Supervisory Control for Dynamic Feature Configuration in Product Lines

Supervisory Control for Dynamic Feature Configuration in Product Lines

Michel Reniers and Sander Thuijsman
Department of Mechanical Engineering, Eindhoven University of Technology, Eindhoven, The Netherlands
{m.a.reniers, s.b.thuijsman}@tue.nl

Abstract— **In this paper a method for engineering supervisory controllers for product lines with dynamic feature configuration is proposed. The variability in valid configurations is described by a feature model. Behavior of system components is achieved using (extended) finite automata and both behavioral and dynamic configuration constraints are expressed by means of requirements as is common in supervisory control theory. Supervisory control synthesis is applied to compute a behavioral model in which the requirements are adhered to. For the challenges that arise in this setting, multiple solutions are discussed. Some of these solutions are exemplified in the CIF tool set using a wiper system model.**

I. Introduction

In present day development of systems and products, reuse of both software and hardware components is sought to reduce development and production costs, and shorten time-to-market. The goal of Software/System Product Line Engineering (SPLE) is to facilitate reuse throughout all phases of systems engineering [1]. Adoption of this paradigm requires identification of the core assets of the products in the domain in order to exploit their commonality and manage their variability, often defined in terms of features. A feature is defined as a logical unit of behavior specified by a set of functional and non-functional requirements [2] or a distinguishable characteristic of a concept (system, component, etc.) that is relevant to some stakeholder [3]. Feature models may be used to define which combinations of features are considered valid product configurations [4].

In literature there has been much attention for correct configuration of SPLs [4]. Since [5], behavioral correctness is studied. Typically the approaches that are used for guaranteeing a proper functioning (i.e., correct with respect to its requirements or specifications) SPL are verification technologies such as theorem provers [6], model checkers [7], and correct-by-construction approaches such as supervisory control synthesis [8]. In [8], for the first time supervisory control synthesis has been considered for constructing supervisory controllers for an SPL described by a feature model.

In supervisory control synthesis [9], starting from a model of the uncontrolled system and a model of the behavioral requirements, a model of a supervisory controller is synthesized. Typically, the models that are input to supervisory control synthesis are discrete-event systems models such as (extended) finite automata [10], [11].

Research leading to these results has received funding from the EU ECSEL Joint Undertaking under grant agreement n^{o} 826452 (project Arrowhead Tools) and from the partners national programs/funding authorities.
978-1-7281-8928-4/20/$31.00 ©2020 IEEE

There are two tool suites that support supervisory control synthesis for models expressed as extended finite automata: Supremica [12] and CIF [13]. In [8], it has been shown how the CIF language and tool set can be used for synthesizing a supervisory controller that is suited for an SPL. The approach uses the concept of algebraic variables extensively, which is not available in Supremica.

In [8], the treatment was restricted to the setting where the configuration of the system is static, i.e., is assumed not to change during system behavior. In this paper, this work is extended to also consider the dynamic configuration of the system in terms of the presence of features. A number of challenges arise when dealing with dynamic configuration:

- How to model presence and absence of features?
- How to model the uncontrolled system in such a way that it properly takes the current configuration into account?
- How to model behavioral requirements depending on presence of features?
- How to deal with transitional behavior during dynamic configuration?

For most of these challenges there are multiple solutions and it depends on the case at hand which one is most appropriate. When discussing these challenges we will mention the alternatives and illustrate some of them.

The contribution of this paper is a model-based approach for engineering supervisory controllers for SPLs with dynamic feature configuration. The main ingredients are:

- Capturing dynamic configuration of features in models
- Modeling the behavior of the components comprising the systems
- Linking event availability to presence of features in a configuration
- Modeling of behavioral and dynamic configuration constraints taking into account the configuration
- Synthesizing a correct-by-construction supervisory controller from the developed models

To exemplify the method discussed in this paper, we use the relatively small wiper system from [14] as a running example. In [15], the much larger body comfort system from [16] is successfully used as a case study. This body comfort system is the largest case study (in terms of number of features and number of valid configurations) from literature [8], [14], [16], [17] as far as we know. Although supervisory controller synthesis is known to suffer from scalability problems [10], synthesis for this larger case study was completed within seconds.

A. Related work

In [18] an approach for dynamic software reconfiguration in sensor networks is presented. The dynamic reconfiguration is based on formal constraints in terms of quality-of-service parameters that are measured at runtime.

Dynamic runtime variability of software product lines in embedded automotive software systems is applied to create adaptable and reconfigurable software architectures in [19]. Also [20] discusses reconfiguration with the purpose of determining an optimal configuration at runtime. In both papers the dynamic configuration is under control, which is typically not the case in the present paper.

In [21], a feature-oriented method is proposed to support runtime variability reconfiguration by introducing an intermediate level between feature variations and implementations.

The authors of [22] argue that it is not reasonable to anticipate all relevant context changes during design-time and therefore propose a model that combines learning of adaptation rules with evolution of the configuration space. In the approach of the present paper, it is assumed that the available features are known at design-time.

In [23], the authors deal with reconfiguration of real-time embedded systems to cope with hardware/software faults.

In [24], priced featured automata were translated to extended finite automata and the structure of the SPL was used to greatly reduce the number of controller syntheses required to solve game-based energy problems.

[25] apply supervisor synthesis to featured modal contract automata. They synthesize orchestrations, that match service requests to service offers, for all valid products in a product line, by joining the orchestrations of a small subset of the valid products. By means of a composition operation, the product line can dynamically be updated and new services can join composite services.

In none of the mentioned related work, dynamic feature configuration in relation to supervisory control engineering with a clear separation of uncontrolled system behavior and specification of behavioral and dynamic reconfiguration requirements has been discussed.

B. Structure

In Section II we introduce (static) feature models and CIF. In Section III, the modeling of dynamically configured feature models is discussed. In Section IV, modeling of behavior in the setting of dynamic configuration is discussed. Section V concludes the paper.

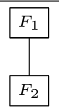
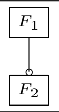
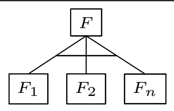
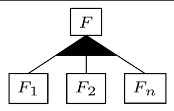
II. Preliminaries

A. Feature Models

A feature model [26] is a graph with a collection of nodes representing features, and a number of relations between these features, called feature constraints. The feature constraints that can be expressed are summarized in Table I, which is taken from [8]. The right column provides a logical formula that expresses how presence of the features (denoted by the F_i) is restricted by the different types of constraints.

For any valid configuration a *root* feature needs to be present. *Mandatory* features are required to be present when their parents are, and optional features may be present when their parents are. For a set of *alternative* features, exactly one is present. And for a set of *or* features, at least one is present. It can also be defined that the presence of a certain feature requires or excludes another feature to be present.

TABLE I
DIFFERENT FEATURE CONSTRAINTS OF A FEATURE MODEL.

Constraint		Formula
root	$\boxed{F_0}$	$F_0 \iff true$
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	$\boxed{F_1} \dashrightarrow \boxed{F_2}$	$F_1 \implies F_2$
excludes	$\boxed{F_1} \dashleftarrow\dashrightarrow \boxed{F_2}$	$\neg(F_1 \wedge F_2)$

Example: Feature model for wiper system

Consider a product line for a wiper system [14], that wipes the front window of a car. The wiper system essentially consists of a sensor and a wiper. Both are available in a low and a high quality version. The wiper system is optionally equipped with a permanent wiping feature. A feature model that captures the allowed configurations is presented in Fig. 1. An example of a valid configuration is where the following features are present: a root (Wiper System), sensor, wiper, low quality sensor, and high quality wiper feature.

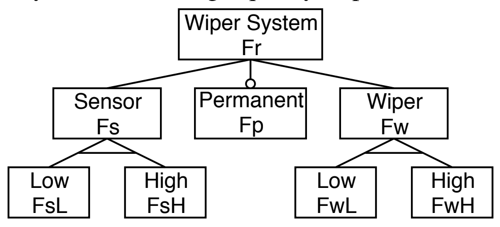


Fig. 1. Feature model for the wiper system [14].

Besides the shown feature constraints, also attribute constraints [27] can be expressed using attributes of features. An example of such an attribute could be the weight (or price) of a feature. A constraint could be a maximal value for the total weight (price) of the system. In the wiper system there are no such attribute constraints. In [8] supervisor synthesis on a feature model with attribute constraints of a coffee machine without dynamic reconfiguration is considered. Our methodology for dynamic reconfiguration would straightforwardly extend to such feature models with attribute constraints.

B. CIF

CIF is a language and tool set that supports model-based engineering of supervisory controllers involving modeling, (visualized) simulation, synthesis, verification, and code generation [13]. In the past years CIF has been applied to many industrial-size case studies such as lithography machines [28], health-care systems [29], automotive applications [30], and infrastructural systems [31]. Although CIF allows modeling of real-valued variables that evolve continuously over time (as described by differential equations), for the purpose of this paper attention is restricted to discrete-event models.

Discrete-event models of the uncontrolled system (also called plants) can be developed in the form of a collection of *extended finite automata* [11]. The automata that comprise the plant synchronise over shared events [10] and interact through the reading of each other's (discrete) variables. An automaton consists of locations and edges between these locations. The edges are labeled by an event, a guard and an update. The guard describes a condition (in terms of the variables) that enables the occurrence of the event associated with the edge. The update describes how the values of the variables change in such a transition. In CIF variables are declared inside an automaton and follow the 'global read, local write' principle, which means that each variable may be inspected in any of the automata, but may only be adapted in its defining automaton. A CIF automaton has at least one initial location, and variables have at least one initial value.

Events are defined to be *controllable* or *uncontrollable*. Uncontrollable events cannot be prevented from occurring by a supervisory controller, whereas controllable events can be blocked. The extended finite automata may have *marked* states. Marked states are states in which the system has finished a task. By applying supervisor synthesis, the controllable events are restricted in such a way that from each reachable state, a marked state can eventually be reached.

An example of a CIF automaton is given in Listing 1. Its graphical representation is given in Fig. 2. Locations are represented by small circles with their name next to them. Initial states have a dangling incoming arrow (and possibly an expression stating the initial values of the variables). Marked states have a double circle representation. Edges that are labeled by a controllable event are represented by a solid arrow and edges with an uncontrollable event by a dashed arrow. The optional guard is indicated by the keyword `when` and the update using `do`. In this paper we use both textual and graphical representations as we see fit.

In CIF requirements are specified by means of automata that state in which orderings the contained events are allowed

Listing 1. Textual CIF model of an automaton.

```
plant automaton ExampleAutomaton:
controllable start, process;
uncontrollable finish;
disc int c = 0;
  location Idle: initial; marked;
    edge start goto Busy;
  location Busy:
    edge process do c:=c+1;
    edge finish when c>4 do c:=0 goto Idle;
end
```

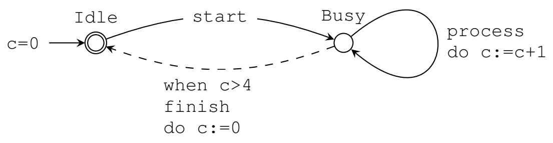


Fig. 2. Graphical representation of the automaton from Listing 1.

to occur or by using state-based expressions such as event conditions and state invariants [32], [33]. An event condition restricts the occurrences of an event to situations where a certain condition in terms of the variables of the model is satisfied. A state invariant expresses in which states the system is allowed to be.

CIF has several concepts that facilitate modeling of large systems, such as a definition/instantiation mechanism for automata and requirements and algebraic variables. An algebraic variable is a global variable (not associated with a single automaton) the value of which is defined to be identical to the value of some expression (in terms of other variables). In this paper algebraic variables are used abundantly.

With each location L in each automaton A, CIF associates a location variable $A.L$ that may be used in guards, in right-hand sides of updates and algebraic variables and in state-based requirements. Updates of these variables are implicit and according to the location change of an automaton.

C. Static feature models in CIF

Before we turn our attention to modeling feature models that can configure dynamically, in this section we demonstrate modeling of feature models that do not configure dynamically, i.e., they are static, as proposed in [8]. In the next section we will show how these static models can be extended to feature models supporting dynamic feature configuration.

A CIF model representing all allowed configurations for a given feature model is obtained as follows. An automaton is introduced for each feature that captures whether the feature is present or not. It uses a Boolean variable `present`, the value of which is fixed initially, that is `true` when the feature is present and `false` otherwise. This variable is also used to capture the relations expressed in the feature model.

Since all feature automata have the same structure we use an automaton definition in CIF, which is then instantiated for each feature in the feature model. The CIF specification for this automaton definition is given in the first four lines of Listing 2. In CIF, every automaton needs to have at least one location, hence the dummy location (without a name) defined in Listing 2. Note that the initial value of `present` is left implicit (is allowed to be either `true` or `false` by using the keywords `in any`). For each feature in the feature model an instance of this feature automaton definition is obtained by a statement such as the ones in Lines 6 and 7.

Listing 2. Automaton definition for features and instantiation for features.

```
1 plant def FEATURE():
2   disc bool present in any;
3   location: initial; marked;
4 end
5
6 F1: FEATURE();
7 F2: FEATURE();
8 ...
```

Feature constraints arising from a feature model can be modeled in CIF in such a way that the transformation from a feature model to a CIF model can easily be automated. For each of the constraint types in Table I, an algebraic CIF expression is shown in Listing 3. '`//`' in the listing denotes that the remainder of the line is a comment, which we use to specify the constraint type. The notation also allows to express more complex constraints between features that are not considered here.

Listing 3. Several feature constraint expressions.

```
alg bool r1 = F0.present <=> true; //root
alg bool r2 = F1.present <=> F2.present; //mandatory
alg bool r3 = F2.present => F1.present; //optional
alg bool r4 = (F1.present <=> (not(F2.present) and F.present))
       and (F2.present <=> (not(F1.present) and F.present));
       //alternative
alg bool r5 = F.present <=> (F1.present or F2.present); //or
alg bool r6 = F1.present => F2.present; //requires
alg bool r7 = not (F1.present and F2.present); //excludes
```

A valid configuration is obtained if and only if all feature constraints are satisfied. To this end, the algebraic expressions for the seperate feature constraints, such as in Listing 3, can be used. In Listing 4 we introduce an algebraic expression `sys_valid` that evaluates to true if all feature constraints are satisfied. We also define automaton `Validity` in Lines 3-5 in Listing 4. At the moment this automaton only states that the system initially is in a valid system configuration.

Listing 4. Validity of configuration.

```
1 alg bool sys_valid = r1 and r2 and r3 and ...;
2
3 plant automaton Validity:
4   location: initial sys_valid; marked;
5 end
```

Example: Static feature model for wiper system in CIF

The CIF specification of the feature model for the wiper system is given in Listing 5. Construction of the state space of this model in CIF results in a structure with 8 allowed configurations each represented by an initial state (and nothing more as we have not yet modeled any behavior).

Listing 5. Feature instances of the wiper system.

```
plant def FEATURE():
  disc bool present in any;
  location: initial; marked;
end

Fr:  FEATURE();
Fs:  FEATURE();
Fp:  FEATURE();
Fw:  FEATURE();
FsL: FEATURE();
FsH: FEATURE();
FwL: FEATURE();
FwH: FEATURE();

alg bool r1 = Fr.present;
alg bool r2 = Fr.present <=> Fs.present;
alg bool r3 = Fp.present => Fr.present;
alg bool r4 = Fr.present <=> Fw.present;
alg bool r5 = Fr.present <=> (Fs.present and Fw.present);
alg bool r6 = (FsL.present <=> (not(FsH.present) and Fs.present))
        and (FsH.present <=> (not(FsL.present) and Fs.present));
alg bool r7 = (FwL.present <=> (not(FwH.present) and Fw.present))
        and (FwH.present <=> (not(FwL.present) and Fw.present));

alg bool sys_valid= r1 and r2 and r3 and r4 and r5 and r6 and r7;

plant automaton Validity:
  location: initial sys_valid; marked;
end
```

III. Dynamic configuration

In the setting discussed in the previous section, the configuration is decided upon initialization of the system and can not change at any later stage. In this section we consider the situation that features may configure dynamically.

Different types of reconfiguration can be imagined. For example, it can be decided if reconfigurations take place in isolation, or may occur simultaneously. Both alternatives can be modeled in CIF, albeit with different adaptations of the feature definition. In this paper we assume single feature reconfiguration, which is the subject of Section III-A.

If one allows models to dynamically configure, there may be situations where a specific change in configuration would result in a violation of the feature constraints. An example is removing the high quality wiping feature from the wiper system which would result in a configuration where neither the high quality nor the low quality wiper is present. Hence, it must be decided if such violations of the feature constraints are allowed, this is discussed in Section III-B.

A. Single feature reconfiguration

In Section II-C, for each feature an automaton with a variable named `present` is introduced that captures whether the feature is present. To allow change of presence status of a feature, the value of the corresponding `present` variable needs to be able to change. This can be modeled with a relatively small adaptation to the current feature definition. For each feature a `come` and `go` event are introduced that represent the addition and removal of the feature from the configuration. The resulting feature definition is shown in Listing 6. The events `come` and `go` are defined inside the automaton. As a consequence, there is an instance of both events for each instance of the plant definition. These events are chosen to be uncontrollable since they occur outside the influence of the supervisory controller (to be designed).

Listing 6. Automaton definition for features with reconfiguration.

```
plant def FEATURE():
uncontrollable come, go;
disc bool present in any;
  location: initial; marked;
    edge come when not present do present:=true;
    edge go when present do present:=false;
end
```

We have now obtained a state space that contains each possible reconfiguration, also those that are invalid by the feature model. Restricting reconfigurations to valid configurations can be achieved by adding a plant invariant such as presented in Listing 7. Adding this invariant removes all states where sys_valid evaluates to `false`, and all transitions toward these states in the plant's behavior.[1]

Listing 7. Invariant restricting reconfiguration to valid configurations.

```
plant invariant sys_valid;
```

B. Strictness of the feature constraints

As noted before, reconfiguration could result in violation of the feature constraints. In the previous example violation of feature constraints was strictly prohibited. The dynamic configuration where the high quality sensor is replaced by the low quality sensor cannot be accommodated since it is impossible to move from the start configuration to the target configuration without violating the feature constraints.

It may be desirable to temporarily allow violation of feature constraints during a reconfiguration phase, where the system configuration moves from one valid configuration to another. This would allow any feature to configure at any moment. Consequently, the system may get into a configuration that does not satisfy the feature constraints. It should be decided what is the allowed behavior in such a reconfiguration phase. This is discussed in more detail in the next section.

It should be noted that one may feel the need to express that some of the feature constraints really need to be satisfied at all times. Of course this can still be enforced.

Example:

If one only requires that the root feature `Fr` of the wiper system is present at all times, then this is achieved using the following model fragment. The resulting state space consists of 128 states, among which 8 initial states. There are 896 come and go transitions, but the come and go transitions of the root feature are not among those. Given the possibilities offered by CIF and the modularly defined feature constraints, it is possible to make more complex exceptions to the strictness of feature constraints.

```
alg bool r1 = Fr.present;
plant invariant r1;
```

IV. Modeling of behavior

Next, a model of the uncontrolled system and requirements is needed. These can then be used to synthesize a supervisory controller.

A. Behavior of the uncontrolled system

The plant modeling aims at capturing all uncontrolled behavior regardless of features, solely focusing on the potential behavior of the physical components. The aim is to get an one-to-one mapping of physical components to plants in CIF. This style of modeling is used previously in the context of modeling waterway locks [34] and automotive systems [30].

[1]The authors note that plant invariant definitions are supported for simulation in CIF, but currently not supported for the synthesis tool. In order to be able to perform synthesis, the modeler can straightforwardly manually restrict all come and go events toward states where sys_valid would evaluate to false.

Example:

In [14], no physical components for the wiper system are indicated. Therefore, a logical interpretation is made, making use of the feature model in Fig. 1 and the informal system description. The system constitutes of uncontrollable components: user control button (`button`), low quality sensor (`sensorLQ`), and high quality sensor (`sensorHQ`), and controllable components: low quality wiper (`wiperLQ`), and high quality wiper (`wiperHQ`). For each of the components an automaton is provided that describes its behavior, see Fig. 3. Although the different models use the same event names, because the events are defined within the automata, they are different, and do not synchronize.

The system that is composed of these five components has a state space of 216 states and 2,340 transitions, when there is no imposed (supervisory) control, i.e., the controllable events can occur at any time that they are defined in the system.

The CIF model consisting of the component automata does not yet take into account that in specific configurations specific components are not allowed to show behavior, because they are 'connected' to features that are not present. For example, the event `littleRain` of component `sensorHQ` (denoted `sensorHQ.littleRain`) is only available in case the high quality sensor feature is part of the configuration.

In the wiper system example, for the sensor components there is a one-to-one correspondence with the features. However, in case of the wiper components there is no one-on-one relation to the feature model because they need to perform the permanent wiping feature as well.

In general we require the modeler to indicate for each event that occurs in a component model which features need to be present for that event to be able to occur. In CIF this can then be captured by means of additional conditions on such events. For an event `e` that requires the features `Fa1` and `Fa2` this is achieved as shown in Listing 8. The connection is stated in the form of a plant, because it models the physical incapability to perform some events when certain features are not present.

Listing 8. Definition of link between events and features.

```
plant automaton event_feature_conditions:
  location: initial; marked;
    edge e when Fa1.present and Fa2.present;
end
```

The events that are made available by the button are available in every wiper system product, and are therefore not restricted. The events of the two sensor components are only restricted by the presence of the corresponding feature. A similar situation occurs for the wiper events, but for the permanent wiping events also the presence of the permanent wiping feature is required. For the wiper system, the connection between events and features is captured by the plant `event_feature_link` in Listing 9.

The way we expressed the availability of events in relation to the presence of features is conceptually similar as the solution adopted in featured transition systems [6]. In these featured transition systems events are also available condi-

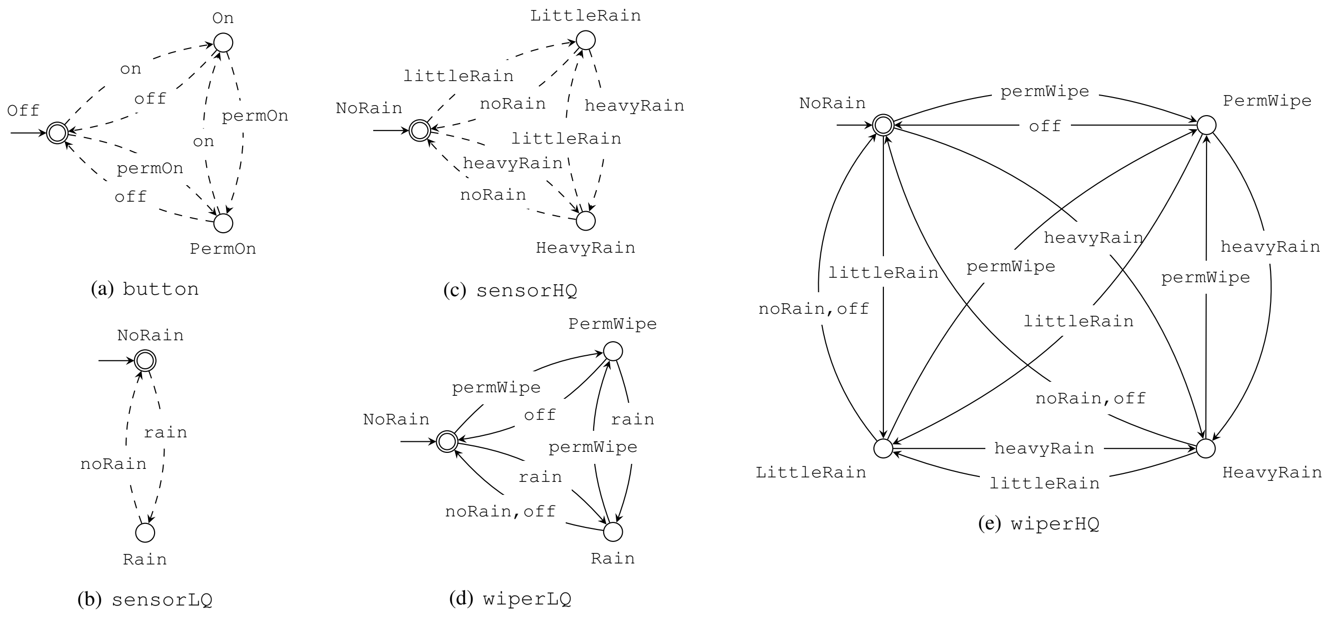


Fig. 3. Plant automata for the wiper components.

Listing 9. Connection between events and features.

```
plant automaton event_feature_link:
  location: initial; marked;
    edge button.off when true;
    edge button.on when true;
    edge button.permOn when true;

    edge sensorLQ.noRain when FsL.present;
    edge sensorLQ.rain when FsL.present;

    edge sensorHQ.noRain when FsH.present;
    edge sensorHQ.littleRain when FsH.present;
    edge sensorHQ.heavyRain when FsH.present;

    edge wiperLQ.off when FwL.present;
    edge wiperLQ.noRain when FwL.present;
    edge wiperLQ.rain when FwL.present;
    edge wiperLQ.permWipe when FwL.present and Fp.present;

    edge wiperHQ.off when FwH.present;
    edge wiperHQ.noRain when FwH.present;
    edge wiperHQ.littleRain when FwH.present;
    edge wiperHQ.heavyRain when FwH.present;
    edge wiperHQ.permWipe when FwH.present and Fp.present;
end
```

tionally depending on feature presence. In [15], it is shown that, alternatively, one could also capture featured transition systems in CIF. The most prominent difference between this paper and the approach using featured transition systems as well as the approach provided in [8], is that the description of the relation between features and events is separated from the behavioral models of the components. Another difference is that in the approach using featured transition systems not the uncontrolled system and requirements are modeled, but the supervisory controller is developed directly.

B. Behavioral requirements

In the previous section, we have discussed how to model the uncontrolled system and how to link event occurrences to availability of features. In this section, we discuss the modeling of behavioral requirements. As explained in Section II, such requirements are specified by means of automata that synchronize with the plant or by using state-based expressions such as event conditions and state invariants.

Example: Requirements wiper system

Below we state both the informal system requirements and the corresponding CIF formulation:

1) a wiper can only be turned off when the user has turned wiping off:

```
requirement wiperLQ.off needs button.Off;
requirement wiperHQ.off needs button.Off;
```

2) permanent wiping can only be done when the user has requested so:

```
requirement wiperLQ.permWipe needs button.PermOn;
requirement wiperHQ.permWipe needs button.PermOn;
```

3) the other wiping activities can only be done when wiping is turned on:

```
requirement wiperLQ.noRain needs button.On;
requirement wiperLQ.rain needs button.On;
requirement wiperHQ.noRain needs button.On;
requirement wiperHQ.littleRain needs button.On;
requirement wiperHQ.heavyRain needs button.On;
```

4) the wiper can stop wiping only if the sensor indicates that there is no rain:

```
requirement wiperLQ.noRain needs FsL.present => sensorLQ.NoRain;
requirement wiperLQ.noRain needs FsH.present => sensorHQ.NoRain;
requirement wiperHQ.noRain needs FsL.present => sensorLQ.NoRain;
requirement wiperHQ.noRain needs FsH.present => sensorHQ.NoRain;
```

5) the wiping level should be in accordance with available wiper feature and sensor reading:

```
requirement wiperLQ.rain needs FsH.present => (sensorHQ.
      LittleRain or sensorHQ.HeavyRain);
requirement wiperLQ.rain needs FsL.present => sensorLQ.Rain;

requirement wiperHQ.littleRain needs FsL.present=> sensorLQ.Rain;
requirement wiperHQ.littleRain needs FsH.present => sensorHQ.
      LittleRain;
requirement wiperHQ.heavyRain needs FsH.present => sensorHQ.
      HeavyRain;
requirement wiperHQ.heavyRain needs not FsL.present;
```

Requirements (4) and (5) illustrate requirement formulations that take the presence of features into consideration.

These requirements can be added to the model containing the plant behavior and the feature model. We consider the case the system initially is in a valid configuration and reconfiguration is restricted to only valid configurations, so essentially only the permanent wiper feature is able to reconfigure from the initial configuration. The state space of this system consists of 198 states and 1,025 transitions. Applying supervisory controller synthesis to this model results in the observation that the model was already nonblocking and controllable, i.e., the supervisor does not further restrict the behavior that was not already restricted in the plant and requirement definitions.

C. Behavior during configuration

In the previous example we have assumed that the system was always in a valid configuration. If we loosen this assumption and allow invalid configurations, decisions must be made about allowed behavior in such configurations. There are several ways to deal with the specification of allowed behavior during the configuration phase: (1) disable some events from occurring, and (2) additional requirements. Each of these approaches may be suitable for certain applications. In the following subsections these possibilities are investigated.

1) Disabling events: The approach in which some events need to be disabled in case the system is in an invalid configuration is the most elementary one. For each such an event, an event condition, such as the one presented for event `e` in Listing 10, can be defined that restricts that event to occur only when the system is in a valid configuration.

Listing 10. Disabling an event in an invalid system configuration.

```
requirement e needs sys_valid;
```

This approach assumes that the system will exhibit safe behavior by not exercising any of the events disabled in this way. As soon as the system returns to a valid configuration these events are no longer disabled.

2) Additional requirements: Another approach is stating additional requirements for the transitional situation. For the wiper system these are detailed in the next example.

Example: Dynamic configuration constraints for wiper system

In the wiper case, invalid configurations may sometimes be allowed. For example, an upgrade from a low quality wiper to a high quality wiper can be achieved via an invalid configuration with either no wiper features or with both wiper features. In the later situation, both wipers can perform wiping actions at the same moment. For this example, we will consider it unsafe that both wipers are present and wiping at the same time. This unsafe situation can be prevented by adding the requirement that in case both wiper features are present, at least one of them is in a non-wiping state. See Listing 11 for the CIF formulation of this requirement.[2]

[2]Opposed to *plant* invariants, *requirement* invariants are supported in CIF's supervisor synthesis tool.

Listing 11. Constraint during invalid configuration.

```
requirement invariant (FwL.present and FwH.present) => (wiperLQ.
      NoRain or wiperHQ.NoRain);
```

Applying supervisory controller synthesis to the described system results in a supervisory controller that, in addition to the requirements, applies the guards formulated in Listing 12 to the controllable events.

Listing 12. Additional guards provided by supervisory controller synthesis.

```
supervisor automaton sup:
  location: initial; marked;
    edge wiperHQ.heavyRain when (wiperHQ.NoRain or wiperHQ.
      LittleRain) and wiperLQ.NoRain or (wiperHQ.PermWipe or
      wiperHQ.HeavyRain);
    edge wiperHQ.littleRain when (wiperHQ.NoRain or wiperHQ.
      LittleRain) and wiperLQ.NoRain or (wiperHQ.PermWipe or
      wiperHQ.HeavyRain);
    edge wiperHQ.noRain when true;
    edge wiperHQ.off when true;
    edge wiperHQ.permWipe when (wiperHQ.NoRain or wiperHQ.
      LittleRain) and wiperLQ.NoRain or (wiperHQ.PermWipe or
      wiperHQ.HeavyRain);
    edge wiperLQ.noRain when true;
    edge wiperLQ.off when true;
    edge wiperLQ.permWipe when wiperHQ.NoRain;
    edge wiperLQ.rain when wiperHQ.NoRain;
end
```

The state space of the controlled system where invalid configurations are allowed and all previous requirements are added holds 27,648 states and 327,744 transitions. As a result of synthesis this controlled system is nonblocking (a marked state can always be reached), controllable (no uncontrollable events are disabled) and maximally permissive (no behavior is disabled that doesn't strictly need to be disallowed with regards to the aforementioned properties and requirements) by construction.

D. Component reappearance

Until now, the appearance and disappearance of features is not directly affecting the states of the involved components. Therefore, when a feature disappears, and in a future configuration reappears, the component(s) linked with this feature is still in the same state. Sometimes it may be required to start one or more components in a different state upon the status change of a feature. For instance, in the previous example it was not allowed for both wipers to be wiping when they are both present. It may happen that one of the wipers leaves the configuration while it is wiping. The supervisor will not allow the other wiper to start wiping, even if it is raining, because when the wiper that is already wiping (uncontrollably) reappears, the requirement in Listing 11 is not satisfied.

The plant model of a component can easily be adapted in order to transition to some desired reset state whenever the component enters or leaves the configuration. This is done by adding an edge, label with the `come` or `go` event of the respective component, from each state in the plant model of the component to its desired reset state. Because of synchronization, upon occurrence of the `come` or `go` event (from the feature plant) the transition with the same label in the component is taken as well. By adding this transition to each state of the component model, the proposed addition does not restrict reconfiguration possibilities.

Example: Re-initialization in the wiper system

Let us consider the case that we want the low quality wiper to turn off when it leaves the system. Applying the proposed approach results in the adapted plant automaton shown in Fig. 4. In this case, the state of the low quality wiper component is reset to the initial state, regardless of its current state.

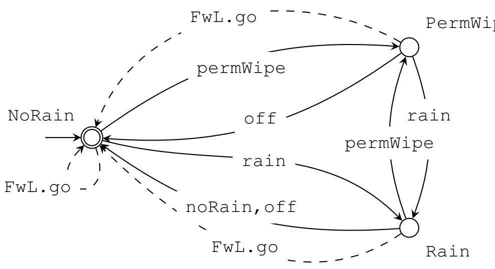


Fig. 4. Adapted plant for resetting the low quality wiper component in case of removal from the configuration.

V. Concluding Remarks

We have presented a method for engineering supervisory controllers for product families of which the valid configurations are described by a feature model and where dynamic configuration of the features is allowed. The CIF language has shown to be adequate for modeling the involved concepts. Several types of solutions have been elaborated upon, and some of them were illustrated using the wiper system example. Although the wiper system is small, feasibility has been demonstrated for a much larger system in [15].

References

[1] K. Pohl, G. Böckle, and F. van der Linden, *Software product line engineering - Foundations, Principles, and Techniques*. Springer, 2005.

[2] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. ACM Press/Addison-Wesley, 2000.

[3] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[4] D. Benavides, S. Segura, and A. Ruiz-Cortés, "Automated analysis of feature models 20 years later: A literature review," *Inform. Syst.*, vol. 35, no. 6, pp. 615–636, 2010.

[5] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," *ACM/IEEE Int. Conf. Softw, Eng.*, vol. 1, pp. 335–344, 2010.

[6] A. Classen, M. Cordy, P. Schobbens, P. Heymans, A. Legay, and J. Raskin, "Featured transition systems: Foundations for verifying variability-intensive systems and their application to ltl model checking," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1069–1089, Aug 2013.

[7] C. Baier and J. Katoen, *Principles of Model Checking (Representation and Mind Series)*. MIT Press, 2008.

[8] M. ter Beek, M. Reniers, and E. de Vink, "Supervisory Controller Synthesis for Product Lines Using CIF 3," in *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques: 7th Int. Symp.*, 2016, pp. 856–873.

[9] P. Ramadge and W. Wonham, "Supervisory control of a class of discrete event processes," *SIAM J. Control Optim.*, no. 1, pp. 206–230, 1987.

[10] C. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Springer, 2008.

[11] M. Sköldstam, K. Åkesson, and M. Fabian, "Modeling of discrete event systems using finite automata with variables," in *IEEE Conf. Decis. Control*, Dec 2007, pp. 3387–3392.

[12] K. Åkesson, M. Fabian, H. Flordal, and R. Malik, "Supremica - an integrated environment for verification, synthesis and simulation of discrete event systems," in *Int. Workshop Discrete Event Syst.*, Jul 2006, pp. 384–385.

[13] D. van Beek, W. Fokkink, D. Hendriks, A. Hofkamp, J. Markovski, J. van de Mortel-Fronczak, and M. Reniers, "CIF 3: Model-based engineering of supervisory controllers," *Tools Algorithms Construction Anal. Syst.*, pp. 575–580, 2014.

[14] A. Classen, "Modelling with FTS: a collection of illustrative examples," University of Namur, Tech. Rep., 2010.

[15] M. Tuitert, "Supervisory controller synthesis for dynamic software product lines," Master's thesis, Eindhoven University of Technology, 2017.

[16] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer, "Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study," Technische Universitat Braunschweig, Tech. Rep., 2013.

[17] J. Millo, S. Ramesh, S. Krishna, and G. Narwane, "Compositional verification of software product lines," *Lect. Notes Comput. Sci.*, vol. 7940, pp. 109–123, 2013.

[18] S. Kogekar, S. Neema, B. Eames, X. Koutsoukos, and A. Ledeczi, "Constraint-Guided Dynamic Reconfiguration in Sensor Networks," *Int. Symp. Inform. Process. Sensor Netw.*, pp. 379–387, 2004.

[19] H. Shokry and M. Babar, "Dynamic Software Product Line Architectures Using Service-Based Computing for Automotive Systems," *Int. Workshop Dynamic Softw. Product Lines*, pp. 53–58, 2008.

[20] M. Rosenmüller, N. Siegmund, M. Pukall, and S. Apel, "Tailoring dynamic software product lines," *ACM Int. Conf. Generative Programming Compon. Eng.*, pp. 3–12, 2011.

[21] L. Shen, X. Peng, J. Liu, and W. Zhao, "Towards feature-oriented variability reconfiguration in dynamic software product lines," *Int. Conf. Softw. Reuse*, no. 12, pp. 52–68, 2011.

[22] A. Sharifloo, A. Metzger, C. Quiton, L. Baresi, and K. Pohl, "Learning and Evolution in Dynamic Software Product Lines," in *Proc. Int. Symp. Softw. Eng. Adaptive Self-Maniging Syst.*, 2016, pp. 158–164.

[23] H. Gharsellaoui, J. Maazoun, N. Bouassida, and S. Ahmed, "A Software Product Line design based approach for real-time scheduling of reconfigurable embedded systems," *Comput. Human Behavior*, 2017.

[24] D. Basile, "Applying supervisory control synthesis to priced featured automata and energy problems," *Int. J. Softw. Tools Technol. Transfer*, vol. 21, no. 6, pp. 679–689, Dec 2019.

[25] D. Basile, M. ter Beek, P. Degano, A. Legay, G. Ferrari, S. Gnesi, and F. Di Giandomenico, "Controller synthesis of service contracts with variability," *Sci. Comput. Programming*, vol. 187, 2020.

[26] P. Heymans, P. Schobbens, J. Trigaux, Y. Bontemps, R. Matulevičius, and A. Classen, "Evaluating formal properties of feature diagram languages," *IET Softw.*, vol. 2, no. 3, pp. 281–302, Jun 2008.

[27] D. Benavides, P. Martín-Arroyo, and A. Ruiz-Cortés, "Automated reasoning on feature models," in *Int. Conf. Adv. Inf. Syst. Eng.*, 2005, pp. 491–503.

[28] B. van der Sanden, M. Reniers, M. Geilen, T. Basten, J. Jacobs, J. Voeten, and R. Schiffelers, "Modular model-based supervisory controller design for wafer logistics in lithography machines," in *ACM/IEEE Int. Conf. Model Driven Eng. Languages Syst.*, Sept 2015, pp. 416–425.

[29] R. Theunissen, M. Petreczky, R. Schiffelers, D. van Beek, and J. Rooda, "Application of supervisory control synthesis to a patient support table of a magnetic resonance imaging scanner," *IEEE Trans. Autom. Sci. Eng.*, vol. 11, no. 1, pp. 20–32, Jan 2014.

[30] T. Korssen, V. Dolk, J. van de Mortel-Fronczak, M. Reniers, and M. Heemels, "Systematic model-based design and implementation of supervisors for advanced driver assistance systems," *IEEE Trans. Intell. Transp. Syst.*, vol. 19, no. 2, pp. 533–544, 2018.

[31] F. Reijnen, M. Goorden, J. van de Mortel-Fronczak, and J. Rooda, "Supervisory control synthesis for a waterway lock," in *IEEE Conf. Control Technol. Appl.*, Aug 2017, pp. 1562–1563.

[32] C. Ma and W. Wonham, "Nonblocking supervisory control of state tree structures," *IEEE Trans. Automat. Contr.*, vol. 51, no. 5, pp. 782–793, 2006.

[33] J. Markovski, K. Jacobs, D. van Beek, L. Somers, and J. Rooda, "Coordination of resources using generalized state-based requirements," in *Int. Workhop Discrete Event Syst.*, 2010, pp. 287–292.

[34] M. Goorden, J. van de Mortel-Fronczak, M. Reniers, and J. Rooda, "Structuring multilevel discrete-event systems with dependency structure matrices," in *IEEE Conf. Decis. Control*, Dec 2017, pp. 558–564.