Software Product Line Analysis with mCRL2

Maurice H. ter Beek
ISTI-CNR
Pisa, Italy
m.terbeek@isti.cnr.it

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

ABSTRACT

The mCRL2 language and supporting software provide a state-of-the-art tool suite for the verification of distributed systems. In this paper, we present the general principles, extrapolated from [7,8], which make us believe that mCRL2 can also be used for behavioral variability analysis of product families. The mCRL2 data language allows to smoothly deal with feature sets and attributes, its process language is sufficiently rich to model feature selection, as well as product behavior based on an FTS-like semantics. Because of the feature-orientation, our modeling strategy allows a natural refactoring of the semantic model of a product family into a parallel composition of components that reflects coherent sets of features. This opens the way for dedicated abstraction and reduction techniques that strengthen the prospect of a scalable verification approach to software product lines. In this paper, we sketch how to model product families in mCRL2 and how to apply a modular verification method, preparing the ground to further assess the scalability of our approach, in particular regarding model checking.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal methods, Model checking, Validation*; D.2.13 [**Software Engineering**]: Reusable Software—*Domain engineering*

General Terms

Design, Experimentation, Verification

Keywords

Product families, Variability, Behavioral analysis, Modular verification, Model checking

SPLC '14, September 15 - 19 2014, Florence, Italy
Copyright 2014 ACM 978-1-4503-2739-8/14/09...$15.00.
http://dx.doi.org/10.1145/2647908.2655970

1. INTRODUCTION

Feature-oriented software development (FOSD) is a popular paradigm for software product line engineering (SPLE), in which the distinguishing concept of a *feature* is pervasive throughout all phases of the software life cycle [2]. The last decades have seen formal methods and accompanying analysis tools gaining momentum in FOSD and SPLE alike. While initially focusing on the analysis of structural properties of concepts like (attributed) feature models [12], recently a lively community has broadened the scope to the analysis of the behavior of variability-intensive software systems [15].

The correctness of software (components) intended for systematic reuse as well as the correctness of the developed software families is of paramount importance. However, while formal methods and tools are successfully applied in single system engineering (of a product) for decades now, this is not the case for product line engineering (involving a product family). One reason is that formal methods traditionally do not cater for variability as a first-class concept.

In recent years, several behavioral models have been tailored to deal with the variability of software product lines. These include variants of transition systems [4,19,30,40,42], process algebras [10,32,37], Petri nets [48], Event-B [33] and state machines [46]. As a result, behavioral analysis techniques like model checking are now being applied to verify temporal properties of software product lines [11,15,18,46].

Our contribution so far has been to show the feasibility of using the formal modeling language mCRL2 and its industrial-strength toolset for the specification and (modular) verification of product families [7,8]. Since a product line's variability is exponential in the number of features, the challenge is to make modeling and analysis techniques for single products scale to product families. mCRL2 is highly optimized and comes with powerful behavioral abstraction techniques, which is why we believe that scalable verification of software product lines with mCRL2 is within reach. In this paper, we focus on the general set-up and specific features of our approach. More detailed examples can be found in [7,8]. In the future, larger case studies are needed to assess the scalability of the approach.

In Sect. 2, we briefly present the main features of the mCRL2 language and toolset, including a description of two successful applications. Consequently, we show in Sect. 3 how mCRL2 can be applied to product families by providing an overview of [7, 8], extrapolating the general principles. We compare our approach with related work and tools in Sect. 4, after which we draw some conclusions and sketch some future work in Sect. 5.

2. USING mCRL FOR SINGLE PRODUCT ANALYSIS

mCRL2 is a formal specification language for the modeling of distributed systems and their interactions, which comes with a state-of-the-art toolset for the qualitative analysis of behavior [24, 35]. The modeling language stems from μCRL [14], is based on the process algebra ACP [6] and embraces the concept of multi-actions. It supports standard data types like Int and Real and constructors like enumeration and lists, as well as user-defined higher-order abstract equational data types that can be used as action parameters. Since its release in 2003, the toolset is actively maintained and used in both academic and practical contexts.

mCRL2 has an open workflow, granting the user maximal access to all intermediate models, states spaces and verification structures constructed during analysis, thus allowing tailored manipulation. The general workflow involves three to five tools; for more specific analysis over 60 tools are available for dedicated transformations and optimization techniques, behavioral reductions, visualization, formatted export to other software tools, etc. Model checking can be done via (multi-core) explicit state space exploration or solving parametrized Boolean equation systems directly or via parity games, with speeds up to 10^5 states per second for state spaces of size 10^9, and experimental tools boosting symbolic exploration to 10^6 states per second for state spaces in the range of 10^{12} states. The supported logic is a variant of the first-order modal μ-calculus augmented with data, allowing arbitrary alternation of fixpoints. The logic subsumes other common linear and branching logics like LTL and (A)CTL [16, 25, 28]. The mCRL2 toolset strives to implement highly-optimized and up-to-date algorithms for its computations for a successful application of formal analysis techniques of industrial strength.

To illustrate its merits, we mention the application of the mCRL2 toolset in two settings outside of SPLE. The first concerns the massive data collection system used for the high-energy experiments conducted at the large hadron collider of CERN [50]. System parts occasionally entered inconsistent states, leading to a loss of efficiency and potentially data. Critical subsystems were modeled in mCRL2 and safety and liveness requirements, stating e.g. that jobs are always processed once submitted and jobs never enter an inconsistent state, were verified via model checking. Violations of these requirements revealed livelocks and race conditions, disclosing and explaining phenomena from the actual system.

The second application of mCRL2 we mention here concerns the FlexRay communication protocol and is taken from the automotive field. The protocol aims to provide a reliable, high-bandwidth communication channel between car components. Moreover, 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 find each other and correctly initialize their local view on the global schedule, was modeled and checked for correctness using mCRL2 [23]. The expressivity of the mCRL2 language allowed 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.

We argue that for product families the use of mCRL2 casts the 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.

On the website www.mcrl2.org documentation and binaries of the mCRL2 toolset can be found. The toolset is open source; the associated boost license allows free use for any academic and industrial purposes.

An mCRL2 model is expressed in an elementary process language, where actions and datatypes are introduced, processes are defined, and typically the initial process is given in the following standard concurrent form [47]

```
hide( { hided actions },
allow( { synchronized and autonomous actions },
comm( { combinations a_1|..|a_n -> a },
  process_1 || .. || process_n )))
```

i.e. a parallel composition of sequential processes with multiparty synchronization specified (comm), encapsulated to forbid loose-ends of communications (allow) and with irrelevant actions abstracted away by a hiding operation (hide). The function comm is used for the synchronization of actions by explicitly stating which actions combine into another action. Possible data parameters must be the same for all actions involved. E.g. a_1...a_n may combine into a, similar to a synchronization of actions a and $\bar{a}$ yielding τ in CCS [47]. Moreover, to constrain the interaction of processes and to prune the state space, we forbid unmatched actions by explicitly listing (allow) which actions are allowed to happen.

A toy example of an mCRL2 process is proc Soda(st:Int), a snippet of which is depicted below, whose parameter st (an integer) holds the state and whose actions are pay...close. It models the labeled transition system (LTS) in Fig. 1 (which is a product of the FTS in Fig. 2 inspired by [20]).

```
proc Soda(st:Int) =
  ( st == 1 ) -> ( pay . Soda(2) ) +
    ...
  ( st == 7 ) -> ( close . Soda(1) ) ;
```

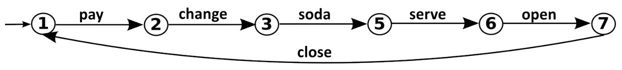
$$\begin{tikzcd} \arrow[r] & 1 \arrow[r, "pay"] & 2 \arrow[r, "change"] & 3 \arrow[r, "soda"] & 5 \arrow[r, "serve"] & 6 \arrow[r, "open"] & 7 \arrow[lllll, bend left, "close"] \end{tikzcd}$$

Figure 1: Behavior process Soda

The general workflow for checking a property expressed as modal formula for an mCRL2 model consists of (i) translating the model foo.mcrl2 into an intermediate 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 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 [47, 54]) allows one to transform the state space and to focus on specific behavioral aspects of the specification. Using the latter technique, a state space of potentially millions of states can be reduced substantially, bringing it in scope for human examination (see [8] for an SPL application).

For our approach to variability analysis, so far we have only used a smaller part of the mCRL2 toolset. In particular, because of the structure of the models, specific syntactic transformations on intermediate representations and 3D visualizations appear less relevant for application. However, it is likely that there will be occasions to add tools dedicated to software product lines in the future. The architecture of the toolset supports such extensions.

3. USING mCRL2 FOR PRODUCT FAMILY ANALYSIS

In [8] we showed how the formal specification language mCRL2 and toolset can be exploited to model and analyze software product lines (SPL). In particular, we presented a basic example to illustrate the use of mCRL2's data language to model and select valid product configurations, in the presence of feature attributes and quantitative constraints, and to model and check the behavior of valid products. This is in line with the analysis recommendations from [5] to "adopt and extend state-of-the-art analysis tools" and to "examine[s] only valid product variants". In Sect. 3.1, we describe at a global level how a product family can be encoded in mCRL2 by means of a combination of a selection process and a (parametrized) product behavior process. We stress the flexibility of the approach and underline the advantage of keeping feature selection and product execution together.

In [7], we introduced a modular verification technique to analyze SPL behavior with the mCRL2 toolset. Exploiting the same example from [8], we showed how its behavioral model can be modularized (in a feature-oriented fashion) into components, with interfaces that allow a driver process to glue them back together on the fly. This is a powerful abstraction technique that allows mCRL2 to concentrate on the relevant components (features) for a specific property under scrutiny, and in accordance with the modeling recommendation from [5] to "support (feature) modularity" in order "to visualize and (manually or automatically) analyze feature combinations corresponding to products of the product line". In Sect. 3.2, we discuss how refactoring an mCRL2 specification along the conceptual boundaries of the feature model may help in reducing the SPL model, leading to improved response times of the verification tools.

3.1 Combining Feature Models and Behavior

Our approach to SPL modeling and analysis using mCRL2 has two main processes that together represent a product family:

1. the selection process Sel reflects the product family's underlying (attributed) feature model with the constraints regarding the presence or absence of features;

2. the behavior process Prod reflects the product family's underlying featured transition system (FTS) semantics and thus concisely models individual product behavior.

Although selection and behaviour are modeled as separate processes, for model checking purposes in particular it proves useful to consider them as a whole.

The process Sel is responsible for the selection of a product, i.e. for the configuration of features. More concretely, the process performs a breadth-first traversal over the feature diagram, meanwhile resolving choices due to variability. In the simple situation, cross-tree constraints and attribute constraints can be dealt with after an initial selection has been completed. Alternatively, the checking of these additional constraints can be intertwined with the selection of features. When the selection process reaches a successful final state, a product, as identified by a set of features, is selected that is consistent with the requirements put forward by the feature model. From such a final state of Sel, control is moved to an initial state of the process Prod, which takes the set of features as collected by Sel as a parameter. Unsuccessful final states may be the starting point of a process that catches the error. Our focus, though, is on the combination of variability and behavior.

To illustrate the selection, suppose the feature model contains a node for feature f that has four children, a node for a mandatory feature m, two nodes with optional features o1 and o2, of which at most one can be chosen, and a node for an optional feature p which requires a feature r with a node elsewhere in the tree. Moreover, node f is the 123rd node to visit in the breadth-first traversal of the tree, while the node for a feature g is the next node, i.e. the 124th node to visit, and the decision to include or not include feature r is taken before. (Thus the parent node of r has position 122 or earlier in the traversal.) For the selection process declared with two parameters as Sel(st:Int,fset:FeatureSet), using st for its state and fset for the feature set under construction, we would have an mCRL2 fragment of the form

```
  .....
%% processing node 123 for feature f
( ( st == 123 ) && ( f in fset ) ) ->
  select(m) . Sel( 123', fset+{m} ) +
( st == 123' )  -> (
  skip . Sel( 123'', fset ) +
  select(o1) . Sel( 123'', fset+{o1} ) +
  select(o2) . Sel( 123'', fset+{o2} ) ) +
( ( ( st == 123'' ) && ( r in fset ) ) -> (
  skip . Sel( 124, fset ) +
  select(p) . Sel( 124, fset+{p} ) )
<>
  skip . Sel( 124, fset ) ) +
%% processing node 124 for feature g
( ( st == 124 ) && ( g in fset ) ) -> (
  .....
```

Here, 123' and 123'' are integers that do not clash with the number of the nodes. Note that Int is a built-in type, FeatureSet is a user-defined type, supposedly sets over the user-defined enumeration type Feature which includes e.g. the features r and f.

Thus, if Sel is in state 123 and the feature f has been selected previously, the action select(m) is executed and Sel continues in state 123' with updated feature set fset, thus now including the feature m. In state 123', action o1 or o2 may be taken, but the action skip may be chosen instead (at most one of o1 and o2 is allowed) and Sel continues in state 123'' with an updated feature set. Just to be explicit, we could have included a test for f in fset here too, but this is superfluous.

However, in state 123'' we need to check whether the feature r that is required for the feature p we consider now, has been selected previously and hence is in fset, the feature set under construction. If this is the case, then we can either select or not select feature p. But, if r was *not* selected previously, then we end up in the else-branch of the conditional, denoted by <>, and thus do not include p now but continue directly to state 124. Of course, it is also possible to skip the test for r in fset here and do this later, which is more modular but at the expense of computation time.

The process Prod models the behavior of products which is assumed to be given in terms of the semantics of an FTS. In essence, Prod is a finite state automaton, i.e. with transitions between states. A transition may be enabled or disabled, depending on the current set of features and possibly on other conditions that may be static or dynamic. Like the selection process Sel, the product behavior Prod has parameters st of type Int and fset of type FeatureSet. Additional parameters may be included as well, in particular to maintain the cost functions or attribute values. The parameter st again represents the current state, while fset is the set of features for the product under consideration. However, now fset is set once for Prod, initially, after which it remains fixed, i.e. we assume it will not change during execution of Prod (unless we aim at modeling dynamic features).

In a simple situation where an action a belongs to the feature f and an action b belongs to the feature g, and there are transitions from state 81 to state 82 on action a, and from state 81 to state 13 on action b, this is captured in the mCRL2 code for Prod by

```
  .....
( ( st == 81 ) && ( f in fset ) ) ->
  a . Prod( 82, fset ) +
( ( st == 81 ) && ( g in fset ) ) ->
  b . Prod( 13, fset ) +
  .....
```

If the guarding of transitions is more refined, e.g. when actions are coupled to propositional formulas and transitions may depend on run-time cost functions, the mCRL2 conditions are more elaborate as well. For example, if a transition from state 73 to state 69 on action a depends on the presence of feature f or feature g, the cost incurred so far do not exceed a threshold 10 and taking the transition increases the costs by 1, and a transition from state 73 to state 54 on action b does not depend on any feature, needs to have the cost incurred so far to be more than threshold 5 and taking the transition resets the costs, we would have

```
  .....
( ( st == 73 ) &&
  ( ( f in fset ) || ( g in fset ) ) &&
  ( costs < 10 ) ) ->
    a . Prod( 69, fset, costs+1 ) +
( ( st == 73 ) &&
  ( true ) &&
  ( costs > 5 ) ) ->
    b . Prod( 13, fset, 0 ) +
  .....
```

Here, || denotes disjunction. In fact, the mCRL2 process language supports first-order formulas over user-defined equational datatypes, allowing to express a rich set of conditions.

Validation and verification of product families can be done using the visualization and model checking facilities of the mCRL2 toolset. See [7, 8, 24, 36]. However, the fact that the selection process Sel and the product behavior process Prod have been kept together, now turns out to be an advantage. After all, the mCRL2 specification of the product family is a text file and can be adapted and manipulated at will. Hence, when focusing on a specific combination of features, conceptually the underlying feature model gets trimmed. Apart from the features involved, all (reachable) mandatory options need to be taken, all optional features can be ignored. This yields a selection process Sel' which is typically much smaller than the process Sel.

Also the product behavior process can be represented in a more abstract way. Transitions (positively) depending on non-relevant optional features can be left out. But on top of this, transitions that do not depend on any of the relevant features can be hidden, i.e. renamed into the silent action τ. This may yield a modest reduction in states only as many transitions, although labeled with τ now, still remain. However, subsequent transformation of the resulting LTS modulo branching bisimulation identifies many of the states and removes many of the τ-transitions while retaining an LTS which satisfies the same set of formulas (in the fragment of the μ-calculus respected by branching bisimulation [26]). Care has to be taken though, when dealing with dynamic aspects, such as the costs in the code snippet above. But also in such cases, an attribute-enhanced version of the branching bisimulation minimization algorithm may lead to significant reduction in the numbers of states and transitions. As one can imagine, this is not standardly provided and would indeed require another tool in the toolset, an SPL-dedicated C++-module that implements the algorithm.

We see that besides the possibility of doing behavior-oriented variability analysis working with a model checker, the use of mCRL2 for product family analysis comes with a bonus, since its process-algebraic language allows user-defined datatypes in addition to the built-in types of integers, natural numbers, Boolean, etc. and standard type constructors as enumeration, lists and sets. We expect this will support the analysis of SPL with dynamic attributes. Moreover, the modal μ-calculus variant used in mCRL2 [34] allows to quantify over data.

3.2 Modular Verification of Product Families

Existing compositional approaches to model checking exploit the native modular structure of a design to decompose system properties into an equivalent combination of properties over system modules or components. Although an appealing idea in order to cope with state space explosion, it turns out that the idea is not easily applied, mainly due to the difficulty to (de)compose properties, cf. [1, 13, 29, 38, 39]. However, Fisler and Krishnamurthi observed that in feature-oriented system designs, components typically reflect behavioral borders [31, 43, 44]. Because properties of interest of such systems are feature geared, their decomposition aligns well with the overall architecture. Therefore, variability analysis that can exploit the alignment of system properties and system components, may contribute towards the upscaling of behavior-oriented model checking to industrial-size product lines. We argue that our approach to SPL modeling fits this characteristic.

When focusing on a subsystem of components, the crux is to introduce a ‘stub’ that mimics the environment in its interaction with the subsystem. In programming, stubs are used as placeholders for unknown implementations, whose interfaces *are* known. Stubs contain just enough code to be compiled and linked with the rest of the program. In our approach, a stub provides an interface to the selected components and simulates the relevant part of the transition sequences from every possible exit from the subsystem to each reachable entry or re-entry of the subsystem.

Generally, the components involved and the stub together are substantially smaller than the complete system. We thus created an abstract system on which to do the actual analysis. If the abstract system and the original are branching bisimilar [54], i.e. the two systems respect a notion of equivalence that preserves their branching structure, then for many properties (i.e. properties expressive in CTL* without the next operator [26]) validity for the two systems is the same. (In fact, the choice for branching bisimulation is relatively arbitrary; any other behavioral equivalence that respects validity of the property under consideration will work as well.)

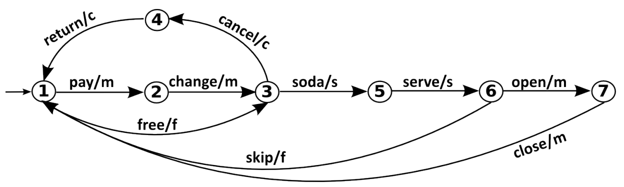

Figure 2: Behavior process `Prod`

The general workflow of the compositional `mCRL2` approach is as follows. Start with the combined attributed feature model and LTS as described above, i.e. the 'sequential' composition of the selection process `Sel` and the parametrized product behavior process `Prod`. The latter reflects the underlying FTS semantics, see e.g. Fig. 2 inspired by [20] (of which the LTS in Fig. 1 is a product). Next, the `Prod` process is refactored into a driver process `Driver` in parallel with a number of components depending on disjoint sets of features, see Fig. 3. When verifying a specific property for a single component $\mathtt{Comp}_0$ (for simplicity), an abstraction of the irrelevant components $\mathtt{Comp}_1, \ldots, \mathtt{Comp}_n$ is formulated, called `Stub`. In case the specification of the abstract system

$$\mathtt{Sel}; (\mathtt{Driver} \parallel \mathtt{Comp}_0 \parallel \mathtt{Stub})$$

is branching bisimilar to the specification

$$\mathtt{Sel}; (\mathtt{Driver} \parallel \mathtt{Comp}_0 \parallel \mathtt{Comp}_1 \parallel \ldots \parallel \mathtt{Comp}_n)$$

of the original system, then the property holds for the latter specification exactly when the property holds for the former. However, as noted, the state space is reduced in general and and so is thus the effort to model check the property.

In the refactored `mCRL2` encoding, a product as given by an eligible feature set is not represented by a monolithic LTS, but rather as the superposition of component processes, preceded by the selection process of the product line that configures the product. Typically, there is a component process for each coherent group of nontrivial features and a component process for the root feature. After being woken up by the selection process, the behavior process belonging to a product is thus captured by a system of parallel processes, each giving and gaining control when needed. In order to enforce a proper flow of control, the component processes are put in parallel with a driver process which coordinates their interaction. The driver process is relatively simple. It keeps track of the current state of the underlying FTS semantics, allowing any component with an active transition to perform an action. Next, it catches the new state number, raised by the component's transition, and the driving starts anew from that state, etc.

When we replace a number of components by a stub, we need to combine the behavior of the components comprising the stub such that the combined transitions tell where the driver ends up when leaving the perimeter of the component currently under focus. Consider, e.g., the refactoring of the behavior process in Fig. 3, where we see four components. In this simple case the components each belong to a single feature. Gluing the isolated components on equally numbered states together yields the original behavior process of Fig. 2. The driver process plays this role, allowing e.g. a transition labeled `soda` from state 3 in the s-component after a transition labeled `change` to state 3 in the m-component.

More specifically, the driver is modeled by

```
proc Driver(st:Int) =
  drv_start(st) . sum st':Int . drv_end(st') .
  Drv(st')
```

The driver thus announces the current state `st` and is willing to accept any state as the next state. To this aim, `comm` combines driver actions `drv_start(st)` and `drv_end(st')` with component actions `cmp_start(st)` and `cmp_end(st')`, respectively. The sum construction is to be interpreted as a non-deterministic choice over all states. The soda component is refactored into the process

```
proc Soda(fset:FeatureSet) =
  ( s in fset ) -> (
    cmp_start(3) . soda . cmp_end(5) +
    cmp_start(5) . serve . cmp_end(6) )
```

If the driver announces that the current state is `3`, the soda component can make it `5` by performing a `soda` action. The component can bring the system in state `6` by executing the action `serve` if the current state is `5`. The stub process should capture the behavior of the other components together. For the example the `Stub` will contain

```
  .....
( ( c in fset ) && ( m in fset ) ) -> (
  cmp_start(3) . tau . cmp_end(3) ) +
( m in fset ) -> (
  cmp_start(6) -> tau . cmp_end(3) ) +
( ( m in fset ) && ( f in fset ) ) -> (
  cmp start(6) -> tau . cmp_end(3) ) +
  .....
```

Like the soda component the stub interacts with the driver. Since the stub reflects the combined behavior of the remaining components it provides a loop from state `3` to itself, based on actions of both the `c` and `m`-component. However, this involves the condition to be met that both the (root) feature `m` and the (optional) feature `c` are present in the feature set of the product or group of products under consideration. When the system is in state `6` the stub can bring control to state `3` where the soda component can pick up again. This can be done in two ways, either via the `skip` shortcut of the `f`-component together with actions of the `m`-component or by the `m`-component alone. In this case the stub can be simplified, as the latter possibility subsumes the former. Like the refactoring, such transformation can be done by acting directly on the `mCRL2` code of the model itself.

4. RELATED WORK AND TOOLS

There is a vast amount of work on the analysis of product families. In this section we focus on model-checking tools that were specifically developed for behavioral variability analysis (in Sect. 4.1) and we discuss approaches to modularize SPL analysis (in Sect. 4.2).

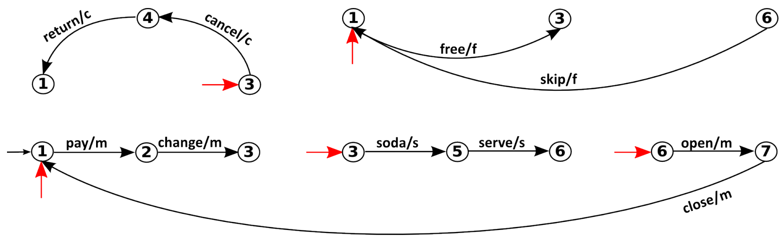

Figure 3: Refactored behavior process

4.1 Model Checkers for Variability Analysis

SNIP [18] is a model checker for product families modeled as FTSs specified in a feature-oriented variant of Promela, which is the input language of the SPIN model checker (cf. spinroot.com). A textual encoding of a feature diagram can be consulted by SNIP's explicit-state on-the-fly model-checking algorithm to verify properties expressed in a feature-oriented version of LTL interpreted over FTSs.

Symbolic FTS model checking was implemented as an extension of the NuSMV model checker (cf. nusmv.fbk.eu) by a fully symbolic algorithm for a feature-oriented version of CTL. Special-purpose exhaustive model checking algorithms enable SNIP to continue a search also after a violation is found. As such, all products of a product family can be verified at once, after which counterexamples are produced for all products that violate a property. This differs from the NuSMV extension which only produces a counterexample for the first violating product it finds.

SNIP has recently been re-engineered and the resulting tool suite ProVeLines [21] can handle feature attributes [22].

VMC [11] (cf. fmt.isti.cnr.it/vmc) is a model checker for behavioral variability analysis in product families modeled as modal transition systems (MTSs) specified in a value-passing modal process algebra [9]. Textual variability constraints on actions can be consulted by VMC's explicit-state on-the-fly model-checking algorithm to verify properties expressed in a variability-aware version of ACTL interpreted over MTSs. VMC offers automatic generation of one, some or all valid products (modeled as LTSs) of a product family (modeled as an MTS). The user can simulate, visualize or model check either the entire product family or a set of valid products. Moreover, VMC offers the possibility to inspect the (interactive) explanations of a verification result.

We close this comparison with a few words on SPL analysis approaches that, like mCRL2, are based on process algebras. Similar to mCRL2, the approach of [37] allows the verification of properties expressed in the multi-valued modal μ-calculus, whereas that of [42] is based on CTL model checking. The only approach that is actually implemented, viz. in the Maude toolset (cf. maude.cs.uiuc.edu) which comes with LTL model checking, is the one presented in [10].

None of the above tools support modular verification.

4.2 Modularization of Variability Analysis

Fisler and Krishnamurthi [31, 43, 44] were the first to note that product families naturally decompose around features so that the resulting components align well with properties of interest. Partly improving this pioneering work, [45] contains an incremental compositional model checking approach to product families using so-called variation point obligations expressed in CTL to guarantee that the (sequential) feature-based composition satisfies a property if and only if the added features satisfy the relevant obligations. Verification results are reused in an incremental fashion within the product being composed, reducing the overall verification effort. The approach however does not aim to reuse properties of behavioral feature models across different products.

In [51], an existing compositional verification technique for safety properties of flow-graph behavior of general-purpose programs is adapted to program families, organized according to a hierarchical variability model defining interfaces and variation points. This compositional approach scales well, but it is not feature-based and limited to control-flow behavior, for which it can express properties in a fragment of the modal μ-calculus.

In [48], feature Petri nets are defined as a modular (feature- and interface-based) behavioral modeling formalism. Some correctness criteria, based on bisimulation, for preserving properties in composed models are given. Model checking is not addressed and there is no reuse of verification results.

In [46], for each feature of a product family two finite state machines with variability (implemented by guarded variables on transitions) are built, one for the requirements and one for the design level, after which their conformance can be checked in a compositional, feature-based fashion. The prototype SPLEnD uses SPIN to implement conformance checking. Reuse of verification results is not considered.

5. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented the general principles for which we advocate the use of the mCRL2 toolset for the behavioral analysis of product families, as initiated in [7, 8].

The data language of mCRL2 allows to deal with feature sets and attributes smoothly, the process language is sufficiently rich to model feature selection and product behavior based on an FTS semantics, and the variant of the μ-calculus that acts as property language for the toolset supports the use of data in formulas. We sketched how product families can be modeled and presented a modular verification method that takes advantage of the factorization of a software product line based on its features. Still, larger case studies need to be done to assess the scalability of our setup, in particular regarding model checking.

Our approach differs from modular or compositional verification in the classic sense of (re)composing smaller verification results on modules or components to derive properties of the composed system. It remains to investigate whether we could apply compositional model checking under sequential composition as defined in [41] to our feature-oriented modularization of behavioral SPL models. Likewise it remains to study whether a notion like modular validity (a property holds over a module if it holds over any system that includes that module [49]) can be effectively used in our setting.

Another challenge for our feature-oriented modular verification approach stems from the fact that, ideally, we want to be able to handle dynamic feature-based composition. If a feature is added, then on the one hand we want to prove that properties of the system continue to hold, while on the other hand we want to prove new properties that the new system should now satisfy. This is complicated by the well-known fact that features may interact [3, 17].

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) [53]. 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` and `ProVeLines` are well-known and successful representatives. `VMC` offers a bit of both types of analysis, 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).

An alternative route for `mCRL2` would be to consider groups of products where it is left unspecified whether a feature is included or not. For this to work, cross-tree constraints would need to be treated cautiously. Such a solution would only require textual transformation of the `mCRL2` model and might therefore be feasible. Future work should confirm this.

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` and `ProVeLines`. In fact, in [18] it is said that `SPIN` generally outperforms `SNIP` due to `SPIN`'s many optimizations, among which partial order reduction. In this respect, an evaluation of the `mCRL2` toolset for SPL analysis may lead to the desire to implement, on top of facilities to transform `mCRL2` models as described above, some product line-specific features into its model-checking algorithms. We are currently investigating an FTS-specific notion of branching bisimulation and its associated minimization procedure.

The open `mCRL2` workflow means that any output of the toolset can be exported to other tools, like SAT/SMT-solvers. Reversely, specific feature settings, e.g. resulting from other means of analysis, can be used as a starting point by jumping directly to the right state of the `Sel` or `Prod` process.

Finally, we intend to compare our modular verification approach with [52], where behavioral models of the feature-oriented requirements modeling language FORML are decomposed into so-called feature modules, expressed as one or more parallel UML-like state machines called feature machines, each of which specifies the behavioral requirements for a single feature of the SPL.

6. ACKNOWLEDGEMENTS

Ter Beek was supported by MIUR project CINA (PRIN 2010LHT4KM) and EU project QUANTICOL (FP7 600708).

7. REFERENCES

[1] M. Abadi and L. Lamport. Conjoining Specifications. *ACM TOPLAS*, 17(3):507–534, 1995.

[2] S. Apel and C. Kästner. An Overview of Feature-Oriented Software Development. *J. Object Technology*, 8(5):49–84, 2009.

[3] S. Apel, S. S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *FOSD'13*, pages 1–8. ACM, 2013.

[4] P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal Description of Variability in Product Families. In *SPLC'11*, pages 130–139. IEEE, 2011.

[5] 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'13*, pages 43–49. IEEE, 2013.

[6] J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, 2010.

[7] M. H. ter Beek and E. P. de Vink. Towards Modular Verification of Software Product Lines with mCRL2. In *ISoLA'14*, LNCS. Springer, 2014. To appear.

[8] M. H. ter Beek and E. P. de Vink. Using mCRL2 for the analysis of software product lines. In *FormaliSE'14*, pages 31–37. IEEE, 2014.

[9] M. H. ter Beek, S. Gnesi, and F. Mazzanti. Model Checking Value-Passing Modal Specifications. In *PSI'14*, LNCS. Springer, 2014. To appear.

[10] M. H. ter Beek, A. Lluch-Lafuente, and M. Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. In *SPLC'13*, volume 2, pages 10–17. ACM, 2013.

[11] M. H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A Tool for Product Variability Analysis. In *FM'12*, volume 7436 of *LNCS*, pages 450–454. Springer, 2012.

[12] D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Inf. Syst.*, 35(6), 2010.

[13] S. Berezin, S. V. A. Campos, and E. M. Clarke. Compositional Reasoning in Model Checking. In [27], pages 81–102.

[14] S. Blom, W. Fokkink, J. F. Groote, I. van Langevelde, B. Lisser, and J. van de Pol. μCRL: A Toolset for Analysing Algebraic Specifications. In *CAV'01*, volume 2102 of *LNCS*, pages 250–254. Springer, 2001.

[15] P. Borba, M. B. Cohen, A. Legay, and A. Wąsowski. Analysis, Test and Verification in The Presence of Variability. *Dagstuhl Reports*, 3(2):144–170, 2013.

[16] J. Bradfield and C. Stirling. Modal μ-calculi. In *Handbook of Modal Logic*, pages 721–756. Elsevier, 2007.

[17] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Comput. Networks*, 41(1):115–141, 2003.

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

[19] 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 TSE*, 39(8):1069–1089, 2013.

[20] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model Checking Lots of Systems. In *ICSE'10*, pages 335–344. ACM, 2010.

[21] M. Cordy, A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay. ProVeLines: a product line of verifiers for software product lines. In *SPLC'13*, volume 2, pages 141–146. ACM, 2013.

[22] 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'13*, pages 472–481. IEEE, 2013.

[23] S. Cranen. Model Checking the FlexRay Startup Phase. In *FMICS'12*, volume 7437 of *LNCS*, pages 131–145. Springer, 2012.

[24] 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'13*, volume 7795 of *LNCS*, pages 199–213. Springer, 2013.

[25] R. De Nicola and F. W. Vaandrager. Action versus State based Logics for Transition Systems. In *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.

[26] R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *J. ACM*, 42(2):458–487, 1995.

[27] W. P. de Roever, H. Langmaack, and A. Pnueli. COMPOS'97. volume 1536 of *LNCS*. Springer, 1997.

[28] E. A. Emerson. Model Checking and the mu-calculus. In *DIMACS Workshop on Descriptive Complexity and Finite Models*, pages 185–214. AMS, 1996.

[29] B. Finkbeiner, Z. Manna, and H. Sipma. Deductive verification of modular systems. In [27], pages 239–275.

[30] D. Fischbein, S. Uchitel, and V. A. Braberman. A foundation for behavioural conformance in software product line architectures. In *ROSATEA'06*, pages 39–48. ACM, 2006.

[31] K. Fisler and S. Krishnamurthi. Modular Verification of Collaboration-Based Software Designs. In *ESEC/FSE'01*, pages 152–163. ACM, 2001.

[32] S. Gnesi and M. Petrocchi. Towards an executable algebra for product lines. In *SPLC'12*, volume 2, pages 66–73. ACM, 2012.

[33] A. Gondal, M. Poppleton, and M. Butler. Composing Event-B Specifications - Case-Study Experience. In *SC'11*, volume 6708 of *LNCS*, pages 100–115. Springer, 2011.

[34] J. F. Groote and R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In *AMAST'98*, volume 1548 of *LNCS*, pages 74–90. Springer, 1998.

[35] J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. van Weerdenburg. The Formal Specification Language mCRL2. In *Methods for Modelling Software Systems*, volume 06351 of *Dagstuhl Seminar Proceedings*, 2007.

[36] 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*, pages 99–128. Chapman & Hall, 2009.

[37] A. Gruler, M. Leucker, and K. Scheidemann. Modeling and Model Checking Software Product Lines. In *FMOODS'08*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.

[38] O. Grumberg and D. Long. Model checking and modular verification. *ACM TOPLAS*, 16(3):843–871, 1994.

[39] O. Kupferman and M. Y. Vardi. Modular Model Checking. In [27], pages 381–401.

[40] K. G. Larsen, U. Nyman, and A. Wąsowski. Modal I/O Automata for Interface and Product Line Theories. In *ESOP'07*, volume 4421 of *LNCS*, pages 64–79. Springer, 2007.

[41] K. Laster and O. Grumberg. Modular Model Checking of Software. In *TACAS'98*, volume 1384 of *LNCS*, pages 20–35. Springer, 1998.

[42] K. Lauenroth, K. Pohl, and S. Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *ASE'09*, pages 269–280, 2009.

[43] H. C. Li, K. Fisler, and S. Krishnamurthi. The Influence of Software Module Systems on Modular Verification. In *SPIN'02*, volume 2318 of *LNCS*, pages 60–78. Springer, 2002.

[44] H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for Modular Feature Verification. In *ASE'02*, pages 195–204. IEEE, 2002.

[45] J. Liu, S. Basu, and R. R. Lutz. Compositional model checking of software product lines using variation point obligations. *Aut. Softw. Eng.*, 18(1):39–76, 2011.

[46] J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane. Compositional Verification of Software Product Lines. In *IFM'13*, volume 7940 of *LNCS*, pages 109–123. Springer, 2013.

[47] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[48] R. Muschevici, J. Proença, and D. Clarke. Modular Modelling of Software Product Lines with Feature Nets. In *SEFM'11*, volume 7041 of *LNCS*, pages 318–333. Springer, 2011.

[49] A. Pnueli. In transition from global to modular temporal reasoning about programs. In *Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1985.

[50] 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'12*, pages 335–343. IEEE, 2012.

[51] I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional Algorithmic Verification of Software Product Lines. In *FMCO'10*, volume 6957 of *LNCS*, pages 184–203. Springer, 2012.

[52] P. Shaker, J. M. Atlee, and S. Wang. A feature-oriented requirements modelling language. In *RE'12*, pages 151–160. IEEE, 2012.

[53] T. Thüm, S. Apel, C. Kästner, I. Schaefer, and G. Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *Comput. Surv.*, 2014. To appear.

[54] R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996.