Supervisory Controller Synthesis for Product Lines using CIF 3

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

[1] ISTI–CNR, Pisa, Italy
[2] Eindhoven University of Technology, The Netherlands
[3] CWI, Amsterdam, The Netherlands

Abstract. Using the CIF 3 toolset, we illustrate the general idea of controller synthesis for product line engineering for a prototypical example of a family of coffee machines. The challenge is to integrate a number of given components into a family of products such that the resulting behaviour is guaranteed to respect an attributed feature model as well as additional behavioural requirements. The proposed correctness-by-construction approach incrementally restricts the composed behaviour by subsequently incorporating feature constraints, attribute constraints and temporal constraints. The procedure as presented focusses on synthesis, but leaves ample opportunity to handle e.g. uncontrollable behaviour, dynamic reconfiguration, and product- and family-based analysis.

1 Introduction

In the current globalised economy, businesses are eager to offer a myriad of diversified products as a strategy to increase turnover. To reduce development costs and time-to-market, reuse of components (systems as well as software) is becoming common practice. The aim of Software or Systems Product Line Engineering (SPLE) is to institutionalise reuse throughout all phases of product development [37]. According to this paradigm, enterprises shift from the production, maintenance and management of single products to that of a family or product line of related products, amenable to mass customisation. This requires the identification of the core assets of the products in the domain to exploit their commonality and manage their variability, often defined in terms of features. A feature can be seen as an (increment in) functionality of a product that is visible or relevant to a customer. Consequently, to the developer feature models define the combination of features that constitute valid product configurations [13].

While the automated analysis of structural variability models (e.g. the detection of so-called dead or false optional features in feature models) has a long-standing history [13], that of behavioural variability models has received considerable attention only after the landmark paper by Classen et al. [18]. Since product lines often concern massively (re)used and critical applications (like smartphones and cars), indeed it is important to demonstrate that they are not only configured correctly, but also behave correctly.

⋆ Corresponding author: evink@win.tue.nl

Many approaches aim to engineer systems (of systems) that are provably correct with respect to their requirements. At one side of the spectrum, (post-hoc) verification concerns the application of formal analysis techniques *after* a system (specification) has been constructed. Typically, a formal specification of the implemented system, or abstraction thereof, describes the intended behaviour, after which verification techniques like model checking or theorem proving are applied to verify whether the implementation indeed satisfies the specification [39, 2]. While applications of theorem proving in SPLE have concentrated on the analysis of requirements and code [34, 20, 45] with tools like `Coq` and `KeY`, a number of model-checking tools have been equipped to deal with variability in their specification models for application in SPLE. These range from modal transition system [31, 7] and process-algebraic models [24, 8] to tools like `NuSMV` and `mCRL2` [16, 6], as well as dedicated model checkers like `SNIP`, `VMC`, and `ProVeLines` [15, 10, 19]. Research on applying model checking and theorem proving to product lines is also reflected in recent editions of ISoLA [1, 25, 32, 14, 4, 33, 9].

At the other end of the spectrum, the principle of correctness-by-construction has the aim of developing error-free systems from rigorous and unambiguous specifications, based on stringent correctness criteria in each refinement step. Dijkstra and Hoare focussed on the construction of provably correct *programs* based on weakest precondition semantics [21, 28], whereas Hall and Chapman focussed on an effective and economical software development *process*, from user requirements to implementation, based on zero tolerance of defects [26, 27]. We consider another approach to correctness-by-construction, namely synthesis seen as the development of a supervisor (or supervisory controller) in order to coordinate an assembly of (local) components into a (global) system that functions correctly. Supervisory Control Theory (SCT) [38] synthesises a supervisory control model from models of system components and a set of given requirements. Moreover, the ensemble of components controlled by the supervisor satisfies a number of desirable properties, like the possibility to reach stable local states, so-called marker states, and the impossibility to globally disable events under local control. To the best of our knowledge, we are the first to apply supervisory controller synthesis in SPLE.

At Eindhoven University of Technology, the `CIF 3` toolset [11] is developed and maintained. This toolset targets model-based engineering of supervisory controllers and supports such an engineering process by offering functionality for modelling, simulation, visualisation, synthesis, and code generation. More concretely, in this paper, we show how the `CIF 3` toolset [11] can automatically synthesise a single (family) model representing an automaton for each of the valid products of a product line from (i) an attributed feature model, (ii) component behaviour models associated with the features and (iii) additional behavioural requirements like state invariants, event orderings and guards on events (reminiscent of the Feature Transition Systems (FTSs) of Classen et al. [17]). *By construction*, the resulting `CIF 3` model satisfies all feature-related constraints as well as all behavioural requirements that are assumed to be given beforehand. Note that it was not needed to extend the CIF toolset for our purposes. `CIF 3`

moreover allows, among others, the export of such models in a format accepted by the mCRL2 model checker, which can be used to verify arbitrary behavioural properties expressed in the modal μ-calculus with data or its feature-oriented variant of [6]. An important advantage is that both CIF 3 and mCRL2 can be used off-the-shelf, meaning that no additional tools are required. Moreover, it is important to note that the explicit consideration of features as first-class citizens is a completely new way of using the CIF 3 toolset.

We thus present a unifying SCT approach to deal with structural and behavioural variability, i.e. the resulting synthesised supervisory controller not only manages feature models (product generation), but also product line behaviour (variability encoding) and further behavioural requirements (admissible scenarios). The only other integrated approach that we are aware of is a recent extension of the general-purpose modelling language Clafer [3], that was originally designed to unify (attributed) feature models with class and meta-models. Behavioural Clafer [30] provides (i) feature modelling by means of a constraint language reminiscent of Alloy [29], a light-weight class modelling language with an efficient constraint notation and an effective analyser for instance generation, (ii) behavioural variability by means of hierarchical UML state diagrams and automata (in FTS-style) and (iii) additional behavioural constraints (assertions) in the form of scenarios, allowing for (bounded) LTL model checking.

Compared to the CIF 3 toolset, Behavioural Clafer provides first-class support for architectural modelling through Clafer's rich repertoire for structural modelling, but it offers less advanced behavioural modelling facilities, little support for modularisation of feature-based variants (as in Delta-modelling [40]), and no support for controller synthesis. CIF 3 provides ample facilities to model a system's requirements and behaviour. It does so in a highly modular fashion, with a formal and compositional semantics based on (hybrid) transition systems. In fact, although not shown in this paper, CIF 3 allows to describe timed behaviour and supports the translation of timed discrete event models to UPPAAL [12], a tool for modelling, simulation and verification of real-time systems.

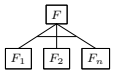
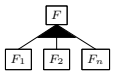
The remainder of the paper is organised as follows: Section 2 briefly introduces the notion of an attributed feature model and describes our running example of a family of coffee machines. Section 3 provides background on supervisory control and illustrates the modelling with CIF 3. In Section 4, we explain, for the product line of coffee machines, how controller synthesis with CIF 3 can be used to bring together feature constraints, component behaviour and system requirements. Section 5 discusses a number of directions for future work.

2 Product Lines

A feature model is a hierarchical and/or-tree of features [13]. A trivial root feature is considered to be present in any product, *mandatory* features must be present provided their parent is, while *optional* features may be present provided their parent is. Exactly one *alternative* feature must be present provided their parent is, and at least one *or* feature must be present whenever their parent is.

A cross-tree constraint either *requires* the presence of another feature for a feature to be present, or it *excludes* two features to be both present. In an attributed feature model, the primitive features (leaves of the tree) are moreover equipped with a non-functional attribute, like cost or weight, and complex constraints over features. Attributes further constrain the feature configuration process, in particular by limiting the cost or weight of features, or of products.

A feature model is equivalent to a propositional formula over features defined as the conjunction of the formulas obtained from the mapping on the right (adapted from [13]). As a result, deciding whether or not a product is valid according to the feature model reduces to a Boolean satisfiability problem, which implies that it can efficiently be computed with BDD or SAT solvers. However, in case of feature models displaying non-Boolean attributes and complex constraints, one needs to resort to SMT solvers, for example.

relationship		formula
root		$F_0 \iff \mathit{true}$
mandatory		$F_1 \iff F_2$
optional		$F_2 \implies F_1$
alternative		$(F_1 \iff (\neg F_2 \wedge \cdots \wedge \neg F_n \wedge F)) \wedge \cdots \wedge (F_n \iff (\neg F_1 \wedge \cdots \wedge \neg F_{n-1} \wedge F))$
or		$F \iff (F_1 \vee F_2 \vee \cdots \vee F_n)$
requires		$F_1 \implies F_2$
excludes		$\neg (F_1 \wedge F_2)$

As a running example, we use a family of coffee machines. This product line was used earlier too in work on the application of formal methods and tools such as `VMC` and `mCRL2` to (software) product lines, cf. e.g. [10, 5, 4, 7]. In short, coffee machines from our example product line are described as follows:

- A coffee machine either accepts one-euro coins (1 €), exclusively for European products, or one-dollar coins (1 $), exclusively for Canadian products.
- After inserting a coin, the user has to decide whether or not she wants sugar, by pressing one out of two buttons.
- Next, a beverage must be selected, which is either coffee (which is always available), tea or cappuccino (tea is optionally available, cappuccino is optionally available from European machines).
- After delivering a beverage, optionally a ringtone is rung. However, in case the product is offering cappuccino this must be the case.
- After the beverage is taken, the machine returns to its idle state.
- Optionally, coins of other denominations than one euro or one dollar can be inserted. Change will be returned when appropriate.

The attributed feature model depicted in Figure 1 organises 11 features, reflecting the description of the above product family. The root feature `M`, mandatory features `S`, `O`, `B`, and `C`, and optional features `E`, `D`, `R`, `P`, `T` and `X`. Sibling features `E` and `D` are alternatives, whereas independent features `D` and `P` are mutually exclusive. The feature `R` is required by feature `P`. The primitive features come equipped with an attribute for costs, an integer value between 3 and 10.

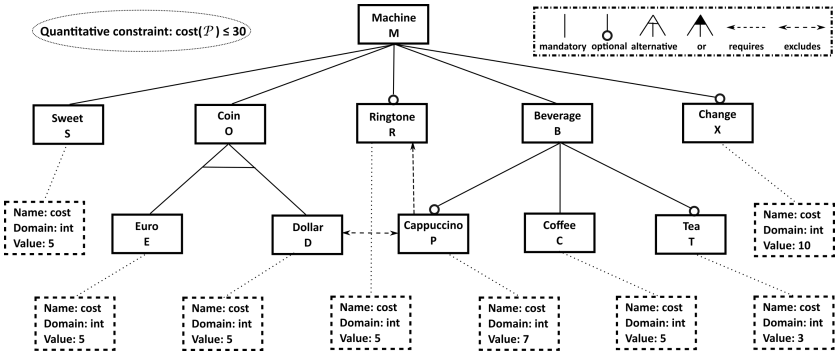


Fig. 1. Attributed feature model for the family of coffee machines [5].

More formally, the feature model yields 20 different products when ignoring the attribute constraints. Each product can be identified with a subset $\mathcal{P}$ of the set $\mathcal{F}$ of all features. For example, the subset of features $\{\mathtt{M}, \mathtt{S}, \mathtt{O}, \mathtt{E}, \mathtt{B}, \mathtt{C}\}$ describes a European coffee machine of a minimal number of features. The attribute function $\mathit{cost}\colon \mathcal{F} \to \mathbb{N}$ associated to the attribute `cost` extends to products in a straightforward manner, $\mathit{cost}(\mathcal{P}) = \sum \{\, \mathit{cost}(f) \mid f \in \mathcal{P} \,\}$, by assigning cost 0 to non-primitive features. If we consider the attributes and their overall quantitative constraint requiring $\mathit{cost}(\mathcal{P}) \leqslant 30$ for all $\mathcal{P} \subseteq \mathcal{F}$, then the attributed feature model only defines 16 valid products. For instance, the product $\{\mathtt{M}, \mathtt{S}, \mathtt{O}, \mathtt{E}, \mathtt{R}, \mathtt{B}, \mathtt{C}, \mathtt{T}, \mathtt{X}\}$ meets the feature requirements of the feature model, but has a cost of 33, exceeding the bound of 30.

3 Supervisory Control Synthesis

SCT provides a mechanism to obtain a model, an automaton, of a supervisory controller from given (component) models of the uncontrolled system and its requirements. The synthesised supervisory controller, if successfully produced, is such that the controlled system, which is the synchronous product of the uncontrolled system and the supervisory controller, satisfies the requirements and is additionally non-blocking, controllable and maximally permissive [38]. In the context of supervisory control, an automaton is called *non-blocking* in case from each state at least one of the so-called *marker states* can be reached. This indicates that the system always has the capability to return to an accepted rest state or stable state. The user has to indicate for each of the component models which are such marker locations.

In SCT, one distinguishes *controllable* and *uncontrollable* events. *Controllability* means that the supervisory controller is not permitted to block uncontrollable events from happening. The controller is only allowed to disable behaviour of the uncontrolled system indirectly by preventing controllable events from happening. Intuitively, controllable events correspond to stimulating or actuating the system, while the uncontrollable events correspond to messages provided by the

sensors (which may be neglected, but cannot be denied from existing). However, in the application of SCT demonstrated in this paper, all events are assumed to be controllable for simplicity. The resulting supervisory controller is *maximally permissive* (or least restrictive). This means that as much behaviour of the uncontrolled system as possible is still present in the controlled system without violating neither the requirements, nor the controllability nor the non-blocking condition on the reachability of marker states.

In earlier work, both the components and the requirements were expressed by means of finite automata. Thus the complete model of the system is a network or composition of automata. These automata may share certain events, and it is assumed that shared events will only occur at the system level if all automata that share that event execute it simultaneously. It is this form of *multi-party synchronisation* that allows a compact and modular specification [11, 42]. More recently, in order to increase modelling comfort, finite state machines were replaced by extended finite automata, which allow the use of variables in the automata [41]. In addition, the original algorithm for synthesis was strengthened to be able to deal with these as well [36]. Requirements for the controlled system to hold may be specified in various ways. First of all, allowed event sequences may be specified using automata. Also, state invariants and event conditions are typically used [35]. Invariants are predicates evaluating the overall state of the system. An event condition restricts the occurrence of an event to states that satisfy a specific state predicate.

As mentioned above, component models and (part of) the requirements are provided by means of extended finite automata. More specifically, a (component or requirement) automaton has a name. Refer to Listings 1 and 2 for examples of the concepts introduced here. Its name is used in other automata and requirements to refer to concepts that are defined inside the automaton, such as its events, variables and locations. In the automaton, local events may be declared (together with the indication that these are controllable). Similarly, local variables may be declared with their type and initial value (Listing 2). Furthermore, locations are declared together with the transitions emitting from them. Transitions are described using the keyword `edge`. A transition may have an event name, a condition or guard (following the keyword `when`), and an update or assignment (following the keyword `do`). The guard is a Boolean expression in terms of the values of variables and the current location of other automata. The update is an assignment of new values (by using an expression over variables and locations) to local variables. In `CIF 3`, a variable may only be assigned in the automaton it is declared in, but may be read/used in all other automata and requirements. `CIF 3` distinguishes algebraic variables, like `cost` in Listing 3, and discrete variables, like `cnt` in Listing 2. An algebraic variable is a variable for which the value is at all times defined as the result of an expression in the right-hand side of the declaration of that variable.

In Listing 1, the textual description of the component automaton `COFFEE` is given. It declares controllable events `done`, `coffee`, `cappuccino`, `pour_coffee`

and pour_milk. Other automata may refer to these events by prefixing the name of the defining automaton, e.g. COFFEE.cappuccino.

Listing 1. Automaton COFFEE

```
controllable
   done, coffee, cappuccino, pour_coffee, pour_milk;
location NoChoice: initial, marked;
   edge coffee goto Coffee;
   edge cappuccino goto Cappuccino;
location Coffee: marked;
   edge pour_coffee;
   edge done goto NoChoice;
   edge cappuccino goto Cappuccino;
location Cappuccino: marked;
   edge pour_coffee;
   edge pour_milk;
   edge done goto NoChoice;
   edge coffee goto Coffee;
```

The automaton of the COFFEE component model has three locations, of which the location NoChoice is the initial location. Note that all locations are marked. From the location NoChoice with the event coffee the automaton may transit to location Coffee. Note that in the automaton there are no variables and, therefore, no conditions and updates are specified for the transitions. If the description of a transition does not reveal a target location explicitly (using the keyword goto) then a loop is implied.

As another example, consider the requirement SWEETNESS specified in Listing 2. In this automaton a discrete variable with name cnt is introduced of type int[0..2], which means it can only take one of the values 0, 1, or 2. Initially, it has value 0. In this automaton, the use of conditions and updates is illustrated. For the transition labeled by event pour_sugar (from automaton SWEET, introduced later) it is required that the value of variable cnt is at most 1. Taking this transition results in adding 1 to the value of the variable by means of the assignment described after do. Observe that the order of transitions as described does *not* imply any priority among them.

Listing 2. Requirement SWEETNESS

```
disc int[0..2] cnt:=0;
location Idle: initial, marked;
   edge SWEET.sugar goto SugarNeeded;
   edge SWEET.done when SWEET.NoSugar;
location SugarNeeded: marked;
   edge SWEET.pour_sugar when cnt⩽1 do cnt:=cnt+1;
   edge SWEET.done when cnt=2 do cnt:=0 goto Idle;
```

Note how the requirement forces the sweet component to provide two portions of sugar when sugar is requested.

CIF 3 has ample features for defining templates with parameters and for reusing those. Please refer to http://cif.se.wtb.tue.nl for more information. We only make limited use of these mechanisms in this paper. CIF 3 has been applied to several industrial size case studies, cf. e.g. [22, 44].

4 Modelling Product Lines with CIF 3

In this section, we demonstrate several aspects involving the modelling of product lines with CIF 3. First, we consider the modelling of the set of acceptable products as defined by a feature model. Then we add to this model the uncontrolled behaviour of components, with the behaviour of the components as is. Furthermore, we show how behavioural requirements can easily be incorporated in the CIF 3 model, and we describe how these may be used to obtain a supervisory controller for the family of valid products, satisfying both the feature-related and the behavioural requirements.

4.1 Valid Products

In this section, we propose a simple way of obtaining all valid products from a feature model. In line with Section 2, we introduce Boolean variables for the presence and absence of features. We demonstrate how the restrictions imposed by a feature model can be described by invariants on these Boolean variables.

We introduce a generic definition FEATURE for features, shown in Listing 3. The definition may have multiple instances (cf. e.g. FM and FS representing an automaton for the features M and S, respectively). Here, the cost of each feature is taken as a so-called algebraic parameter. In this declaration an if-then-else expression is used to provide different values for the variable depending on a condition (present in this case). In CIF 3, every automaton needs to have at least one location, hence the dummy location (with name Dummy) defined in Listing 3.

Listing 3. Generic feature definition FEATURE

```
def FEATURE(alg int cost):
   alg int cost = if present : cost else 0 end;
   disc bool present in any;
   location Dummy: initial, marked;
end

FM: FEATURE(0); FS: FEATURE(5); ... ; FX: FEATURE(10);
```

We next discuss how in-tree, cross-tree and attribute constraints, as given by an attributed feature model, can be represented as CIF 3 requirements.

An example of a mandatory feature is the link between the beverage feature B and the coffee feature C in Figure 1. In CIF 3, we define a requirement that states that, invariantly, presence of the beverage feature B, represented by the Boolean variable present of automaton FB, coincides with presence of the coffee feature, i.e. the Boolean variable present of automaton FC (cf. the mapping in Section 2).

```
requirement invariant FB.present ⇔ FC.present;
```

An example of an optional feature is the connection between features B for beverage and T for tea in Figure 1. As an optional feature is allowed to be present when the parent feature is present, but it is not allowed to be present when the parent feature is absent, we have the following invariance requirement in CIF 3.

```
requirement invariant FT.present ⇒ FB.present;
```

Selection from alternative features occurs in Figure 1 concerning the features O, E and D. The intended meaning of the alternative features E and D is that presence of the parent feature O implies presence of exactly one of these alternative features, and the other way around: presence of E or D requires presence of O. The requirement generalises straightforwardly to more than two alternatives.

```
requirement invariant
   FO.present ⇔ ( FE.present ⇔ not FD.present );
```

An example of a 'requires' cross-tree constraint occurs in Figure 1 between the requiring cappuccino feature P and the required ringtone feature R. We define the following invariant as CIF 3 requirement.

```
requirement invariant FP.present ⇒ FR.present;
```

Figure 1 shows an 'excludes' cross-tree constraint between the dollar feature D and the cappuccino feature P. The meaning is that these features are not allowed to be both present in the same product, i.e. dollar machines do not offer cappuccino. Therefore, in CIF 3 we define the following requirement.

```
requirement invariant not (FD.present and FP.present);
```

The example feature model of Figure 1 includes one (global) attribute constraint. It states that the total cost of the selected features may not exceed the threshold of 30 units. We have modelled the cost associated with each feature as a parameter (cf. Listing 3). The total cost can therefore be modelled in CIF 3 as the sum of the costs of the features that are present. Note that in the generic feature definition the cost of a non-present feature is defined to be 0.

```
requirement invariant FM.cost + FS.cost + ··· + FT.cost ⩽ 30;
```

Apart from the five categories of constraints mentioned above, for feature models it is often assumed that a product contains at least one non-trivial feature, i.e. a feature which is not the root feature. The requirement below encodes a disjunction over all non-trivial features.

```
requirement invariant
   (FS.present or FO.present or ··· or FT.present);
```

Observe that the in-tree and cross-tree requirements arising from a feature model have been modelled in CIF 3 in such a way that the transformation from a feature model to a CIF 3 model can easily be automated. Translation of the attribute constraints requires some attribute constraint specific modelling; this can be automated too with a modest effort dependent on the expressiveness of the constraints allowed.

Synthesis with CIF 3 of a supervisory controller capturing the constraints of Figure 1 yields a product automaton of all features together with a supervisor which has 16 initial states, each corresponding to a valid product. This is the same number as reported in [5, 4].

4.2 Component Behaviour

The CIF 3 toolset is very much suited to describe the dynamic behaviour of components. With CIF 3 we initially define the *potential* behaviour of each individual component. It follows that the combined potential behaviour of the components together may contain undesired behaviour. In a later stage, we impose the additional behavioural requirements that are needed to obtain meaningful and acceptable behaviour (cf. Section 4.3).

For the coffee machine example we identified seven components in [4]: COIN, CANCEL, SWEET, RINGTONE, COFFEE, TEA and MACHINE. We will specify the potential behaviour of each of these in isolation by means of CIF 3 automata. In principle this needs to be described textually, but for presentation purposes we provide it in a graphical way as illustrated in Figure 2. The textual description of automaton COFFEE has been given in Listing 1 in Section 3.

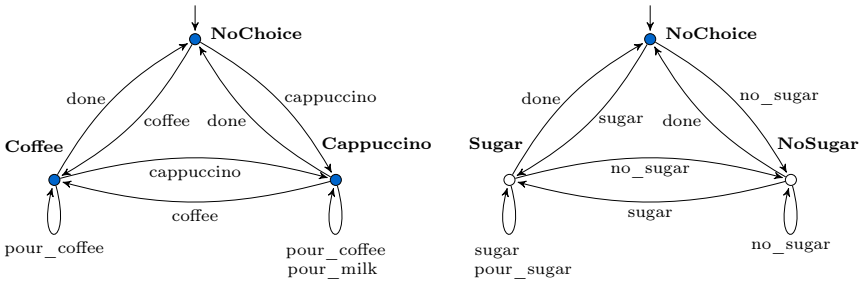


Fig. 2. Graphical representation of CIF 3 automata for components COFFEE and SWEET. Initial states indicated by an incoming arrow, marker states indicated by a filled state.

Using the CIF 3 toolset one can easily obtain the uncontrolled behaviour of the seven components together, an automaton with 18 states and 207 transitions. It contains all possible behaviour for the case in which all components are included (which may very well be prohibited by the feature constraints when imposed). More interestingly, based on the attributed feature model defined in CIF 3 and the component automata we can synthesise a state space that contains 16 initial states, one for each valid product, and display the behaviour for each of them. The state space has 147 states and 1254 transitions. It is obtained in less than 2 seconds of user time on a standard laptop. In the composed system at hand we

have included the requirement PRESENCE_CHECK below, which couples features and components, i.e. it states for each event of the components which features need to be present for it to be available. This is similar to the notion of an FTS [17] mentioned in the introduction, where transitions are not labelled with action names only but by a Boolean expression of features as well. Note that here we choose to give global conditions for events. However, if we instead require these conditions locally in the component automata, different occurrences of the same event can be made subject to different conditions (as in FTSs).

Listing 4. Requirement PRESENCE_CHECK

```
location Dummy: initial, marked;
   edge CANCEL.cancel when FX.present;
   edge COFFEE.coffee when FB.present and FC.present;
   edge TEA.tea when FB.present and FT.present;
      ...
```

With the model presented so far we have achieved to describe the behaviour of all 16 valid products of the running example, as specified by the feature model and the component models. Note that so far the products are not supervised yet, we have only achieved to enforce behaviour that is consistent with the feature model. This means that we may still be allowing unacceptable behaviour, such as pouring coffee while no coin has been inserted. In the next subsection, we describe how to model further requirements that the controller should enforce.

4.3 Behavioural Requirements

As mentioned, the constructed product behaviours are still uncontrolled, in the sense that we have not yet attempted to implement a supervisory controller that forces the products to behave according to a list of desired behavioural requirements. To illustrate the flexibility of the approach, we will introduce some of these. We consider the following types of requirements: (i) state invariants, (ii) event conditions, (iii) event ordering requirements, (iv) requirements using observers and (v) requirements using additional variables.

State invariants In many applications of supervisory controller synthesis one uses so-called state invariants to express that certain combinations of states of components should not occur at the same time. In the case study of this paper, for instance, one may desire to require that it is impossible to be ready for pouring coffee and tea at the same time. In CIF 3 this can be expressed as follows.

```
requirement invariant not (COFFEE.Coffee and TEA.Tea);
```

Event conditions An event condition is a requirement in which a state predicate must be satisfied before an event may be executed. In CIF 3, the following notation is used for such event conditions: `requirement <event> needs <pred>`, where `<event>` is an event name and `<pred>` is a state predicate. The meaning

of this requirement is that the event may only be executed in case the predicate holds. For instance, we may want to impose that it is not allowed to change the choice of a beverage (coffee, cappuccino or tea) once the choice has been made. This means that a choice may only be made if no choice has been made yet. Observe that reference is made to events and locations of component automata in such requirements.

Listing 5. Example event conditions

```
requirement
   COFFEE.coffee needs COFFEE.NoChoice and TEA.NoChoice;
requirement
   COFFEE.cappuccino needs COFFEE.NoChoice and TEA.NoChoice;
requirement
   TEA.tea needs COFFEE.NoChoice and TEA.NoChoice;
```

Event ordering requirements Another type of requirement is used to express that certain events may only occur in specific orderings. For instance, one may have the requirement that a ringtone may only occur after a drink has been delivered. We can use automata to model such requirements, as in Listing 6.

Listing 6. Requirement `RING_AFTER_BEVERAGE_COMPLETION`

```
location NotCompleted: initial, marked;
   edge COFFEE.done when FR.present goto Completed;
   edge TEA.done when FR.present goto Completed;
   edge COFFEE.done, TEA.done when not FR.present;
location Completed:
   edge RINGTONE.ring goto NotCompleted;
```

Requirements using observers Many of the events that can be performed by the components of the coffee machine should only occur if a coin is in the machine. For such a requirement, we define an additional automaton, commonly called an observer, which establishes whether or not a coin is in the machine. It is depicted in Listing 7. It uses the events from the component automata to observe their occurrences and then uses these to decide on the logical state of the system. Note that we have taken care to develop this observer in such a way that all its events are possible from any of its states. Thus, the automaton itself does not restrict the behaviour of the components.

Listing 7. Observer automaton `COIN_PRESENCE`

```
location NoCoinPresent: initial, marked;
   edge COIN.insert goto CoinPresent;
   edge CANCEL.cancel, Machine.take_cup;
location CoinPresent:
   edge CANCEL.cancel goto NoCoinPresent;
   edge MACHINE.take_cup goto NoCoinPresent;
   edge Coin.insert;
```

Next, this observer automaton may be used in requirements. For example, coffee may only be poured if a coin is in the system.

```
requirement COFFEE.coffee needs COIN_PRESENCE.CoinPresent;
```

Using additional variables An exemplary quantitative requirement to restrict the behaviour of products in such a way that if sugar is chosen, then always two portions are used, was provided in Listing 2 in Section 3.

4.4 Synthesis

Above we have illustrated how various types of requirements regarding the behaviour of the components may be modelled in CIF 3. With this in place, we can obtain a supervisor for each of the valid products by simply combining the feature model, the component behaviour models and all of the requirements into one model. The synthesis algorithm then constructs the synchronous product of the component and requirement automata and starts an iterative process of removing states that do not satisfy the invariants and the nonblocking property until a proper supervisor is obtained, or an empty supervisor results indicating that no supervisor may exist at all. The synthesis algorithm suffers from the same state space explosion problem as for model checking [23].

In this case, application of the supervisory controller synthesis options offered by CIF 3 then results in a single CIF 3 model that represents an automaton for each of the valid products. The state space of the 16 valid products together contains 503 states an 868 transitions. Again, it was obtained within 2 seconds user time on a standard laptop.

Among others, the resulting CIF 3 model describes for each event the additional conditions that need to be satisfied in terms of the locations and values of variables for the presence of features, component, observer and requirement automata. Listing 8 provides part of this. Note that in these conditions several automata, such as X, Y, Z, RESTRICTED_CANCEL and NO_FREE_LUNCH are referenced that have not been shown in this paper.

Listing 8. Supervisor automaton

```
edge COFFEE.cappuccino when not FD.present and (FP.present
    and X.Idle) and (TEA.NoChoice and (RESTRICTED_CANCEL.
    CancelAllowed and NO_FREE_LUNCH.Full));

edge COFFEE.pour_milk when not X.OneUnitNeeded and
    Z.OneUnitNeeded or (X.OneUnitNeeded and (Z.OneUnitNeeded
    and SWEET.NoSugar) or X.OneUnitNeeded and (Z.
    OneUnitNeeded and SWEET.Sugar));

edge SWEET.pour_sugar when X.Idle and FT.present and
    (TEA.Tea and SWEETNESS.cnt≠2) or (X.OneUnitPoured and
    SWEETNESS.cnt≠2 or X.OneUnitNeeded and SWEETNESS.cnt≠2);
```

```
edge TEA.pour_tea when Y.OneUnitNeeded and
    (not Y.OneUnitNeeded or not SWEET.NoChoice);

edge TEA.tea when X.Idle and FT.present and
    (RESTRICTED_CANCEL.CancelAllowed and NO_FREE_LUNCH.Full);
...
```

It must be mentioned that these conditions are not simplified in any way. For example the condition of event `pour_tea` may be simplified to `Y.OneUnitNeeded and not SWEET.NoChoice`.

5 Concluding remarks

We have shown how CIF 3 can be put to work for feature-guided integration of components. Given (i) an attributed feature model capturing in-tree, cross-tree and attribute constraints, (ii) a description of the potential behaviour of a number of components and (iii) additional static, dynamic and quantitative requirements, the CIF 3 machinery synthesises a composition of the components that is consistent with the attributed feature model and adheres to the additional requirements, if possible at all. Otherwise CIF 3 reports that a composition is non-existent. Each initial state of the overall system corresponds to a unique valid product from the product line as defined by the feature model. All products are sound *by construction* and the set of products is complete with respect to the combined feature model and behavioural requirements, because of the maximal permissiveness guaranteed by CIF 3 [38, 36].

Since we focussed on supervisory controller synthesis for product lines as supported by CIF 3, many notions from SCT that are useful to SPLE have been left unspoken here. For instance, one can imagine a coffee machine to come equipped with a sensor to monitor whenever one of the ingredients, say sugar, has become depleted. By its nature, the sensor's messaging that the machine is out of sugar is an *uncontrollable* event. From the perspective of model-based engineering such a distinction is relevant; the modelling formalism of CIF 3 is sufficiently expressive to take this into account. Incorporation of uncontrollable events may be useful to detect feature interaction as failure of synthesis may reveal unexpected dependencies. For this to work the synthesis algorithm needs to be refined.

In the present case study, it is assumed that presence/absence of features is statically organised. However, many systems, including the coffee machine, may be reconfigured while operating. For instance, one may wish to remove or add the tea feature dynamically (possibly dependent on the ingredients that are available). This can easily be achieved by adding a self-loop transition in the FT automaton labelled with a reconfigure event which switches presence of the feature. More complicatedly, a reconfiguration modifying a specific feature may immediately result in a violation of some of the constraints imposed by the feature model because of the interplay of the feature with other features. It is possible to also model this type of reconfigurations in CIF 3 by introducing events

that represent the simultaneous reconfiguration of several features, temporarily lifting the constraints stemming from the feature model.

Often, especially in earlier design phases, the set of requirements that are to be enforced has not become clear yet. As a consequence, supervisory control synthesis may result in the impossibility to produce a supervisor, or lead to a controlled system that omits states of which the designer would not expect their omission. In recent work [43], the synthesis algorithm has been adapted to retrieve the reason why the controlled system is blocked from reaching specific states. This information can then be used for understanding which requirements (under which conditions) are conflicting, whereupon the model of the components and/or the requirement may be refined. For future work, we plan to consider adaptations to the synthesis algorithm that may reveal conflicts among features and/or their attributes. Such conflicts may be used to discover that changes in the components interfere with the feature model in force, excluding products that were valid before.

Acknowledgments Ter Beek is supported by EU project QUANTICOL, 600708. We are thankful to the ISOLA reviewers for their constructive comments.

References

1. P. Asirelli, M. H. ter Beek, A. Fantechi, and S. Gnesi. A Compositional Framework to Derive Product Line Behavioural Descriptions. In T. Margaria and B. Steffen, editors, *ISoLA'12*, volume 7609 of *LNCS*, pages 146–161. Springer, 2012.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. K. Bąk, Z. Diskin, M. Antkiewicz, K. Czarnecki, and A. Wąsowski. Clafer: unifying class and feature modeling. *Software & Systems Modeling*, 2015.
4. M. H. ter Beek and E. P. de Vink. Towards modular verification of software product lines with mCRL2. In T. Margaria and B. Steffen, editors, *ISoLA'14*, volume 8802 of *LNCS*, pages 368–385. Springer, 2014.
5. 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.
6. M. H. ter Beek, E. P. de Vink, and T. A. C. Willemse. Towards a Feature mu-Calculus Targeting SPL Verification. In *FMSPLE'16*, volume 206 of *EPTCS*, pages 61–75, 2016.
7. M. H. ter Beek, A. Fantechi, S. Gnesi, and F. Mazzanti. Modelling and analysing variability in product families: Model checking of modal transition systems with variability constraints. *Journal of Logical and Algebraic Methods in Programming*, 85(2):287–315, 2016.
8. M. H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. Statistical Analysis of Probabilistic Models of Software Product Lines with Quantitative Constraints. In *SPLC'15*, pages 11–15. ACM, 2015.
9. M. H. ter Beek, A. Legay, A. Lluch Lafuente, and A. Vandin. Statistical Model Checking for Product Lines. In T. Margaria and B. Steffen, editors, *ISoLA'16*, LNCS. Springer, 2016.
10. 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.

11. D. A. van Beek, W. J. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. M. van de Mortel-Fronczak, and M. A. Reniers. CIF 3: Model-Based Engineering of Supervisory Controllers. In E. Ábrahám and K. Havelund, editors, *TACAS'14*, volume 8413 of *LNCS*, pages 575–580. Springer, 2014.
12. G. Behrmann, A. David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi, and M. Hendriks. UPPAAL 4.0. In *QEST'06*, pages 125–126. IEEE, 2006.
13. D. Benavides, S. Segura, and A. Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: a Literature Review. *Information Systems*, 35(6), 2010.
14. R. Bubel, R. Hähnle, and M. Pelevina. Fully abstract operation contracts. In T. Margaria and B. Steffen, editors, *ISoLA'14*, volume 8803 of *LNCS*, pages 120–134. Springer, 2014.
15. 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.
16. A. Classen, M. Cordy, P. Heymans, A. Legay, and P.-Y. Schobbens. Formal semantics, modular specification, and symbolic verification of product-line behaviour. *Science of Computer Programming*, 80:416–439, 2014.
17. 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.
18. A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking Lots of systems: Efficient verification of temporal properties in software product lines. In *ICSE'10*, pages 335–344. ACM, 2010.
19. 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.
20. B. Delaware, W. R. Cook, and D. S. Batory. Product Lines of Theorems. In C. V. Lopes and K. Fisher, editors, *OOPSLA'11*, pages 595–608. ACM, 2011.
21. E. W. Dijkstra. A constructive approach to the problem of program correctness. *BIT Numerical Mathematics*, 8(3):174–186, 1968.
22. S. T. J. Forschelen, J. M. van de Mortel-Fronczak, R. Su, and J. E. Rooda. Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems*, 22(4):511–540, 2012.
23. P. Gohari and W. M. Wonham. On the complexity of supervisory control design in the RW framework. *IEEE Trans. Systems, Man, and Cybernetics, Part B*, 30(5):643–652, 2000.
24. A. Gruler, M. Leucker, and K. D. Scheidemann. Modeling and model checking software product lines. In G. Barthe and F. S. de Boer, editors, *FMOODS'08*, volume 5051 of *LNCS*, pages 113–131. Springer, 2008.
25. R. Hähnle and I. Schaefer. A Liskov Principle for Delta-Oriented Programming. In T. Margaria and B. Steffen, editors, *ISoLA'12*, volume 7609 of *LNCS*, pages 32–46. Springer, 2012.
26. A. Hall. Correctness by Construction: Integrating Formality into a Commercial Development Process. In L. Eriksson and P. A. Lindsay, editors, *FME'02*, volume 2391 of *LNCS*, pages 224–233. Springer, 2002.
27. A. Hall and R. Chapman. Correctness by Construction: Developing a Commercial Secure System. *IEEE Software*, 19(1):18–25, Jan/Feb 2002.
28. C. A. R. Hoare. Proof of a Program: FIND. *Communications of the ACM*, 14(1):39–45, 1971.

29. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2006.
30. P. Juodisius, A. Sarkar, R. R. Mukkamala, M. Antkiewicz, K. Czarnecki, and A. Wąsowski. Clafer: Lightweight Modeling of Structure and Behavior with Variability. Unpublished manuscript.
31. K. Lauenroth, K. Pohl, and S. Töhning. Model checking of domain artifacts in product line engineering. In *ASE'09*, pages 269–280. IEEE, 2009.
32. 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.
33. M. Lochau, S. Mennicke, H. Baller, and L. Ribbeck. DeltaCCS: A Core Calculus for Behavioral Change. In T. Margaria and B. Steffen, editors, *ISoLA'14*, volume 8802 of *LNCS*, pages 320–335. Springer, 2014.
34. M. Mannion and J. Cámara. Theorem Proving for Product Line Model Verification. In F. van der Linden, editor, *PFE'03*, volume 3014 of *LNCS*, pages 211–224. Springer, 2003.
35. J. Markovski, K. G. M. Jacobs, D. A. van Beek, L. J. A. M. Somers, and J. E. Rooda. Coordination of resources using generalized state-based requirements. In J. Raisch, A. Giua, S. Lafortune, and T. Moor, editors, *WODES'10*, pages 287–292. International Federation of Automatic Control, 2010.
36. L. Ouedraogo, R. Kumar, R. Malik, and K. Åkesson. Nonblocking and Safe Control of Discrete-Event Systems Modeled as Extended Finite Automata. *IEEE Transactions on Automation Science and Engineering*, 8(3):560–569, 2011.
37. K. Pohl, G. Böckle, and F. J. van der Linden. *Software Product Line Engineering: Foundations, Principles, and Techniques*. Springer, 2005.
38. P. J. Ramadge and W. M. Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.
39. J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. MIT Press, 2001.
40. I. Schaefer. Variability Modelling for Model-Driven Development of Software Product Lines. In D. Benavides, D. S. Batory, and P. Grünbacher, editors, *VaMoS'10*, volume 37 of *ICB-Research Report*, pages 85–92. Universität Duisburg-Essen, 2010.
41. M. Skoldstam, K. Åkesson, and M. Fabian. Modeling of discrete event systems using finite automata with variables. In *CDC'07*, pages 3387–3392, 2007.
42. B. van der Sanden, M. A. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, and R. R. H. Schiffelers. Modular model-based supervisory controller design for wafer logistics in lithography machines. In *MoDELS'15*, pages 416–425. IEEE, 2015.
43. L. Swartjes, M. A. Reniers, D. van Beek, and W. Fokkink. Why Is My Supervisor Empty? Finding Causes for the Unreachability of States in Synthesized Supervisors. In *WODES'16*. IEEE, 2016. To appear.
44. R. J. M. Theunissen, D. A. van Beek, and J. E. Rooda. Improving evolvability of a patient communication control system using state-based supervisory control synthesis. *Advanced Engineering Informatics*, 26(3):502–515, 2012.
45. T. Thüm, I. Schaefer, M. Hentschel, and S. Apel. Family-Based Deductive Verification of Software Product Lines. In *GPCE'12*, pages 11–20. ACM, 2012.