allows the verification of properties expressed in so-called variability-ACTL interpreted over MTSs; it moreover offers the possibility to inspect the (interactive) explanations of a verification result. An extension handling data is forthcoming.

In [20], FSMs are extended with variability by means of guards over variables on transitions and a global predicate defining the valid configurations by value assignments. For each product line feature, two FSMs are built, one for the requirements and one for the design level, and it is specified how to check their conformance. The prototype tool `SPLEnD` transforms pairs of XML files for the FSMs into a file that can be fed to `SPIN`, which either returns the conformance mappings or declares nonconformance, after which the behavior of the product line can be checked by SAT solving. As far as we know, `SPLEnD` does not cater for feature attributes.

5. CONCLUDING REMARKS

We have presented a proof-of-concept for formal variability analysis with `mCRL2`, highlighting its main features concerning both modeling (e.g. parametrized processes, data handling) and analysis (e.g. minimization techniques, model checking properties in the modal μ-calculus with data).

Most model-checking analyses described in this paper fall in the category of *product-based* analyses, i.e. operating on individually generated products (or at most a subset) [26]. This contrasts with *family-based* analyses, operating on an entire product line at once using variability knowledge about valid feature configurations to deduce results for products, of which `SNIP` is a well-known and successful representative. `VMC` offers the full spectrum of analyses, but—contrary to the special-purpose FTS model-checking algorithms of `SNIP`—when a formula is verified over an entire product line, then a negative result does not actually list the specific products in

which the property fails to hold. However, both in VMC and in mCRL2, the full list of violating products can be obtained by model checking the formula against each individual product of the product line (inspection of a counterexample reveals one violating product only).

There might be a trade-off between brute-force product-based analysis with model checkers that have been highly optimized for single system engineering, like SPIN and—to a lesser degree—mCRL2, and highly innovative family-based analysis with model checkers that have been developed specifically for product lines, like SNIP. In fact, SPIN generally outperforms SNIP [6] (according to the authors of [6] this is due to SPIN's many optimizations, among which partial order reduction). In this respect, an evaluation of the state-of-the-art mCRL2 toolset, which may lead to the desire to implement some product line-specific features into its model-checking algorithms, is left for future work.

6. ACKNOWLEDGMENTS

Maurice ter Beek conducted part of this work while 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. Maurice ter Beek also acknowledges support of the EU FP7-ICT FET-Proactive project QUANTICOL (600708) and of the Italian MIUR project CINA (PRIN 2010LHT4KM).

7. REFERENCES

[1] P. Asirelli, M.H. ter Beek, A. Fantechi, and S. Gnesi. Formal description of variability in product families. In *SPLC*. IEEE, 2011, 130–139.

[2] J.M. Atlee, S. Beidu, N.A. Day, F. Faghih, and P. Shaker. Recommendations for improving the usability of formal methods for product lines. In *FormaliSE*. IEEE, 2013, 43–49.

[3] M.H. ter Beek, A. Lluch Lafuente, and M. Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. In *FMSPLE*. ACM, 2013, 10–17.

[4] M.H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A tool for product variability analysis. In *FM. LNCS* 7436, Springer, 2012, 450–454.

[5] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Inf. Syst.* 35, 6 (2010), 615–636.

[6] A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *STTT* 14, 5 (2012), 589–612.

[7] 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 TOSEM* 39, 8 (2013), 1069–1089.

[8] M. Cordy, P.-Y. Schobbens, P. Heymans, and A. Legay. Beyond Boolean product-line model checking: Dealing with feature attributes and multi-features. In *ICSE*. ACM, 2013, 472–481.

[9] S. Cranen. Model checking the FlexRay startup phase. In *FMICS. LNCS* 7437, Springer, 2012, 131–145.

[10] S. Cranen, J.F. Groote, J.J.A. Keiren, F.P.M. Stappers, E.P. de Vink, W. Wesselink, and T.A.C. Willemse. An overview of the mCRL2 toolset and its recent advances. In *TACAS. LNCS* 7795, Springer, 2013, 199–213.

[11] D. Fischbein, S. Uchitel, and V. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA*. ACM, 2006, 39–48.

[12] S. Gnesi and M. Petrocchi. Towards an executable algebra for product lines. In *SPLC*. ACM, 2012, 66–73.

[13] J.F. Groote, A. Mathijssen, M.A. Reniers, Y.S. Usenko, and M.J. van Weerdenburg. Analysis of distributed systems with mCRL2. In *Process Algebra for Parallel and Distributed Processing*. Chapman & Hall, 2009, 99–128.

[14] J.F. Groote and R. Mateescu. Verification of temporal properties of processes in a setting with data. In *AMAST. LNCS* 1548, Springer, 1998, 74–90.

[15] A. Gondal, M. Poppleton, and M. Butler. Composing Event-B Specifications - Case-Study Experience. In *SC. LNCS* 6708, Springer, 2011, 100–115.

[16] A. Gruler, M. Leucker, and K.D. Scheidemann. Modeling and model checking software product lines. In *FMOODS. LNCS* 5051, 2008, 113–131.

[17] Ø. Haugen and K. Stølen. STAIRS: Steps to analyze interactions with refinement semantics. In *UML. LNCS* 2863, Springer, 2003, 388–402.

[18] K.G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O automata for interface and product line theories. In *ESOP. LNCS* 4421, Springer, 2007, 64–79.

[19] K. Lauenroth, K. Pohl, and S. Töhning. Model checking of domain artifacts in product line engineering. In *ASE*. IEEE, 2009, 269–280.

[20] J.-V. Millo, S. Ramesh, S.N. Krishna, and G.K. Narwane. Compositional verification of software product lines. In *IFM. LNCS* 7940, Springer, 2013, 109–123.

[21] R. Muschevici, J. Proença, and D. Clarke. Modular modelling of software product lines with feature nets. In *SEFM. LNCS* 7041, Springer, 2011, 318–333.

[22] K. Pohl, G. Böckle, and F.J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.

[23] D. Remenska, T.A.C. Willemse, K. Verstoep, W. Fokkink, J. Templon, and H.E. Bal. Using model checking to analyze the system behavior of the LHC production grid. In *CCGrid*. IEEE, 2012, 335–343.

[24] P.-Y. Schobbens, P. Heymans, J.-C. Trigaux, and Y. Bontemps. Feature diagrams: A survey and a formal semantics. In *RE*. IEEE, 2006, 136–145.

[25] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL conqueror: Toward optimization of non-functional properties in software product lines. *Softw. Qual.* 20 (2012), 487-517.

[26] T. Thüm, S. Apel, C. Kästner, M. Kuhlemann, I. Schaefer, and G. Saake. Analysis strategies for software product lines. Technical Report FIN-004, Universität Magdeburg, 2012.

[27] 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.

APPENDIX

A. mCRL2 SPECIFICATION OF EXAMPLE

```
%% mCRL2 specification of the running example
%% variation of coffee machine with change feature,
%% cost attribute and handling of coins

sort
  Feature = struct
    M | S | O | R | B | X | D | E | P | T | C ;
  FSet = List( Feature );
  Coin = struct
    dime | quarter | half | dollar |
    ct10 | ct20 | ct50 | euro ;
  Currency = struct Dollar | Euro ;

act
  insert, return : Coin ;
  continue, cancel, sorry, no_change,
  sugar, no_sugar, coffee, tea, cappuccino,
  pour_sugar, pour_milk, pour_coffee, pour_tea,
  ring, skip, take_cup ;
  setS, setO, setR, setB, setX,
  setD, setE, setP, setT, setTP, setC ;
  wrong_set, ctc_ok ;
  set_ok : FSet ;
  cost : Int ;


map
  isSorted: FSet -> Bool;
  noDuplicates: FSet -> Bool;
  isSet: FSet -> Bool;
var
  ft,ft': Feature; fset: FSet;
eqn
  isSorted([]) = true;
  isSorted([ft]) = true;
  isSorted(ft |> (ft' |> fset)) =
    ft <= ft' && isSorted(ft' |> fset);
  noDuplicates([]) = true;
  noDuplicates(ft |> fset) =
    !(ft in fset) && noDuplicates(fset);
  isSet(fset) =
    isSorted(fset) && noDuplicates(fset);

map
  insert: Feature # FSet -> FSet;
var
  ft, ft': Feature; fset: FSet;
eqn
  insert(ft, []) = [ft];
  (ft < ft')  -> insert(ft, ft' |> fset) =
    ft |> ft' |> fset;
  (ft == ft') -> insert(ft, ft' |> fset) =
    ft' |> fset;
  (ft > ft')  -> insert(ft, ft' |> fset) =
    ft' |> insert(ft, fset);

map
  union: FSet # FSet -> FSet;
var
  ft, ft': Feature; fset, fset': FSet;
eqn
  union([], fset) = fset;
  union(fset, []) = fset;
  (ft < ft')  -> union(ft |> fset, ft' |> fset') =
    ft |> union(fset, ft' |> fset');
  (ft == ft') -> union(ft |> fset, ft' |> fset') =
    ft' |> union(fset, fset');
  (ft > ft')  -> union(ft |> fset, ft' |> fset') =
    ft' |> union(ft |> fset, fset');

map
  fcost : Feature -> Int ;
eqn
  fcost(M) =  0 ;
  fcost(S) =  5 ;
  fcost(O) =  0 ;
  fcost(B) =  0 ;
  fcost(R) =  5 ;
  fcost(D) =  5 ;
  fcost(E) =  5 ;
  fcost(X) = 10 ;
  fcost(C) =  5 ;
  fcost(T) =  3 ;
  fcost(P) =  7 ;

map
  tcost : FSet -> Int ;
var
  ft : Feature; fset : FSet;
eqn
  tcost([]) = 0;
  tcost(ft |> fset) = fcost(ft) + tcost(fset) ;


proc Sel(st:Int,fs:FSet) =
  %% feature states
  ( st == 0 ) -> (
    ( M in fs ) -> (
      setS . Sel(1, insert(S,fs) )
  ) ) +
  ( st == 1 ) -> (
    ( M in fs ) -> (
      setO .  Sel(2, insert(O,fs) )
  ) ) +
  ( st == 2 ) -> (
    ( M in fs ) -> (
      tau  . Sel(3, fs ) +
      setR . Sel(3, insert(R,fs) )
  ) ) +
  ( st == 3 ) -> (
    ( M in fs ) -> (
      setB .  Sel(4, insert(B,fs) )
  ) ) +
  ( st == 4 ) -> (
    ( M in fs ) ->
      tau .  Sel(5, fs ) +
      setX . Sel(5, insert(X,fs)
  ) ) +
  ( st == 5 ) -> (
    ( O in fs ) -> (
      setD . Sel(6, insert(D,fs) ) +
      setE . Sel(6, insert(E,fs) )
  ) ) +
```

```
  ( st == 6 ) -> (
    ( B in fs ) -> (
      tau   . Sel(7, fs ) +
      setT  . Sel(7, insert(T,fs) ) +
      setP  . Sel(7, insert(P,fs) ) +
      setTP . Sel(7, union([T,P],fs) )
  ) ) +
  ( st == 7 ) -> (
    ( B in fs ) -> (
      setC . Sel(8, insert(C,fs) )
  ) ) +

  %% cross-tree constraints
  ( st == 8 ) -> (
    ( ( D in fs ) && ( P in fs ) ) ->
      wrong_set . delta <>
    ( !( R in fs ) && ( P in fs ) ) ->
      wrong_set . delta <>
    ctc_ok . Sel(9,fs)
  ) +

  %% attribute constraints
  ( st == 9 ) -> (
    ( tcost(fs) <= 30 ) ->
      set_ok( fs ) .
        cost( tcost(fs) ) . Prod(0,fs) <>
    wrong_set . delta );


proc Prod(st:Int,fs:FSet) =
  ( st == 0 ) -> (
    Insert(0,fs)
  ) +
  ( st == 1 ) -> (
    ( S in fs ) -> ( sugar . Prod(2,fs) ) +
    ( S in fs ) -> ( no_sugar . Prod(3,fs) )
  ) +
  ( st == 2 ) -> (
    ( C in fs ) -> coffee . Prod(4,fs) +
    ( T in fs ) -> tea . Prod(5,fs) +
    ( P in fs ) -> cappuccino . Prod(6,fs)
  ) +
  ( st == 3) -> (
    ( C in fs ) -> coffee . Prod(9,fs) +
    ( T in fs ) -> tea . Prod(8,fs) +
    ( P in fs ) -> cappuccino . Prod(7,fs)
  ) +
  ( st == 4 ) -> (
    ( M in fs ) -> ( pour_sugar . Prod(9,fs) )
  ) +
  ( st == 5 ) -> (
    ( M in fs ) -> ( pour_sugar . Prod(8,fs) )
  ) +
  ( st == 6 ) -> (
    ( M in fs ) -> ( pour_sugar . Prod(7,fs) )
  ) +
  ( st == 7 ) -> (
    ( M in fs ) -> ( pour_milk . Prod(10,fs) ) +
    ( M in fs ) -> ( pour_coffee . Prod(11,fs) )
  ) +
  ( st == 8 ) -> (
    ( M in fs ) -> ( pour_tea . Prod(12,fs) )
  ) +
  ( st == 9 ) -> (
    ( M in fs ) -> ( pour_coffee . Prod(12,fs) )
  ) +
  ( st == 10 ) -> (
    ( M in fs ) -> ( pour_coffee . Prod(12,fs) )
  ) +
  ( st == 11 ) -> (
    ( M in fs ) -> ( pour_milk . Prod(12,fs) )
  ) +
  ( st == 12 ) -> (
    ( R in fs ) -> ( ring . Prod(13,fs) ) +
    ( !(R in fs) ) -> ( skip . Prod(13,fs) )
  ) +
  ( st == 13 ) -> (
    ( M in fs ) -> ( take_cup . Prod(0,fs) )
  ) ;

proc Insert(bal:Nat,fs:FSet) =
  ( bal < 100 ) -> (
    ( D in fs ) -> (
      insert(dime) . Insert(bal+10,fs) +
      insert(quarter) . Insert(bal+25,fs) +
      insert(half) . Insert(bal+50,fs) +
      insert(dollar) . Insert(bal+100,fs) ) +
    ( E in fs ) -> (
      insert(ct10) . Insert(bal+10,fs) +
      insert(ct20) . Insert(bal+20,fs) +
      insert(ct50) . Insert(bal+50,fs) +
      insert(euro) . Insert(bal+100,fs) ) ) +
  ( ( bal > 0) && ( bal < 100 ) ) ->
    Return(bal,fs) . cancel . Prod(0,fs) +
  ( bal >= 100 ) -> (
      ( ( !(X in fs) ) ->
        no_change . continue . Prod(1,fs) <>
      Return(Int2Nat(bal-100),fs) .
        continue . Prod(1,fs) )
  );

proc Return(bal:Nat,fs:FSet) =
  ( bal == 0 ) -> tau +
  ( D in fs ) -> (
    ( bal >= 50 ) ->
      return(half) . Return(Int2Nat(bal-50),fs) +
    ( ( bal < 50 ) && ( bal >= 25 ) ) ->
      return(quarter) .
        Return(Int2Nat(bal-25),fs) +
    ( ( bal < 25 ) && ( bal >= 10 ) ) ->
      return(dime) . Return(Int2Nat(bal-10),fs) +
    ( ( bal < 10 ) && ( bal > 0 ) ) ->
      sorry . Return(0,fs) ) +
  ( E in fs ) -> (
    ( bal >= 50 ) ->
      return(ct50) . Return(Int2Nat(bal-50),fs) +
    ( ( bal < 50 ) && ( bal >= 20 ) ) ->
      return(ct20) . Return(Int2Nat(bal-20),fs) +
    ( ( bal < 20 ) && ( bal > 0 ) ) ->
      return(ct10) . Return(Int2Nat(bal-10),fs) +
    ( ( bal < 10 ) && ( bal > 0 ) ) ->
      sorry . Return(0,fs)
  );

init
  Sel(0,[M]) ;
```

```
%% example system properties

mu X.(
  < exists fs:FSet . set_ok(fs) > true ||
  < wrong_set > true ||
  [ true ] X )

[ true* . setX . true* . no_change ] false

%% this is a nice one :)
[ true* . setE . true* . sorry ] false

%% next one is false, that is correct
[ true* . setD . true* . sorry ] false

[ true* . insert(ct20) ]
  mu X.( < cancel || take_cup > true || [ true ] X )

forall c:Coin .
 [ true* . insert(c) ]
 mu X.(
   < cancel || take_cup > true || [ true ] X )

[ true* ] forall c:Coin . [ insert(c) ]
  mu X.(
    < cancel || take_cup > true || [ true ] X )

%% this one is OK
forall fs:FSet . val(isSet(fs)) =>
  [ true* . set_ok(fs) . true* ] < true > true

%% but this equivalent one is verified much quicker
[ true* . (exists fs: FSet . set_ok(fs)) . true* ]
  < true > true

forall fs:FSet .
 val(isSet(fs)) && [ true* . set_ok(fs) ] true =>
   val((D in fs) => !(P in fs))

[ (!continue)* . take_cup ] false

[ true* . take_cup . (!continue)* . take_cup ] false

[ true* ] forall n:Nat . [ cost(n) ] ( val(n <= 30) )
```