Towards Modular Verification of Software Product Lines with mCRL2*

Maurice H. ter Beek[1] and Erik P. de Vink[2,3]

[1] ISTI–CNR, Pisa, Italy
maurice.terbeek@isti.cnr.it
[2] Eindhoven University of Technology
[3] CWI, Amsterdam, The Netherlands
evink@win.tue.nl

Abstract. We introduce by means of an example a modular verification technique for analyzing the behavior of software product lines using the mCRL2 toolset. Based on feature-driven borders, we divide a behavioral model of a product line into a set of separate components with interfaces and a driver process to coordinate them. Abstracting from irrelevant components, we verify properties over a smaller behavioral model, which not only simplifies the model checking task but also makes the result amenable for reuse. This is a fundamental step forward for the approach to scale up to industrial-size product lines.

1 Introduction

Modular or compositional verification by means of model checking has been widely studied as a way to cope with the state space explosion phenomenon (see, e.g., [1, 23] or the survey papers in [16]). Traditionally, the idea is to exploit the native modular structure of a design to decompose system properties into properties over system modules or components. In practice, it turns out that this is far from trivial in practice, mainly due to the difficulty to (de)compose properties. A major reason for this difficulty is the misalignment between behavioral properties and the modular design structures that tend to reflect conceptual rather than behavioral borders. Hence, for modular verification to be successful, it is important that a design can be decomposed into components that align well with the properties under consideration. Fisler and Krishnamurthi were the first to notice that this characteristic is inherent to software product lines or feature-oriented system designs, since most properties of interest concern features and system modules or components, and naturally decompose around features [19, 27, 28]. In line with their findings, in this paper we present a feature-oriented modular approach to the verification of software product lines with mCRL2.

In [5] we showed how the formal specification language mCRL2 and toolset can be exploited to model and analyze software product lines. In particular, we

* Research partly supported by the EU FP7-ICT FET-Proactive project QUANTICOL (600708) and by the Italian MIUR project CINA (PRIN 2010LHT4KM).

presented a basic example to illustrate the use of mCRL2's parametrized 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 [3] to "adopt and extend state-of-the-art analysis tools" and to "examine[s] only valid product variants". We also hinted at the use of model reduction. In this paper, we concretize this. Using the example from [5], we show 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 the specific property under scrutiny, and in accordance with the modeling recommendation from [3] to "support (feature) modularity" in order "to visualize and (manually or automatically) analyze feature combinations corresponding to products of the product line".

Formal methods and analysis tools are gaining popularity in software product line engineering, as can be witnessed from the successful FMSPLE workshop series affiliated with the last four editions of SPLC. While initial approaches focused on their use in proving structural properties, recently a lively community of researchers is verifying behavioral properties in the presence of variability [9]. Given the rise of software product line engineering in embedded, distributed and safety-critical systems, it is important to provide a means of quality assurance. The work closest to ours are the process-algebraic approaches of [26, 6] and, originating from [18], the transition system-based approaches of [25, 2, 12]. However, a product line's variability is exponential in the number of features. So, a major challenge is to make the proposed techniques (more) scalable, in particular by mitigating the input problem with the help of abstraction. Since mCRL2 is highly optimized and comes with powerful behavioral abstraction techniques, it fosters the hope that scalable verification of product lines is not an utopy.

The goal of this paper is to contribute towards making the variability analysis approach introduced in [5] scale to industrial-size product lines. In that approach, a product line is modeled as an mCRL2 process consisting of two (sequential) parts. The first part concerns feature selection and its output are consistent and complete product configurations; the second part capture product behaviour. By keeping the two parts together, model checking can be treated on the system as a whole without restricting to a specific product a priori. This way, also feature interaction is reflected in product behaviour as the execution of an action depends on the presence or absence of the corresponding feature.

The focus of the present paper is on the second part of an mCRL2 specification of a product family, which models product behavior. Based on feature-driven borders, we divide the behavioral mCRL2 model into a set of separate components with interfaces in the form of exit and (re-)entry transitions, and we define an additional driver process that coordinates them into exhibiting the same behavior as before. As a result, we can concentrate property verification on part of the state space, by considering on a specific (set of) components only, abstracting from the other components, i.e. the environment.

Overall the approach with mCRL2 runs like this. An attributed feature model and FTS are represented by a 'sequential' composition of a selection process Sel and a parametrized product behaviour process Beh. Next, the Beh process is refactored in a driver process Driver in parallel with a number of components depending on disjoint sets of features. When verifying a specific property for a component $\texttt{Comp}_0$, an abstraction of the irrelevant components $\texttt{Comp}_1, \ldots, \texttt{Comp}_n$ is formulated called Stub. If it is the case the the specification Sel ; (Driver $\|$ $\texttt{Comp}_0$ $\|$ Stub) is branching bisimilar to the specification Sel ; (Driver $\|$ $\texttt{Comp}_0$ $\|$ $\texttt{Comp}_1$ $\|$ ... $\|$ $\texttt{Comp}_n$) the property holds for the latter specification exactly when the property holds for the former. However, the state space of the abstracted process is signifiantly smaller in general.

The technique is implemented in our mCRL2 model by creating a 'stub' to replace the environment, creating a smaller model that is branching bisimilar [20] with the original one, hence enjoying the same behavioral properties. In programming, stubs are used as placeholders for unknown implementations whose interfaces *are* known. Such stubs contain just enough code to allow them to be compiled and connected with the rest of the program. In our approach, a stub makes use of the interface of the selected component(s) to simulate the transition sequences from every possible output (exit transition) of the component to each reachable input ((re-)entry transition) for the component. This makes it possible to abstract from other, irrelevant components and thus verify local properties over a smaller behavioral model. This not only simplifies the model checking, in the sense that standard algorithms suffice and limited computing power is required, but it moreover allows the result to be reused for other verifications. Under conditions, as long as the interface with the chosen component remains unaltered, and the complete environment and stub are equivalent processes (i.e. branching bisimilar), the property of the component verified already remains valid. In this sense the obtained result can be reused in a subsequent but different setting. We believe this to be an important step towards scaling the approach to industrial-size product lines.

In this paper, we present our ideas on the basis of 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 feature-based modular verification of software product lines using the mCRL2 toolset. To this end, Section 2 introduces the type of feature models we use and the coffee machine product line we use as a running example. Section 3 provides the background on mCRL2 necessary to understand Sections 4 and 5, where our approach to modular verification is illustrated by applying it to our example product line. Section 6 discusses related work, while Section 7 closes the paper with concluding remarks and ideas for future work.

2 A Product Line as Running Example

Our running example is an extension of the family of coffee machines from [2] and a slight adaptation of the one in [5]. It has the following list of requirements:

– To start operation, money must be inserted: either one euro, exclusively for European products, or one dollar, exclusively for Canadian products.
– Optionally, input of money can be canceled via a cancel button, after which the machine returns the inserted coin.
– Once the machine contains money, the user may indicate 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 all products must offer coffee while only European products may offer cappuccino.
– Optionally, a ringtone may be rung after delivering a beverage. However, a ringtone must be rung by all products that offer cappuccino.
– After the beverage is taken, the machine returns idle.

In this paper, we reserve the term *feature diagram* for an and/or-hierarchy of features of a product line, regulating their presence in products, whereas we speak of a *feature model* when a feature diagram is also equipped with *cross-tree constraints*. Finally, by adding (non-functional) attributes to features and quantitative constraints we obtain an *attributed feature model* [8].

Figure 1 depicts the attributed feature model of our example product line, with 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 $cost\colon Feature \to \mathbb{N}$, associated to the attribute *cost*, extends to products straightforwardly: $cost(product) = \sum\{\, cost(feature) \mid feature \in product \,\}$.

Our particular example only involves binary cross-tree constraints, non-interacting feature-wise quantifiable attributes and a single optimization objective. However, more general and complex constraints, properties and objectives can be treated as well [8, 37]. 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 sets of non-trivial features. The feature model reduces this number to 20, while the number 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 Analyzing System Behavior with mCRL2

mCRL2 is a formal specification language with an associated toolset for the modeling and verification of distributed and/or concurrent system behavior and protocols [22]. For about a decade, mCRL2 is actively being maintained and targeting industrial-size applications. Its specification language originates from the process algebra ACP [4]. The user can use abstract datatypes to parametrize actions and has maximal access to artifacts constructed during analysis, allowing tailored manipulation. To this aim, 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 tools.

The mCRL2 toolset has successfully been applied in various settings, among which the massive data collection system used for the high-energy experiments

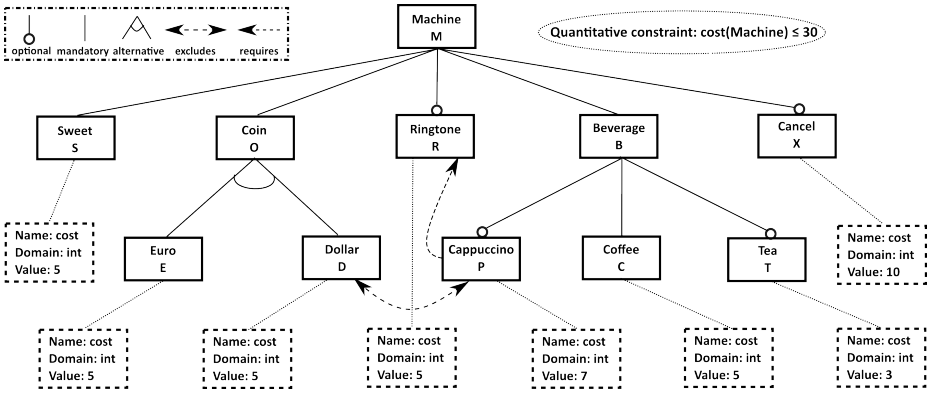


Fig. 1. Attributed feature model of family of coffee machines (with shorthand names)

conducted at the large hadron collider of CERN [35] and the `FlexRay` communication protocol used in the automotive industry to equip car components with a reliable, high-bandwidth communication channel [13]. The toolset is open source and the associated boost license allows free use for any purpose. Its binaries and lots of further documentation can be downloaded from `www.mcrl2.org`.

We will not use the full expressivity of `mCRL2` in our approach here. Relatively simple structured models suffice, which extends the range of the toolset. A simple example is the labeled transition system (LTS) below, which can be modeled by the `mCRL2` process `Foo`, with integer `st` as a state parameter and actions `a` to `e`:

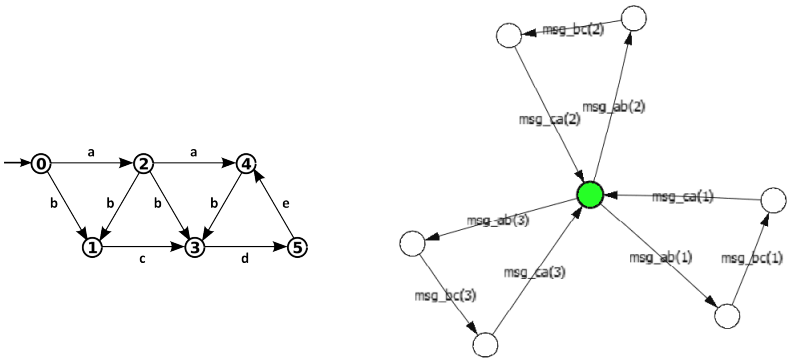
```
proc Foo(st:Int) =
  ( st==0 ) -> ( b.Foo(1) + a.Foo(2) ) +
  ( st==1 ) -> ( c.Foo(3) ) +
  ( st==2 ) -> ( b.Foo(1) + b.Foo(3) + a.Foo(4) ) +
  ...
```

Another construction that is typical for the specification of parallel processes in `mCRL2` is the combined use of the communication and encapsulation operator. Consider the following three processes:

```
proc A = ( send_a(1) + send_a(2) + send_a(3) ) .
           sum n:Int . receive_a(n) . A;
proc B = sum n:Int . receive_b(n) . send_b(n) . B;
proc C = sum n:Int . receive_c(n) . send_c(n) . C;
```

Thus, process `A` starts with sending value `1`, `2`, or `3`. Next it is willing to receive any integer value `n` and then starts all over again. Note that the summation over integers should be interpreted as an infinite non-deterministic choice. Processes `B` and `C` are similar to `A`. However, these processes first receive a value `n`, send it out, and start again.

To enforce matching of actions, e.g. to arrange for `A` sending to `B`, `B` sending to `C`, and `C` sending to `A`, we make use of a communication function. The function states which actions combine into other actions, e.g. `send_a` and `receive_b` may combine into the action `msg_ab`, similar to a synchronization of actions a and $\bar{a}$



yielding τ in CCS [32]. In `mCRL2`, for successful synchronization it is required that the parameters of the actions, if any, are the same. In our case the net result is communication: a receive action with a parameter bound by a summation gets instantiated by the parameter value of the sending action. On top of this, to constrain the interaction of processes and to prune the state space, we forbid unmatched actions by explicitly listing which actions are allowed to happen, excluding actions that are supposed to resolve into another. Note that in `mCRL2` synchronization is multi-party, hence not restricted to handshaking as in CCS.

For the above three processes we may have:

```
allow( { msg_ab, msg_bc, msg_ca },
 comm( { send_a | receive_b -> msg_ab,
         send_b | receive_c -> msg_bc,
         send_c | receive_a -> msg_ca },
       A || B || C ));
```

The resulting state space of the communicating processes is depicted on the right above. We see that only the allowed actions `msg_ab`, `msg_bc`, and `msg_ca` occur, hence no occurrences of unmatched send and receive actions. Also, in the three cycles the same parameter value is mentioned, viz. either 1, 2, or 3; the infinite sums of the processes `B` and `C` have been resolved.

A system property can be expressed as a formula in a variant of the modal μ-calculus [21]. Subsequently, the property can be verified against a `mCRL2` specification of the system using the model checking facilities of the toolset. Here are some properties that hold for the `Foo` process above:

- `[true*]<true>true`: absence of deadlock, i.e. after any sequence of actions, an action can be done.
- `[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.(<d>true || [true] Y)`: a least-fixed-point construction. Always, after a finite amount of steps, a `d`-action can be done (or deadlock occurs earlier). The smallest set of states `Y` that can do a `d`-action or cannot step outside of `Y`, can be computed by iteration: Start from the empty set $Y_0 = \varnothing$.

Then include state 3 which can do `d`, yielding $Y_1 = \{3\}$. Then add states 1 and 4 since their single step leads to Y_1, yielding $Y_2 = \{1, 3, 4\}$. Then include 2 and 5 since all their steps lead to Y_2, yielding $Y_3 = \{1, 2, 3, 4, 5\}$. The next step adds 0 and yields the fixed point $Y_4 = \{0, 1, 2, 3, 4, 5\}$. Since the initial state $0 \in Y_4$, the formula holds.

– `mu Y.((nu Z.(<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 `d` and `e` is possible.

The modal μ-calculus, a.k.a. the 'Logic of Everything', is renowned to be highly expressive and to subsume temporal logics like LTL and (A)CTL [17, 10, 14]. The model-checking approaches of [12, 6] are based on LTL, that of [26] on the multi-valued modal μ-calculus, and those of [25, 2] on (A)CTL. Only the approach of [6] is implemented, viz. in the Maude toolset (`maude.cs.uiuc.edu`). The appeal of the modal μ-calculus variant in `mCRL2` exploited here is the possibility to quantify over data. Moreover, well-chosen hiding of actions and minimization with respect to one of the process equivalences offered by the `mCRL2` toolset (e.g. trace equivalence, weak and branching bisimulation [32, 20]) allow to narrow the state space and to focus on specific behavioral aspects. The latter technique can significantly reduce a state space with millions of states to a state space of a few dozens, making visual inspection feasible.

4 Modeling of the Running Example

To model the product family underlying our example in `mCRL2` we follow the approach set out in [5]. We will have two main processes: a feature selection process and a process (actually a combination of a number of component processes together with a driver process) representing an actual product of the family.

First, a valid feature set is selected by the three-stage non-deterministic process `Sel`. The resolution of the non-determinacy leads to a product configuration which is checked for its consistency with global constraints. First, a breadth-first traversal of the feature model selects features, taking 'mandatority' and possible local constraints, like m-out-of-n selection, into account. Second, cross-tree constraints are checked and violating configurations result in a transition to an error state. In our example we have two such constraints: the mutual exclusion of `Dollar` vs. `Cappuccino`, and the required inclusion of `Ringtone` in the presence of `Cappuccino`. Finally, attribute constraints are checked. For the example it is required for the selected features not to exceed a cost limit of 30. Also here, violating configurations are forced to transit to an error state.

Configurations that have passed through all three stages successfully are genuine sets of features complying to all requirements as expressed by the attributed feature model. These configurations are passed as an argument to the process that represents the corresponding product. An excerpt of the process `Sel` is depicted next.[4]

[4] The full `mCRL2` specification is available from `http://www.win.tue.nl/~evink/research/mCRL2`.

```
proc Sel(st:Int,fs:FSet) =
  %% feature selection
  ( st == 0 ) -> ( ( M in fs ) -> ( setS . Sel(1, ins(S,fs) ) ) ) +
  ( st == 1 ) -> ( ( M in fs ) -> ( setO . Sel(2, ins(O,fs) ) ) ) +
  ( st == 2 ) -> ( ( M in fs ) -> (
      tau  . Sel(3, fs ) +  setR . Sel(3, ins(R,fs) ) ) ) +
    ...
  ( st == 5 ) -> ( ( O in fs ) -> (
      setD . Sel(6, ins(D,fs) ) +  setE . Sel(6, ins(E,fs) ) ) ) +
    ...
  %% cross-tree constraints
  ( st == 8 ) -> ( ( ( D in fs ) && ( P in fs ) ) ->
      dollar_cappo_fault( fs ) . Sel(801,fs) <> skip . Sel(9,fs) ) +
    ...
  %% attribute constraints
  ( st == 10 ) -> ( ( tcost(fs) <= 30 ) ->
      attr_ok . cost( tcost(fs) ) . put_config(fs) <>
        attr_fault( fs , tcost(fs) ) . Sel(1001,fs) ) +
    ...
```

The selection process `Sel` has two parameters: a local state `st` represented by an integer, and a feature set `fs` represented as a sorted list of features without duplicates. As we will see later, the selection process starts with the root feature `Machine`, abbreviated as `M`, chosen. Thus initially we have `Sel(0,[M])`. In state `0` of the `Sel` process, since `M in fs` holds, the mandatory feature `Sweet` is added to the current feature set; `Sel` continues in state `1` and feature set `ins(S,fs)`. Similarly, in state `1` the mandatory coin feature is included. In state `2`, the optional ringtone feature is handled, which may or may not be selected, leading to a non-deterministic choice between `tau.Sel(3,fs)` and `setR.Sel(3,ins(R,fs))`. In the former option the `Sel` parameters remain unchanged, in the latter the `R`-feature is added to the current feature set. In the same vein, but slightly different, is the 1-out-of-2 selection of the dollar or the euro feature. Here either choice leads to an update of the current feature set.

As outcome of the first stage of the `Sel` process a feature set `fs` is selected that is consistent with the local feature requirements (mandatory, optional, alternative, etc.). Next cross-tree constraints are checked for `fs`. For example, the mutual exclusion of `Dollar` vs. `Cappuccino` is captured by the test in state `8` of `Sel`. If both the `D` and the `P` feature are present in `fs` the constraint is violated and control is transferred to the error state `801`. Otherwise the process continues checking for the next cross-tree constraint. Finally, in the third stage of `Sel`, attribute constraints are checked, in the case of the example the costs should not be higher than `30`. If it is too high, `Sel` moves to a specific error state. If it is sufficiently low, i.e. `tcost(fs) <= 30`, the attribute is marked as OK, the costs are outputted, and moreover, via the action `put_config(fs)`, the eligible feature set is passed on to the product process modeling actual behavior.

The potential behavior of our example is shown by the LTS in Figure 2 (left), which is the one from [2] with simplified money insertion. In line with [12, 6],

transitions are assumed to be tagged with a feature (not made explicit here for readability). An action can only occur in a product if the corresponding feature is selected for the product, i.e. the feature set that configures the product needs to be checked for the presence of a feature for feature-dependent actions to occur.

From starting state `0`, a coin (either a dollar or a euro) can be inserted. Control then moves to state `1`. There, either the user cancels the interaction with the machine, and control returns to state `0`, or chooses for sugar or no sugar, and control moves to state `2` or `3`, respectively. The user chooses one of the available drinks, a choice of coffee, tea or cappuccino, and control reaches state `6` or `7`, `5` or `8`, `4` or `9`, depending on the choices made. Then the necessary ingredients are added, control moves to state `7`, `8`, or `9` if sugar was added, and subsequently to state `12` after the drink has been poured. Note that a cappuccino request leads to an interleaving of pouring coffee and milk. Next, control moves to state `13`, ringing or not according to the feature set. Then the user can take her/his cup and control returns to state `0`.

In our `mCRL2` encoding, a product as given by an eligible feature set `fs` is represented by a parallel composition of six component processes, one for each of the features `Sweet`, `Coin`, `Ringtone`, `Beverage`, `Cancel` and one for the root feature `Machine`. After being woken up by the selection process, the process belonging to the product with feature set `fs` as configuration is given by a system of seven parallel processes

```
Driver(0) || Sweet(fs) || Coin(fs) ||                              (1)
   Ringtone(fs) || Beverage(fs) || Cancel(fs) || Machine
```

It is stressed that although not explicitly mentioned, the selection process `Sel` is always part of the `mCRL2` specifcations considered below. By taking feature selection and coupled product behaviour as a whole, verification can be done at the family level, rather than for each product or subset of products separately. See [5] for more detail.

In order to enforce the proper control flow the component processes are put in parallel with a driver process `Driver`. For example, the `Coin` process is given by

```
proc Coin(fs:FSet) =
  cmp_start(0) . (
    ( D in fs ) -> insert(dollar) . raise(1) . Coin() +
    ( E in fs ) -> insert(euro) . raise(1) . Coin() ) +
  cmp_start(1) .  cancel . raise(0) . Coin() ;
```

On a `drv_start(0)` request of the driver, the `Coin` component can execute the matching `cmp_start(0)` action upon which either the action `insert(dollar)` or the action `insert(euro)` follows. As the preceding feature selection process has enforced that exactly one of the two features `D` and `E` is included in the product feature set `fs`, exactly one of the two actions can be taken. After executing either of them, the `Coin` process raises that the driver should proceed in state `1`. The action `raise(1)` of `Coin` is matched by the action `catch(1)` of `Driver`.

The driver process is relatively simple. It proclaims the current state of the product via a `drv_start(st)` action, allowing any component with an active transition in state `st` to perform an action. Next it catches the new state number `st'`, raised by the component, and the driving starts anew from that state:

```
proc Driver(st:Int) =
  drv_start(st) . sum st':Int . catch(st') . Driver(st') ;
```

Now, in state `1` three actions are possible: a `cancel` from the `Cancel` process, or a `sugar` action or a `no_sugar` action from the `Sweet` process, partly defined by

```
proc Sweet(fs:FSet) =
  cmp_start(1) . ( S in fs ) -> sugar . raise(2) . Sweet() +
  cmp_start(1) . ( S in fs ) -> no_sugar . raise(3) . Sweet() +
  ...
```

The non-determinacy in state `1` reflects the user's choice. (S)he can press a button to cancel the interaction with the coffee machine or opt for sugar or no sugar. However, the action `cancel` will only be offered if the feature set `fs` of the product actually holds the `X`. This explains the guarding by the check for `X` in `fs` of the `cmp_start(1)` action of `Cancel` in

```
proc Cancel(fs:FSet) =
  ( X in fs ) -> cmp_start(1) . cancel . raise(0) . Cancel() ;
```

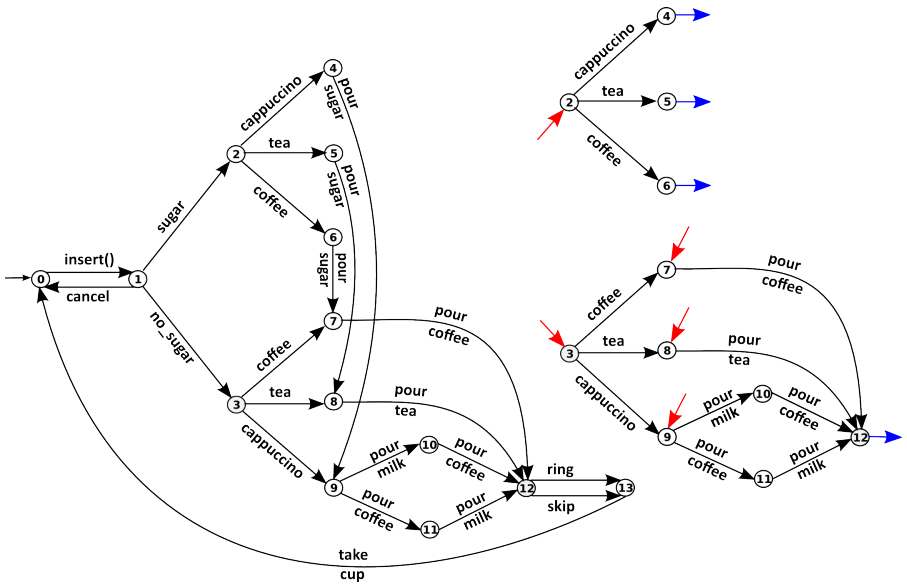


Fig. 2. LTS modeling family behavior (left) and its beverage component (right)

5 Analysis of the Running Example

To illustrate our approach to feature-oriented modular verification we focus on the beverage component. Its isolated sub-LTS is depicted in Figure 2 (right).

Note that the transitions belonging to other components do not appear in this representation. However, next to the component's behavior the interface to the environment is given, as mediated through the driver. It can receive requests from the driver on its unlabeled incoming (re-)entry transitions (red), while it can provide input to the driver via its unlabeled outgoing exit transitions (blue).

An obvious requirement for the beverage component to hold is that coffee is delivered at least when coffee is asked for. We may express this by the formula

```
[ true* . coffee . (!pour_coffee)* . take_cup ] false
```

However, this does not guarantee that a `pour_coffee` will take place, rather that a `take_cup` is avoided. Another disadvantage of the formula is that the action `take_cup` does not belong to the beverage component but to the machine component instead. This can be remedied using a minimal fixed point construction. This is reflected by the modal μ-calculus formula

```
[ true*.coffee ]( mu X. [ !pour_coffee ] X )          (2)
```

i.e., after a `coffee` action a `pour_coffee` action happens within a finite number of steps. Thus, a coffee request is answered by the pouring of coffee eventually. To ensure that the `pour_coffee` action matches the occurrences of the `coffee` action mentioned, we can forbid that the beverage component is left:

```
[true*.coffee](mu X. ([ !pour_coffee ] X && [ event(12) ] false ))
```

i.e., after a coffee request coffee will be poured eventually and this happens before the beverage component is exited. Note that the `mCRL2` toolset supports modal μ-calculus with data.

If control enters the beverage component via the `no_sugar` entrance state `3`, clearly property (2) holds. After the `coffee` request of the transition from state `3` to state `7` there is no other action than the `pour_coffee` action of the transition from state `7` to state `12`, as control may enter at state `7`, but may not leave. This is all different when control enters the beverage component via the `sugar` entrance state `2`. Then a coffee request issued by the transition from state `2` to state `6` relies on the environment, in particular the sweet component, for a transition (or sequence of transitions as far as the beverage component is concerned) leading to state `7` so that the `pour_coffee` action becomes enabled.

In fact, as the actions `coffee` and `pour_coffee` belong to the beverage component, it suffices that the environment caters for (i) a transfer from state `4` to state `9`, (ii) a transfer from state `5` to state `8`, and (iii) a transfer from state `6` to state `7`. Additionally, the environment is expected to allow a return to states `2` and `3` after the beverage component is left via state `12`.

To model this in `mCRL2` we introduce a process `BeverageStub`, a stub for the behavior of the environment of the `Beverage` component, given by

```
proc BeverageStub =
  cmp_start(0)  . other . ( raise(2) + raise(3) ) . BeverageStub +
  cmp_start(4)  . other . raise(9) . BeverageStub +
```

```
cmp_start(5)  . other . raise(8) . BeverageStub +
cmp_start(6)  . other . raise(7) . BeverageStub +
cmp_start(12) . other . ( raise(2) + raise(3) ) . BeverageStub ;
```

With `BeverageStub` in place, rather than considering the 7-process system used previously, i.e. the driver and the six component processes in equation (1), we can now deal with three processes only:

```
Driver(0) || Beverage(fs) || BeverageStub
```
(3)

Note the summand with the `cmp_start(0)` action of `BeverageStub` to be able to pick up the simulation of the environment right from the start. Also, the stub process does not have a feature set as an argument. This is in line with the intuition that it is not for the beverage component to make specific assumptions on the configuration of the environment nor on its behavior, beyond the entering and exiting of control regarding the beverage component itself. Rephrased more technically, the 7-process system of the driver and all six components is branching bisimilar to the 3-process system of the driver, beverage component and its stub. Therefore, modal μ-formulas in the CTL*-fragment without the next operator [15], like the one of (2), equally hold for the two systems.

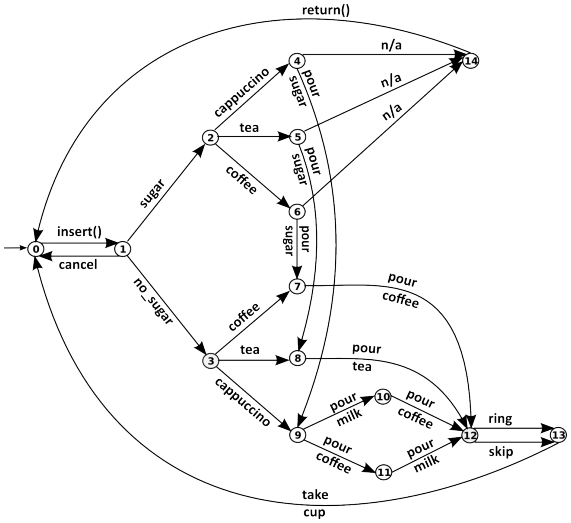


Fig. 3. LTS modeling alternative family behavior

Next we modify the sweet component according to the LTS in Figure 3. Now, as an improved service, whenever the machine is out of sugar, it returns the inserted money instead of delivering the chosen beverage (without sugar). Apart from an adaptation of the `Sweet` process to accommodate the 'not-available' action `n/a` from states `4`, `5`, and `6` to the new state `14`, the `Cancel` process is extended to return the inserted money via the `return` transition from state `14` back to state `0`.

Because of the transition to the new state `14`, from the point of view of the beverage component, the flow of control may unexpectedly be diverted. A request for coffee with sugar, i.e. the action `coffee` leaving state `2`, may be followed by

the action n/a leading away from further handling by the beverage component. Thus no action pour_coffee is performed (in case the n/a is always taken at that point). So, counterintuitively, while the sweet and cancel component have changed, a basic requirement for the beverage component becomes violated.

The crucial point we want to underline is that no model checking is needed for the designer to be warned. The full 7-process system (1), with the 'improved' sweet and cancel component, is no longer branching bisimilar to the smaller 3-process system (3) that includes the beverage stub. The ltscompare tool in the mCRL2 toolset, which can decide e.g. on the branching-bisimilarity of two systems, finds this quickly. Thus, a simple check suffices to alert the designer that a property of the component that was valid previously, may not hold anymore.

If this new behavior to deal with the lack of sugar is to be maintained, the coffee vs. pour_coffee requirement needs to be weakened. One may propose

```
[ true*.coffee ]( mu X. ([ !( pour_coffee || return ) ] X ))
```

i.e., a coffee request is answered by either pouring the coffee or a refund. As the action return does not belong to the action set of the Beverage process, the system to be model checked needs to comprise the Cancel process as well, possibly combined with an adapted stub process to replace the other four components. However, from a scalability perspective it is less attractive to deal with specific combinations of components.

Reconsidering the very idea of isolating the beverage component, we need to make the distinction between the 'acceptable' action pour_sugar and the 'non-acceptable' action n/a visible in the stub process. We introduce for the process BeverageStub2 below the action escape to represent behavior that may/will affect the behavior of the beverage component, besides the indifferent action other that was used earlier. The code for BeverageStub2 reads as follows.

```
proc BeverageStub2 =
  cmp_start(0) .  other . ( raise(2) + raise(3) ) . BeverageStub2 +
  cmp_start(4) .(
    escape . ( raise(2) + raise(3) ) . BeverageStub2 +
    other . raise(9) . BeverageStub2 ) +
  cmp_start(5) . (
    escape . ( raise(2) + raise(3) ) . BeverageStub2 +
    other . raise(8) . BeverageStub2 ) +
  cmp_start(6) . (
    escape . ( raise(2) + raise(3) ) . BeverageStub2 +
    other . raise(7) . BeverageStub2 ) +
  cmp_start(12) . other . ( raise(2) + raise(3) ) . BeverageStub2;
```

E.g., the stub captures that in state 6, right after the request for coffee, control either 'escapes' from the neighborhood of the beverage component and may return via the entrance state 2 or 3, or control remains in the vicinity of the component, having 'other' activity but picking up the beverage thread in state 7. With the BeverageStub2 in place, the enhanced 7-process system and adapted 3-process system can again be shown to be branching bisimilar.

6 Related Work

In this section, we continue the discussion of related work on the compositional verification of software product lines (SPL) initiated in the introduction. In [29], improving part of the pioneering work of [19, 27, 28] mentioned already, an incremental compositional model checking approach for SPL is presented. It uses 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 variation point obligations. Whenever possible, verification results are reused in an incremental fashion within the product being composed, which reduces the overall verification effort, but the approach does not aim to reuse properties of behavioral feature models across different products.

In [36], an existing compositional verification technique for safety properties of flow-graph behavior of general-purpose programs is adapted to programs from the SPL domain, that are organized according to a hierarchical variability model defining variation points and interfaces. 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 [33], feature Petri nets are introduced as a modular (feature- and interface-based) behavioral modeling formalism. A few correctness criteria, based on bisimulation, for the preservation of properties in composed models are given. This is a promising approach that deserves further study, as does the precise relationship with our approach, apart from the fact that model checking is not addressed nor the question for the reuse of verification results.

In [31], for each feature of an SPL 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 tool `SPLEnD` makes use of the `SPIN` model checker (`spinroot.com`) to implement such conformance checking. Reuse of verification results is not considered. Recent work on delta-oriented SPL analysis using `mCRL2` is reported in [30].

Some of the other behavioral variability models mentioned in the introduction come with a special-purpose tool for SPL model checking. `SNIP` [11] is a model checker for product lines modeled as featured transition systems [12] specified in a language based on that of `SPIN`. The tool `VMC` [7] (`fmt.isti.cnr.it/vmc`) is a model checker for product lines modeled as modal transition systems with additional variability constraints [2] specified in a modal process algebra. Neither of these currently make use of modular or compositional verification.

7 Discussion and Future Work

We have presented a proof-of-concept of a feature-oriented modular verification technique for analyzing the behavior of SPL with `mCRL2`. We use branching bisimulation techniques to isolate the behavior of a specific feature (set) by abstracting from the environment. This eases the model checking and allows the result to be

reused in other settings; if adapted behaviour leads to an environment that is brnaching bisimilar, say, to the stub used, a verified property is also valid in the new situation. `mCRL2` is a toolset that has already shown its merits in dealing with huge state spaces consisting of billions of states. Although at present stubs are crafted manually, we believe that scalable verification of SPL can successfully be achieved with `mCRL2` along the lines of the modular verification strategy illustrated in this paper. A demonstration of this fact is left for future work.

Our approach thus 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 [24] 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 [34]) 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 verify new properties that the new system should now satisfy. This is complicated by the well-known fact that features may interact. We have seen this problem arise in our running example when we modeled the 'improved' service signaling the lack of sugar in a coffee machine.

This brings us to concrete future work on our running example. If we consider the notion of a neighborhood of a component, then we can distinguish (re-)entry points, exit points, and interrupt points. The latter come in these three flavors:

Type 1 like `pour_sugar` of the `other` or `continue` type; other components do their business, but control is picked up again by the component at hand.

Type 2 like `n/a` of the `escape` or `break` type; another component takes over control and control re-enters the component at a re-entry point rather than continuing its thread.

Type 3 (not encountered in the above discussion) of the `diverge` type; another component takes over control and the component itself is never visited again.

We may claim that with these three corresponding types of actions (`continue`, `break`, and `diverge`) a sufficiently rich class of stubs can be constructed that can simulate the environment. Checking a component property would then mean:

1. Verify the property for the 3-process system of driver, component, and stub.
2. Check branching bisimilarity for the 3-process system and the complete all-process system.

Finally, we need to identify which class of properties exactly fits within our approach. Evidently, further work is to be done on larger examples and convincing case studies to support our claims.

References

1. M. Abadi and L. Lamport. Conjoining Specifications. *ACM Transactions on Programming Languages and Systems*, 17(3):507–534, 1995.
2. P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. Formal Description of Variability in Product Families. In *Proc. SPLC'11*, pages 130–139. IEEE, 2011.
3. 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 *Proc. FormaliSE'13*, pages 43–49. IEEE, 2013.
4. J. C. M. Baeten, T. Basten, and M. A. Reniers. *Process Algebra: Equational Theories of Communicating Processes*. Cambridge University Press, 2010.
5. M. H. ter Beek and E. P. de Vink. Using mCRL2 for the analysis of software product lines. In *Proc. FormaliSE'14*, pages 31–37. IEEE, 2014.
6. M. H. ter Beek, A. Lluch-Lafuente, and M. Petrocchi. Combining declarative and procedural views in the specification and analysis of product families. In *Proc. SPLC'13*, volume 2, pages 10–17. ACM, 2013.
7. M. H. ter Beek, F. Mazzanti, and A. Sulova. VMC: A Tool for Product Variability Analysis. In D. Giannakopoulou and D. Méry, editors, *FM'12*, volume 7436 of *LNCS*, pages 450–454. Springer, 2012.
8. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Information Systems*, 35(6), 2010.
9. P. Borba, M. B. Cohen, A. Legay, and A. Wąsowski. Analysis, Test and Verification in The Presence of Variability (Dagstuhl Seminar 13091). *Dagstuhl Reports*, 3(2):144–170, 2013.
10. J. C. Bradfield and C. Stirling. Modal μ-calculi. In P. Blackburn, J. F. A. K. van Benthem, and F. Wolter, editors, *Handbook of Modal Logic*, volume 3 of *Studies in Logic and Practical Reasoning*, pages 721–756. Elsevier, 2007.
11. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Model checking software product lines with SNIP. *International Journal on Software Tools for Technology Transfer*, 14(5):589–612, 2012.
12. A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, and J.-F. Raskin. Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking. *IEEE Transactions on Software Engineering*, 39(8):1069–1089, 2013.
13. S. Cranen. Model Checking the FlexRay Startup Phase. In M. Stoelinga and R. Pinger, editors, *FMICS'12*, volume 7437 of *LNCS*, pages 131–145. Springer, 2012.
14. R. De Nicola and F. W. Vaandrager. Action versus State based Logics for Transition Systems. In I. Guessarian, editor, *Semantics of Systems of Concurrent Processes*, volume 469 of *LNCS*, pages 407–419. Springer, 1990.
15. R. De Nicola and F. W. Vaandrager. Three logics for branching bisimulation. *Journal of the ACM*, 42(2):458–487, 1995.
16. W. P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *LNCS*. Springer, 1997.
17. E. A. Emerson. Model Checking and the Mu-calculus. In N. Immerman and P. G. Kolaitis, editors, *Proc. of DIMACS Workshop on Descriptive Complexity and Finite Models*, volume 31 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 185–214. AMS, 1996.
18. D. Fischbein, S. Uchitel, and V. A. Braberman. A foundation for behavioural conformance in software product line architectures. In R. M. Hierons and H. Muccini, editors, *Proc. ROSATEA'06*, pages 39–48. ACM, 2006.

19. K. Fisler and S. Krishnamurthi. Modular Verification of Collaboration-Based Software Designs. In *Proc. ESEC/FSE'01, Vienna*, pages 152–163. ACM, 2001.
20. R. J. v. Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *Journal of the ACM*, 43(3):555–600, 1996.
21. J. F. Groote and R. Mateescu. Verification of Temporal Properties of Processes in a Setting with Data. In A. M. Haeberer, editor, *AMAST'98*, volume 1548 of *LNCS*, pages 74–90. Springer, 1998.
22. J. F. Groote, A. Mathijssen, M. A. Reniers, Y. S. Usenko, and M. J. van Weerdenburg. Analysis of Distributed Systems with mCRL2. In M. Alexander and W. Gardner, editors, *Process Algebra for Parallel and Distributed Processing*, pages 99–128. Chapman & Hall, 2009.
23. O. Grumberg and D. E. Long. Model Checking and Modular Verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.
24. K. Laster and O. Grumberg. Modular Model Checking of Software. In B. Steffen, editor, *TACAS'98*, volume 1384 of *LNCS*, pages 20–35. Springer, 1998.
25. K. Lauenroth, K. Pohl, and S. Töhning. Model Checking of Domain Artifacts in Product Line Engineering. In *Proc. ASE'09, Auckland*, pages 269–280. IEEE, 2009.
26. M. Leucker and D. Thoma. A Formal Approach to Software Product Families. In T. Margaria and B. Steffen, editors, *ISoLA'12*, volume 7609 of *LNCS*, pages 131–145. Springer, 2012.
27. H. C. Li, K. Fisler, and S. Krishnamurthi. The Influence of Software Module Systems on Modular Verification. In D. Bosnacki and S. Leue, editors, *SPIN'02*, volume 2318 of *LNCS*, pages 60–78. Springer, 2002.
28. H. C. Li, S. Krishnamurthi, and K. Fisler. Interfaces for Modular Feature Verification. In *Proc. ASE'02, Edinburgh*, pages 195–204. IEEE, 2002.
29. J. Liu, S. Basu, and R. R. Lutz. Compositional model checking of software product lines using variation point obligations. *Automated Software Engineering*, 18(1):39–76, 2011.
30. M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. DeltaCCS: A Core Calculus for Behavioral Change. In this volume, 2014.
31. J.-V. Millo, S. Ramesh, S. N. Krishna, and G. K. Narwane. Compositional Verification of Software Product Lines. In E. B. Johnsen and L. Petre, editors, *IFM'13*, volume 7940 of *LNCS*, pages 109–123. Springer, 2013.
32. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
33. R. Muschevici, J. Proença, and D. Clarke. Modular Modelling of Software Product Lines with Feature Nets. In G. Barthe, A. Pardo, and G. Schneider, editors, *SEFM'11*, volume 7041 of *LNCS*, pages 318–333. Springer, 2011.
34. A. Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 123–144. Springer, 1985.
35. 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 *Proc. CCGrid'12*, pages 335–343. IEEE, 2012.
36. I. Schaefer, D. Gurov, and S. Soleimanifard. Compositional Algorithmic Verification of Software Product Lines. In B. K. Aichernig, F. S. de Boer, and M. M. Bonsangue, editors, *FMCO'10*, volume 6957 of *LNCS*, pages 184–203. Springer, 2012.
37. 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. *Software Quality Journal*, 20(3-4):487–517, 2012.

A mCRL2 Code

1 driver and 6 components

```
sort
  Feature = struct M | S | O | R |
    B | X | D | E | P | T | C ;
  FSet = List( Feature );
  Currency = struct Dollar | Euro ;

act
  setS, setO, setR, setB, setX,
  setD, setE, setP, setT, setTP, setC ;
  ctc_tree_ok, attr_ok ;
  dollar_cappo_fault,
  ring_cappo_fault : FSet ;
  attr_fault : FSet # Int ;
  cost : Int ;
  loop ;

act
  put_config, get_config, set_config : FSet ;

act
  insert : Currency;
  sugar, no_sugar, pour_sugar;
  coffee, tea, cappuccino,
  pour_coffee, pour_tea, pour_milk;
  ring, skip;
  cancel;
  take_cup;
  drv_start, cmp_start, start : Int;
  raise, catch, event : Int;
  nothing;

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
  ins: Feature # FSet -> FSet;
var
  ft, ft': Feature;
  fset: FSet;
eqn
  ins(ft, []) = [ft];
  (ft < ft')  ->
    ins(ft, ft' |> fset) = ft |> ft' |> fset;
  (ft == ft') ->
    ins(ft, ft' |> fset) = ft' |> fset;
  (ft > ft')  ->
    ins(ft, ft' |> fset) =
      ft' |> ins(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, ins(S,fs) )
  ) ) +
  ( st == 1 ) -> (
    ( M in fs ) -> (
      setO .  Sel(2, ins(O,fs) )
  ) ) +
  ( st == 2 ) -> (
    ( M in fs ) -> (
      tau  . Sel(3, fs ) +
      setR . Sel(3, ins(R,fs) )
  ) ) +
```

```
  ( st == 3 ) -> (
    ( M in fs ) -> (
      setB .  Sel(4, ins(B,fs) )
  ) ) +
  ( st == 4 ) -> (
    ( M in fs ) ->
      tau .  Sel(5, fs ) +
      setX . Sel(5, ins(X,fs) )
  ) +
  ( st == 5 ) -> (
    ( O in fs ) -> (
      setD . Sel(6, ins(D,fs) ) +
      setE . Sel(6, ins(E,fs) )
  ) ) +
  ( st == 6 ) -> (
    ( B in fs ) -> (
      tau   . Sel(7, fs ) +
      setT  . Sel(7, ins(T,fs) ) +
      setP  . Sel(7, ins(P,fs) ) +
      setTP . Sel(7, union([T,P],fs) )
  ) ) +
  ( st == 7 ) -> (
    ( B in fs ) -> (
      setC . Sel(8, ins(C,fs) )
  ) ) +
  %% cross-tree constraints
  ( st == 8 ) -> (
    ( ( D in fs ) && ( P in fs ) ) ->
      dollar_cappo_fault( fs ) .
        Sel(801,fs) <>
    tau . Sel(9,fs)
  ) +
  ( st == 9 ) -> (
    ( !( R in fs ) && ( P in fs ) ) ->
      ring_cappo_fault( fs ) .
        Sel(901,fs) <>
    ctc_tree_ok . Sel(10,fs)
  ) +
  %% attribute constraints
  ( st == 10 ) -> (
    ( tcost(fs) <= 30 ) ->
      attr_ok . cost( tcost(fs) ) .
        put_config(fs) <>
    attr_fault( fs , tcost(fs) ) .
      Sel(1001,fs) ) +
  %% loop on error states
  ( ( st == 801 ) ||
    ( st == 901 ) || ( st == 1001 ) ) ->
      loop . Sel() +
  delta ;

proc Driver(drv_st:Int) =
  drv_start(drv_st) .
    sum e:Int . catch(e) .
      Driver(e) ;

proc Sweet(fs:FSet) =
  cmp_start(1) .
    ( S in fs ) -> sugar .
      raise(2) . Sweet() +
  cmp_start(1) .
    ( S in fs ) -> no_sugar .
      raise(3) . Sweet() +
  cmp_start(4) .
    pour_sugar . raise(9) . Sweet() +
  cmp_start(5) .
    pour_sugar . raise(8) . Sweet() +
  cmp_start(6) .
    pour_sugar . raise(7) . Sweet() +
  delta ;

proc Coin(fs:FSet) =
  cmp_start(0) . (
    ( D in fs ) ->
      insert(Dollar) . raise(1) . Coin() +
    ( E in fs ) ->
      insert(Euro) . raise(1) . Coin() ) +
  delta ;

proc Ringtone(fs:FSet) =
  cmp_start(12) . (
    ( R in fs ) ->
      ring . raise(13) . Ringtone() +
    !( R in fs ) ->
      skip . raise(13) . Ringtone() ) +
  delta;

proc Beverage(fs:FSet) =
  cmp_start(2) . (
    ( C in fs ) ->
      coffee . raise(4) . Beverage() +
    ( T in fs ) ->
      tea . raise(5) . Beverage() +
    ( P in fs ) ->
      cappuccino . raise(6) . Beverage() ) +
  cmp_start(3) . (
    ( C in fs ) ->
      coffee . raise(9) . Beverage() +
    ( T in fs ) ->
      tea . raise(8) . Beverage() +
    ( P in fs ) ->
      cappuccino . raise(7) . Beverage() ) +
  cmp_start(7) . (
    pour_milk . raise(10) . Beverage() +
    pour_coffee . raise(11) . Beverage() ) +
  cmp_start(8) .
    pour_tea . raise(12) . Beverage() +
  cmp_start(9) .
    pour_coffee . raise(12) . Beverage() +
  cmp_start(10) .
    pour_coffee . raise(12) . Beverage() +
  cmp_start(11) .
    pour_milk . raise(12) . Beverage() +
  delta;

proc Cancel(fs:FSet) =
  ( X in fs ) ->
    cmp_start(1) .
      cancel . raise(0) . Cancel() +
  delta ;

proc Machine =
  cmp_start(13) .
    take_cup . raise(0) . Machine +
  delta;

init
  hide( {
    cancel, insert,
    sugar, no_sugar, pour_sugar,
    ring, skip,
    take_cup,
    %%
    setS, setO, setR, setB, setX,
    setD, setE, setP, setT, setTP, setC,
    ctc_tree_ok, dollar_cappo_fault,
```

```
    ring_cappo_fault,
    attr_ok, attr_fault,
    cost,
    loop,
    set_config,
    start, event,
    nothing },
  allow( {
    setS, setO, setR, setB, setX,
    setD, setE, setP, setT, setTP, setC,
    ctc_tree_ok, dollar_cappo_fault,
    ring_cappo_fault,
    attr_ok, attr_fault,
    cost,
    loop,
    set_config,
    insert, cancel,
    sugar, no_sugar, pour_sugar,
    coffee, tea, cappuccino,
    pour_coffee, pour_tea, pour_milk,
    ring, skip,
    take_cup,
    start, event,
    nothing },
  comm( {
    put_config |
      get_config | get_config | get_config |
        get_config | get_config -> set_config,
    drv_start | cmp_start -> start,
    raise | catch -> event },
  Sel(0,[M]) ||
    %%
  Driver(0) ||
  ( sum fs:FSet . get_config(fs) .
    Sweet(fs) ) ||
  ( sum fs:FSet . get_config(fs) .
    Coin(fs) ) ||
  ( sum fs:FSet . get_config(fs) .
    Ringtone(fs) ) ||
  ( sum fs:FSet . get_config(fs) .
    Beverage(fs) ) ||
  ( sum fs:FSet . get_config(fs) .
    Cancel(fs) ) ||
  Machine
    %%
)));
```

Isolated beverage component with stub

```
sort
  Feature = struct M | S | O | R |
    B | X | D | E | P | T | C ;
  FSet = List( Feature );
  Currency = struct Dollar | Euro ;

act
  setS, setO, setR, setB, setX,
  setD, setE, setP, setT, setTP, setC ;
  ctc_tree_ok, attr_ok ;
  dollar_cappo_fault,
  ring_cappo_fault : FSet ;
  attr_fault : FSet # Int ;
  cost : Int ;
  loop ;

act
  put_config, get_config, set_config : FSet ;

act
  coffee, tea, cappuccino,
  pour_coffee, pour_tea, pour_milk;
  drv_start, cmp_start, start : Int;
  raise, catch, event : Int;
  other,
  nothing;

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
  ins: Feature # FSet -> FSet;
var
  ft, ft': Feature;
  fset: FSet;
eqn
  ins(ft, []) = [ft];
  (ft < ft')  ->
    ins(ft, ft' |> fset) =
      ft |> ft' |> fset;
  (ft == ft') ->
    ins(ft, ft' |> fset) =
      ft' |> fset;
  (ft > ft')  ->
    ins(ft, ft' |> fset) =
      ft' |> ins(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, ins(S,fs) )
  ) ) +
  ( st == 1 ) -> (
    ( M in fs ) -> (
      setO .  Sel(2, ins(O,fs) )
  ) ) +
  ( st == 2 ) -> (
    ( M in fs ) -> (
      tau  . Sel(3, fs ) +
      setR . Sel(3, ins(R,fs) )
  ) ) +
  ( st == 3 ) -> (
    ( M in fs ) -> (
      setB .  Sel(4, ins(B,fs) )
  ) ) +
  ( st == 4 ) -> (
    ( M in fs ) ->
      tau .  Sel(5, fs ) +
      setX . Sel(5, ins(X,fs) )
  ) +
  ( st == 5 ) -> (
    ( O in fs ) -> (
      setD . Sel(6, ins(D,fs) ) +
      setE . Sel(6, ins(E,fs) )
  ) ) +
  ( st == 6 ) -> (
    ( B in fs ) -> (
      tau   . Sel(7, fs ) +
      setT  . Sel(7, ins(T,fs) ) +
      setP  . Sel(7, ins(P,fs) ) +
      setTP . Sel(7, union([T,P],fs) )
  ) ) +
  ( st == 7 ) -> (
    ( B in fs ) -> (
      setC . Sel(8, ins(C,fs) )
  ) ) +
  %% cross-tree constraints
  ( st == 8 ) -> (
    ( ( D in fs ) && ( P in fs ) ) ->
      dollar_cappo_fault( fs ) .
        Sel(801,fs) <>
    tau . Sel(9,fs)
  ) +
  ( st == 9 ) -> (
    ( !( R in fs ) && ( P in fs ) ) ->
      ring_cappo_fault( fs ) .
        Sel(901,fs) <>
    ctc_tree_ok . Sel(10,fs)
  ) +
  %% attribute constraints
  ( st == 10 ) -> (
    ( tcost(fs) <= 30 ) ->
      attr_ok . cost( tcost(fs) ) .
        put_config(fs) <>
    attr_fault( fs , tcost(fs) ) .
      Sel(1001,fs) ) +
  %% loop on error states
  ( ( st == 801 ) ||
    ( st == 901 ) || ( st == 1001 ) ) ->
      loop . Sel() +
  delta ;

proc Driver(st:Int) =
  drv_start(st) .
    sum st':Int . catch(st') .
      Driver(st') ;

proc Beverage(fs:FSet) =
  cmp_start(2) . (
    ( C in fs ) ->
      coffee . raise(4) . Beverage() +
    ( T in fs ) ->
      tea . raise(5) . Beverage() +
    ( P in fs ) ->
       cappuccino . raise(6) . Beverage() ) +
  cmp_start(3) . (
    ( C in fs ) ->
      coffee . raise(9) . Beverage() +
    ( T in fs ) ->
       tea . raise(8) . Beverage() +
    ( P in fs ) ->
       cappuccino . raise(7) . Beverage() ) +
  cmp_start(7) . (
    pour_milk . raise(10) . Beverage() +
    pour_coffee . raise(11) . Beverage() ) +
  cmp_start(8) .
    pour_tea . raise(12) . Beverage() +
  cmp_start(9) .
    pour_coffee . raise(12) . Beverage() +
  cmp_start(10) .
    pour_coffee . raise(12) . Beverage() +
  cmp_start(11) .
    pour_milk . raise(12) . Beverage() +
  delta;

proc BeverageStub =
  cmp_start(0) . other .
    ( raise(2) + raise(3) ) . BeverageStub +
  cmp_start(4) . other .
    raise(9) . BeverageStub +
  cmp_start(5) . other .
    raise(8) . BeverageStub +
  cmp_start(6) . other .
    raise(7) . BeverageStub +
  cmp_start(12) . other .
```

```
    ( raise(2) + raise(3) ) . BeverageStub ;

init
  hide( {
    other,
    %%
    setS, setO, setR, setB, setX,
    setD, setE, setP, setT, setTP, setC,
    ctc_tree_ok, dollar_cappo_fault,
    ring_cappo_fault,
    attr_ok, attr_fault,
    cost,
    loop,
    set_config,
    start, event,
    nothing },
  allow( {
    setS, setO, setR, setB, setX,
    setD, setE, setP, setT, setTP, setC,
    ctc_tree_ok, dollar_cappo_fault,
    ring_cappo_fault,
    attr_ok, attr_fault,
    cost,
    loop,
    set_config,
    coffee, tea, cappuccino,
    pour_coffee, pour_tea, pour_milk,
    other,
    start, event,
    nothing },
  comm( {
    put_config | get_config -> set_config,
    drv_start | cmp_start -> start,
    raise | catch -> event },
  Sel(0,[M]) ||
    %%
  Driver(0) ||
  ( sum fs:FSet . get_config(fs) .
    Beverage(fs) ) ||
  BeverageStub
)));
```