Efficient Supervisor Synthesis for Feature Models

Wetzels, B.H.J.

Award date:
2021

Eindhoven University of Technology

Department of Mechanical Engineering

Control System Technology

Graduation project

B.H.J. Wetzels

Efficient Supervisor Synthesis for Feature Models

studentno. 0898706

TU/e supervisor: dr. ir. M.A. Reniers
External supervisor: ir. D. Hendriks
TU/e supervisor: ir. S.B. Thuijsman

Eindhoven, October 11, 2021
This report was made in accordance with the TU/e Code of Scientific Conduct for the Master thesis.

Abstract

Supervisory control theory is a model-based approach that is used to compute a controller that guarantees a desired behavior of a system. Unfortunately, the synthesis of such a controller can need a lot of computing effort and time. In order to reduce this, a couple of nonmonolithic synthesis methods have been proposed in existing literature. These methods are described and compared in the first part of this thesis This results in a clear overview of the methods and the properties that belong to controlled systems using these methods. The second and main part of the thesis focuses on supervisory control models that use feature models to describe valid configurations in the system. A modular method is used to reduce the required effort in supervisor synthesis for these systems. In this method, synthesis is performed for every requirement and the corresponding parts of the plant. Next, a new proposition is introduced in which the feature model is (partly) omitted during the synthesis of a supervisor. The resulting supervisor is then used on the plant from before omitting the feature model. This results in a significant decrease of the effort required for synthesis. The new method is proposed for both static and dynamic configurations of a feature model and validated on two existing models. Also, the consequences of the proposition on the safety, controllability, maximal permissiveness and nonblockingness of the resulting supervisor are discussed.

Contents

1 Introduction **1**

2 Preliminaries **2**
2.1 Supervisory Control Theory . . . 2
2.2 Supervisor synthesis . . . 3
2.3 CIF . . . 4
2.3.1 Binary decision diagrams . . . 4
2.3.2 Peak used BDD nodes and BDD operation count . . . 5
2.3.3 Plant and requirements in CIF . . . 5

3 Application of feature models **8**
3.1 Applying feature models in CIF . . . 9
3.2 The Wiper System . . . 10
3.3 Body Comfort System . . . 11

4 Nonmonolithic synthesis methods **14**
4.1 Modular supervisor synthesis . . . 14
4.2 Decentralized supervisor synthesis . . . 14
4.3 Hierarchical supervisor synthesis . . . 15
4.4 Multilevel supervisor synthesis . . . 17
4.5 Comparison . . . 18

5 Modular supervisor synthesis in feature models **20**
5.1 Applying modular synthesis . . . 20
5.2 Results . . . 23

6 Removing feature model from synthesis **25**
6.1 Removing the entire feature model . . . 25
6.2 Consequences of removing the feature model . . . 26
6.2.1 Consequences for static configurations . . . 26
6.2.2 Consequences for uncontrollable dynamic configuration . . . 28
6.2.3 Consequences for controllable dynamic configuration . . . 28
6.3 Effect on the effort . . . 30

7 Conclusions and recommendations **33**

References **34**

Appendix **37**
Static Wiper System . . . 37
Dynamic Wiper System . . . 40
Static BCS . . . 43
Dynamic BCS . . . 53

1 Introduction

A wide range of cyber-physical systems are used in industrial and non-industrial applications. In order to control these systems which consist of a physical system that is monitored and controlled by algorithms, supervisory control theory can be used [1]. This is a model-based approach in which a model is created from the physical system in which all possible behavior is included, called the plant. Another model is made in which requirements are defined to specify the desired behavior of the system. The requirements and plant models are used to synthesize a supervisor. The synthesis algorithms usually guarantee a safe, nonblocking, maximal permissive and controllable controlled system.

Systems can consist of many interacting subsystems which create a complex cyber-physical system which results in a complex supervisory control model. For large systems, the state space explodes during the supervisor synthesis process, which can cause supervisor synthesis to be infeasable due to a lack of memory or time. Using Binary Decision Diagrams [2, 3], further referred to as BDDs, moderates the large computational effort caused by the explosion and [4] introduces two deterministic and platform independent metrics to quantify the effort of a supervisor synthesis, namely the peak used BDD nodes and the BDD operation count. They represent the space and time effort respectively.

In order to reduce the effort for synthesis, a couple of nonmonolithic synthesis methods have been introduced in the existing literature which include modular [5–7], decentralized [8–12], hierarchical [13–16] and multilevel [17–20] supervisor synthesis. In these methods, the control task of supervising the system does not depend on one supervisor, monolithic supervision, but rather is divided over multiple supervisors. In the first part of this thesis, a brief summary of the existing methods is provided which results in an overview of the methods and their properties.

The second and main part of the thesis focuses on the reduction of effort required for the synthesis of a supervisor that is used to control feature models. These models do not only consist of requirements and component plants, but also of a feature model which defines a range of valid configurations for the system. These feature models can either be static or dynamic. In static feature models, there is only one configuration that is used throughout the use of the system. For dynamic feature models, the configuration can be reconfigured during the execution of the system. Using a modular method for synthesis of a supervisor reduces the effort. In order to reduce the effort even more, a new proposition is introduced in which the feature model is (partly) removed from the plant that is used in the synthesis process.

In Chapter 2, the existing knowledge that has been used during the thesis is given. How feature models are modelled in the modeling language and tool CIF is described in Chapter 3. The nonmonolithic synthesis methods are described and compared in Chapter 4. Then, in Chapter 5, modular synthesis is performed on two existing feature models. In Chapter 6, the new proposition is explained and applied to the two models. In this chapter, also the consequences of the proposition on the safety, controllability, maximal permissiveness and nonblockingness are discussed. Finally a conclusion is made.

2 Preliminaries

In this chapter, the existing knowledge that is used in this research is briefly explained. Supervisory control theory, monolithic supervisor synthesis is explained in the first section. In the second section feature models and their constraints are explained. CIF, supervisor process effort metrics and the application of models in CIF are explained in the last section of this chapter.

2.1 Supervisory Control Theory

In order to control cyber-physical systems, supervisory control theory can be used [1]. This is a model-based approach that is used to find a controller that guarantees the desired behavior of a system. It requires a model of the uncontrolled system in which every physically possible behavior is allowed, which is called the plant often denoted by G. It also requires requirements which dictate the desired behavior of the controlled system. These requirements model how a system or part of a system should behave according to the user requirements. Undesirable states that should not be reached according to the requirements are called bad states. The supervisor synthesis process uses both the plant and the requirements to synthesize a supervisor which disables or enables certain events in the system for certain conditions, and restricting the system's behavior by doing that, in order to guarantee that the system only performs the desired behavior satisfying the requirements. This is called a safe supervisor.

The model consists of a number of Extended Finite Automata, further referred to as EFA, which are finite state automata that have been extended with variables, guards and updates. In Figure 2.1, an example of an EFA is shown.

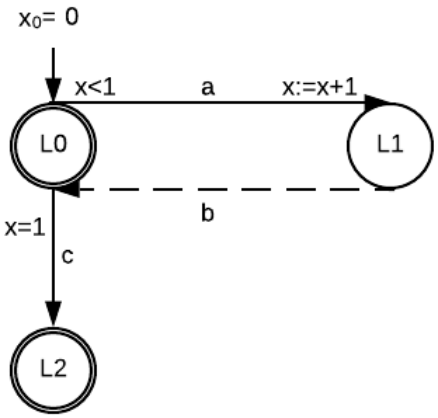


Figure 2.1: *An example of a simple EFA A.*

This automaton A can be defined as a 7-tuple A [21, 22].

$$A = (L, V, \Sigma, \longrightarrow, L_0, V_0, L_m)$$

In A, L is the finite set of locations, V is the finite set of discrete variables, Σ is the alphabet, the finite set of events, $\longrightarrow$ is the finite set of edges, $L_0 \subseteq L$ and $V_0 \subseteq V$ are the sets of initial locations and initial variable values respectively and $L_m \subseteq L$ is the set of marked locations. The active location and variable values is called the system state. In case of the example in Figure 2.1, there are three locations which are represented by circles of which $L0$ and $L2$ are marked, represented by double circles. The initial state, represented by the location with a dangling incoming arrow, is $L0$ with initial value for variable x is $x_0 = 0$.

An edge $\longrightarrow_k$ for an Extended Finite Automaton can be defined as a 5-tuple $\longrightarrow_k$.

$$\longrightarrow_k = (l_{0,k}, l_{t,k}, \sigma_k, g_k, u_k)$$

An edge is the transition from one state to the next state. $l_{0,k} \in L$ and $l_{t,k} \in L$ are the origin and target location before and after the edge. $\sigma_k \in \Sigma$ is an event corresponding to the edge. In Figure 2.1, edges are represented by arrows with the corresponding events written close to the arrow. g_k is a guard predicate. This means that the edge is only enabled when the evaluation of the predicate is true. u_k is the update expression which gives a new value to variables after the edge has been executed. In Figure 2.1, the guard is written at the beginning of the corresponding edge and the update is written at the end of the edge.

In order for a supervisor to be able to restrict the behavior of an automaton, it has to be able to enforce or disable certain edges. However, the alphabet is partitioned into two disjunct sets of events, a controllable event set Σ_c and an uncontrollable event set Σ_u. Whenever an edge $\longrightarrow_k$ has an uncontrollable event $\sigma_k \in \Sigma_u$, the edge itself is uncontrollable as well, which is indicated with a dashed arrow in the example. The same can be said about controllable edges corresponding to controllable events. These are indicated with solid arrows. When an edge is uncontrollable, the supervisor can not disable this edge.

Example 2.1. In the example, there is a controllable edge with event c from location $L0$ to $L2$ with a guard expression $x = 1$, meaning that the edge can only take place when x has value 1. For the controllable edge with event a, the edge can only happen when $x < 1$. The update increases the value of x by 1 every time the edge is executed. The edge with event b is uncontrollable.

Eventually, the supervisor is synthesized using the plant and requirements. The supervisor makes the controlled system safe, controllable, non-blocking and maximal permissive. A safe controlled system is a system that satisfies all requirements, meaning that it can not reach any bad states. A nonblocking supervisor is a supervisor for which the system can always reach a marked state. A controllable supervisor only disables controllable events and a maximally permissive supervisor poses the minimum amount of restrictions in order to enforce the three previous properties.

2.2 Supervisor synthesis

Algorithm 1 `SS` (Supervisor Synthesis)

Input: Plant SEFA $A_S = (X, \Sigma, E, X_0, X_m)$, mutual state exclusion requirements MS, state-edge exclusion requirements SE
Output: Supervisor SEFA S

1: $(N, E_S) =$ `applyRequirements`(MS, SE, E)
2: **repeat**
3: $N' = N$
4: $N = (N, E_S, X_m)$
5: $B = (\texttt{true}, \{(\sigma, g, u) \in E | \sigma \in \Sigma_u\}, \neg N)$
6: $N = N \wedge \neg B$
7: **until** $N = N'$
8: **for all** $(\sigma, g, u) \in E_S$ with $\sigma \in \Sigma_c$
9: $g(X) = g(X) \wedge \exists_{X^+}[N(X^+) \wedge u(X, X^+)]$
10: **end**
11: $S = (X, \Sigma, E_S, X_0 \wedge N, X_m \wedge N)$

The synthesis process that is used to make a supervisor is described in the algorithm in Algorithm 1 from [23] and is based on [22]. This algorithm uses Symbolic Extended Finite Automata, SEFA, instead of the EFA that have been described in Section 2.1. The symbolic representation uses predicates that are efficiently represented by BDDs. A SEFA A_S is a 5-tuple. X is a set of symbols that represent automata and variables and the valuation of these symbols defines the state. Σ is the alphabet and X_0 and X_m are predicates on the symbols X that represent initial and marked states. $e \in E$ is an edge and is a triple with event σ, guard expression g and update predicate u. X^+ represents the new valuation after the edge has been taken.

$$A_S = (X, \Sigma, E, X_0(X), X_m(X))$$
$$e = (\sigma, g(X), u(X, X^+))$$

Example 2.2. As an example [23], take a traffic light with two automata $LightA$ and $LightB$ with locations Red and $Green$ and edges $green_A$, red_A, $green_B$, red_B turning $LightA$ and $LightB$ green and red respectively. Initially both lights are red. The symbols would be {$LightA$, $LightB$}, the initial state predicate is $LightA.Red \wedge LightB.Red$ and the edge that turns $LightA$ green is $(greenA, LightA.Red, LightA^+.Green \wedge LightB^+ = LightB)$.

1. The first step during supervisor synthesis is applying the requirements. In this step, safe state predicates N and a set of safe edges E_S are defined for which the requirements hold.
2. Next, the algorithm computes a set of nonblocking state predicates N, starting from the safe predicate and a set of blocking predicates B. The further calculation is done by means of a backward reachability search. The predicates of the blocking states are then removed from N and this process is repeated until N does not change anymore.
3. In the next step, the guards of the controllable edges are strengthened such that they are true when the nonblocking predicate in the target state of the edge is true.
4. Finally, the supervisor S is constructed. In this supervisor, the initial state predicates are the conjunction of the safe and nonblocking state predicates N and the initial states from the model in order to initialize the system in a safe, nonblocking state.

2.3 CIF

CIF is a Compositional Interchange Format for writing models of physical systems among other things and is part of the Eclipse Supervisory Control Engineering Toolkit (Eclipse ESCET^{TM})[1] [24]. This gives the ability to implement extended finite automata and also allows specifications which can be used to synthesize a supervisor in CIF [22]. The supervisor synthesis algorithm is based on BDDs. With the implementation of the work in [4], two metrics that use these BDDs can be used to measure the effort required for the synthesis of a supervisor.

2.3.1 Binary decision diagrams

An EFA can be described as a Binary Decision Diagram, BDD, [2,3]. They show whether a predicate using locations and variables is *true* or *false*. This is a diagram that consists of decision nodes, edges and terminal nodes. At a decision node, the value of on of the variables in the predicate is evaluated *true* of $false$. Locations are treated the same way as variables by checking whether the location is *true* or $false$. From the parent node, two edges representing *true* and $false$ for the evaluated variable lead to two corresponding child nodes where a different variable is checked. At the end of the diagram all variables and locations in the predicate are evaluated and the trees end in terminal nodes $True$ and $False$.

Example 2.3. In Figure 2.2, the expression $x_1 \wedge x_2$ is evaluated. The solid arrow is an evaluation *true* and a dashed arrow means that the node evaluates to $false$. The expression is $True$, 1, in the terminal node when both nodes are *true* and $False$, 0, when one of the nodes is $false$.

In this research, a BDD refers to a reduced ordered binary decision diagram [25]. These diagrams are canonical for a certain order of variables and use a minimal required number of decision nodes. This is important, since the number of decision nodes is the size of a BDD. The size consequently has an influence on the memory of a computer that is required and the computation time of a synthesis procedure.

[1] The ESCET toolset and documentation is open source and freely available at `https://www.eclipse.org/escet/`. 'Eclipse', 'Eclipse ESCET' and 'ESCET' are trademarks of Eclipse Foundation, Inc.

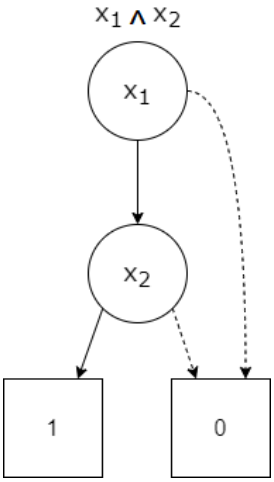


Figure 2.2: *Example of a BDD [26].*

2.3.2 Peak used BDD nodes and BDD operation count

To assess the efficiency of an algorithm, multiple metrics have been proposed in literature to measure the effort that is required for supervisor synthesis. Wall clock time and peak random access memory usage are popular metrics, but in [4] two other metrics were proposed, namely peak used BDD nodes and BDD operation count. These are preferred over the other metrics since they are deterministic. This means that with the same input, the output is the same for every execution. Also, the computing platform has no influence on the output, meaning that the output will be the same on a regular computer and on a supercomputer. The number of decision nodes during a synthesis influences required effort. During the synthesis the number of nodes that are used to describe the system is not constant. The maximal number of the nodes that are used is also the minimal number of nodes for the given variable order that are necessary to represent the predicates to solve the synthesis. The peak used BDD nodes is therefore a good representation of the required effort in terms of space according to the authors of [4]. Whereas the peak used BDD nodes is a metric that looks at the space of the supervisor synthesis, the required effort in terms of time can be described by the BDD operation count. The synthesis consists of performing operations on BDDs, which influences the computational time of the synthesis process. [4] also concludes that the BDD operation count therefore makes the counting of these operations a good representation of the time effort of the synthesis .

2.3.3 Plant and requirements in CIF

All possible behavior of the components is modelled in the plant consisting of plant automata. In Figure 2.3, an example of two plant automata are shown. In CIF, every automaton has at least one location which could be initial or marked or both. $Init$, $S0last$ and $S1last$ are the locations for automaton $BinarySensor$ in the example and $Init$, $V0last$ and $V1last$ are the locations for automaton $Movement$. From each location, edges can be declared, which are denoted as an *edge*. They might have a guard defined after *when*, an update after *do* and a target location after *goto*. The events that are used in the edges can be defined locally in the automaton or globally outside an automaton. In the example, the events are defined globally. The events are also defined as controllable or uncontrollable.

Apart from the plant components, requirements are usually added to define the desired behavior. There are three ways in which requirements can be denoted in CIF. First of all an automaton can be used to define a requirement as is done in Figure 2.4. When an event is shared, meaning that it is used in multiple automata, an edge corresponding with that event can only be executed when the event is enabled in all of those automata. In this way, the requirement automaton dictates the desired behavior of the plant.

```
uncontrollable S0, S1;

plant BinarySensor:
    location Init:
        initial; marked;
        edge S0 goto S0last;
        edge S1 goto S1last;
    location S0last:
        marked;
        edge S0;
        edge S1 goto S1last;
    location S1last:
        marked;
        edge S1;
        edge S0 goto S0last;
end
```

```
controllable V0, V1;

plant Movement:
    location Init:
        initial; marked;
        edge V0 goto V0last;
        edge V1 goto V1last;
    location V0last:
        marked;
        edge V0;
        edge V1 goto V1last;
    location V1last:
        marked;
        edge V1;
        edge V0 goto V0last;
end
```

Figure 2.3: *Example of two plant automata [27].*

Example 2.4. The requirement automaton in Figure 2.4 is based on a section of the swarm formation model from [27]. The requirement automaton dictates that initially, $V0$ is not enabled. Only after $S0$ has been executed, $V0$ is enabled, since then the location has changed to $NoRobotPerceived.V0allowed$. When $V0$ or $S1$ is executed, $V0$ is again disabled by going to $NoRobotperceived.V0Nallowed$.

```
requirement NoRobotPerceived:
    location V0Nallowed:
        initial; marked;
        edge S0 goto V0allowed;
        edge S1;
    location V0allowed:
        marked;
        edge S1 goto V0Nallowed;
        edge V0 goto V0Nallowed;
        edge S0;
end
```

Figure 2.4: *Example of a requirement automaton [27].*

The second kind of requirement definition is a state invariant. The invariant consists of a predicate, which uses the variables and locations of the plant model. When a plant state violates the predicate, then the supervisor will prevent the controlled system from reaching this plant state. The invariant thus defines a condition that must always hold in the controlled system. The invariant should hold in the initial state and all states that can be reached by edges from that state. This is thus also a way to dictate desired behavior. An advantage of this kind of requirement is that the invariant is applied for all edges. It also states an explicit condition for states in relation to variables. The predicates can use logical operators like negation $\neg$, conjunction $\wedge$, disjunction $\vee$ and implication $\implies$ [28].

Example 2.5. A state based invariant is shown in Figure 2.5. This requirement is also based on the swarm aggregation model of [27]. The requirement states that when $BinarySensor$ is in $Init$, $Movement$ must be in location $Init$. This means that the supervisor will disable $V0$ and $V1$ when the system is in $BinarySensor.Init$.

```
requirement BinarySensor.Init => Movement.Init;
```

Figure 2.5: *Example of a state based invariant requirement.*

A third way to define a requirement is using state-event exclusion invariants. This requirement dictates a condition that needs to hold in order to execute or not execute an event using a predicate.

The requirement is now of the form $\rightarrow \sigma \implies predicate$. With this kind of requirement it is easy to define conditions for the execution of events.

Example 2.6. In Figure 2.6 an example is displayed. This is also based on the swarm aggregation model from [27]. The requirement restrict the event $V0$ directly by enabling the event only when the system satisfies the predicates. This requirement could also have been written as `requirement not BinarySensor.S0last disables V0;`. In that case, $V0$ is disabled when the predicate is satisfied.

```
requirement V0 needs BinarySensor.S0last;
```

Figure 2.6: *An example of a state-event exclusion invariant requirement.*

3 Application of feature models

In this chapter, feature models are introduced and explained. The way that feature models are applied in CIF is also explained and exemplified with a model of a wiper system. This system is one of two cases on which experiments are done, which are shown later on in the thesis. The other case is the system of a body comfort system. This system is also briefly explained in this chapter.

Nowadays in the development of systems, ways are sought to reduce development and production costs and shorten the time of development and production. This can be done by reusing software and hardware components of a system. To facilitate the reuse of these components, software product line engineering can be used. In order to find valid product configurations of these components, feature models are made [29–31]. Feature models represent the information of all possible products of a product line in terms of hierarchically arranged features and different relations among the features. The features in these models are aspects of a system which are specified by requirements or characteristics. For example, a feature of a wiper system could be the the low quality wiper FwL, as is the case in the feature model in Figure 3.1 [31].

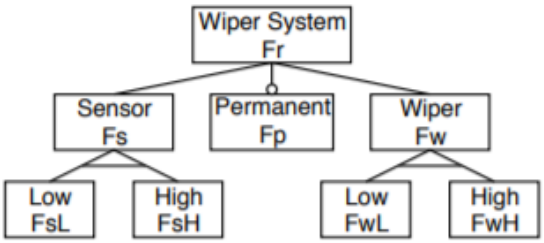


Figure 3.1: *Example of a feature model [31].*

In the hierarchical set of features direct relations between features are included. These feature constraints represent parent-child relations and cross-tree constraints. In Figure 3.2, different feature constraints are presented [32].

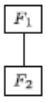
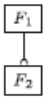
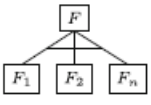
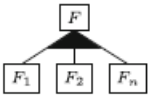
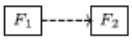
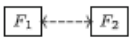
relationship		formula
root		$F_0 \iff true$
mandatory		$F_1 \iff F_2$
optional		$F_2 \implies F_1$
alternative		$(F_1 \iff (\neg F_2 \wedge \cdots \wedge \neg F_n \wedge F)) \wedge \cdots \wedge (F_n \iff (\neg F_1 \wedge \cdots \wedge \neg F_{n-1} \wedge F))$
or		$F \iff (F_1 \vee F_2 \vee \cdots \vee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

Figure 3.2: *A list of feature constraints [32].*

In short, a *root feature* is present in every valid configuration of the system. A *mandatory constraint* means that when the parent feature is present in the configuration, the child feature has to be present as well. The *optional constraint* means that when a parent feature is present, the child feature may be present and that when a child feature is present, the parent must be present as well. An *alternative constraint* means that when the parent feature is present, only one of the child features is present and an *or constraint* means that at least one child feature is present. The constraints thus far are parent-child relations. For the cross-tree constraints, there is a *requires constraint*, which means that a certain feature can only be present when another feature is also present. There is also a cross-tree *excludes constraint* which means that certain features exclude each other from being present in the same configuration.

In [30], multiplicity, called cardinality, of features is considered as relation as well. A feature cardinality is denoted as $[n..m]$, which means that a particular feature can be present in a product multiple times with n as a lower bound and m an upper bound on the instances. Note that a mandatory relationship is the same as a feature cardinality [1..1] and an optional relationship is the same as a feature cardinality [0..1]. A group cardinality $\langle n..m \rangle$ limits the the number of child features that are allowed to be present in a product when the parent is present. $\langle 1..1 \rangle$ is the same as an alternative relationship and $\langle 1..N \rangle$ is the same as an or relationship with N child features. Feature models can also be extended or attributed, which means that it includes more information about a feature, called a feature attribute [29]. An attribute usually has a name, domain and a value. These attributed can also be used in constraints.

Systems with feature models can have a static or dynamic configuration. When the model has a static configuration, the configuration of the system does not change during the execution of the system. This means that when certain features are initially present or absent, they stay that way. When a model has a dynamic configuration, the presence of the features can change during the behavior of the system, meaning that there is a reconfiguration. It can happen that in such case a violation of the feature constraints occurs. When making the model, it has to be decided whether that is allowed or not. By disabling events during reconfiguration or by adding additional requirements, the desired behavior during reconfiguration can be dictated.

3.1 Applying feature models in CIF

CIF can also be used to synthesize a single supervisor for the model of a product line [32, 33]. The models of these systems consist of the regular specifications, but the plant is not just the set of component automata as in Subsection 2.3.3. The feature model itself is also part of the plant and consists of a couple of components.

In order to include a feature model in the CIF model, a *feature plant automaton* is made for every feature as in Figure 3.3. To define the presence or absence of features, a boolean variable *present* is introduced. In the static version in Figure 3.3a a dummy location is present since CIF requires a location in any automata. In the dynamic version in Figure 3.3b, *come* and *go* events are added which represent the entering and leaving of the feature in the configuration. These events can be controllable or uncontrollable. In the example, they are uncontrollable. The static version of a model could be seen as a special case of a dynamic model in which the *come* and *go* events blocked.

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

(a) *Static feature automaton.*

```
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
```

(b) *Dynamic feature automaton.*

Figure 3.3: *Feature automata.*

In order to model the in-tree and cross-tree constraints, boolean algebraic variables can be introduced using the formulas from Figure 3.2, see Figure 3.4 for an example. *sys_valid*, which is used in automaton *Validity*, combines all these variables. This automaton demands that *sys_valid*, and therefore all in- and cross-tree constraints, is initially true. In static configurations, features can not come or go, meaning that *sys_valid* does not change and stays true. For dynamic configurations, *sys_valid* could be violated during reconfiguration. The model maker should therefore decide which behavior should be allowed during reconfiguration as was mentioned in the previous section of this chapter.

To couple the plant component automata with the corresponding feature automata, a *presence check plant automaton* is made in which it is stated that a certain feature needs to be present in order to enable the events of components corresponding to that feature. An example is presented in Figure 3.5. In the example, all events are uncontrollable, but the a presence check automaton can also be used for controllable events.

```
alg bool r1 = Fr.present ; //Root
alg bool r2 = Fp.present => Fr.present ; //Optional
alg bool r3 = Fr.present <=> ( Fs.present and Fw.present ) ; //Mandatory
alg bool r4 = (FsL.present <=> ( not FsH.present and Fs.present ))
              and (FsH.present <=> ( not FsL.present and Fs.present )); //Alternative
alg bool r5 = (FwL.present <=> ( not FwH.present and Fw.present ))
              and (FwH.present <=> ( not FwL.present and Fw.present )); //Alternative
alg bool sys_valid = r1 and r2 and r3 and r4 and r5;

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

Figure 3.4: *Definition of feature constraints belonging to the wiper system in Section 3.2.*

```
plant automaton PRESENCE UNCONTROLLED :
    location:
        initial; marked;
        edge button.u_permOn when Fp.present ;
        edge sensorLQ.u_noRain when FsL.present ;
        edge sensorLQ.u_rain when FsL.present ;
        edge sensorHQ.u_noRain when FsH.present ;
        edge sensorHQ.u_littleRain when FsH.present ;
        edge sensorHQ.u_heavyRain when FsH.present ;
end
```

Figure 3.5: *A presence check plant automaton belonging to the wiper system in Section 3.2.*

3.2 The Wiper System

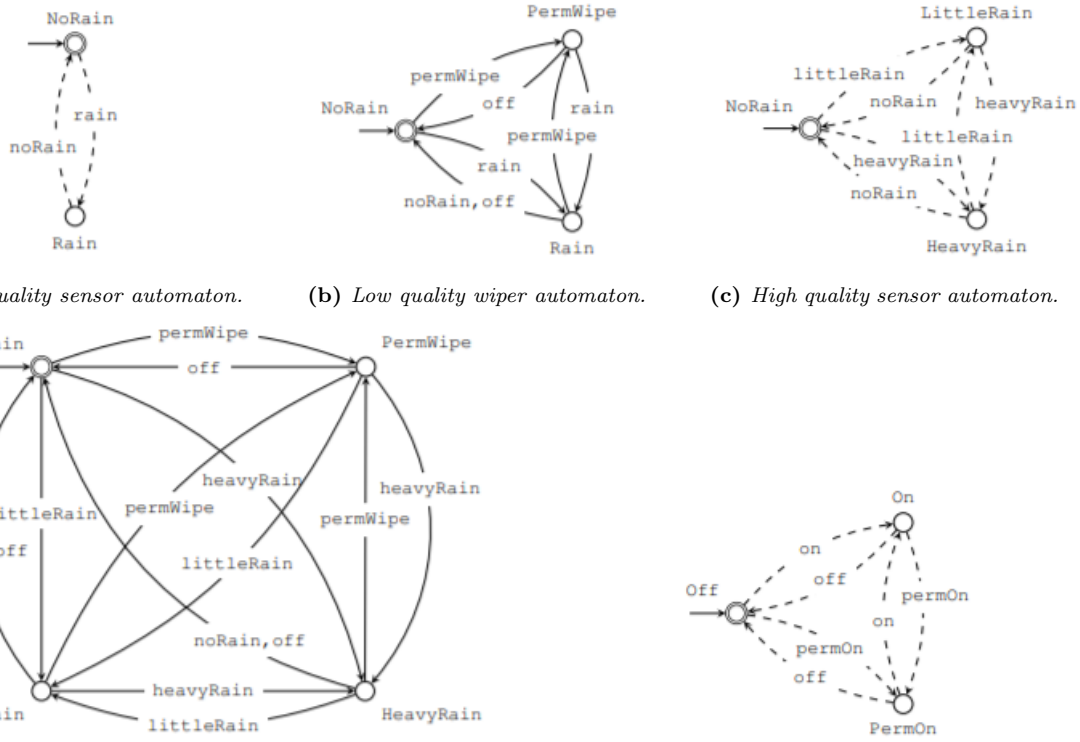
The examples of the previous section all come from a model for the product line of a windscreen wiper of a vehicle that has been used in multiple sources [31, 33, 34]. This model serves as one of the two cases that are used in the experiments that are addressed later in this thesis and its feature model is presented in Figure 3.1. In this section, the wiper system will serve as an example in order to highlight the division of components in a model.

The component plant consists of a sensor of either high or low quality and a wiper of either high or low quality. In Figure 3.6, the automata of the components are shown. The low quality sensor in Figure 3.6a can detect whether it is raining or not, while the high quality sensor in Figure 3.6c can also detect whether it is raining heavily or only a little. The high quality wiper in Figure 3.6d can wipe at two speeds, *LittleRain* or *HeavyRain*, while the low quality wiper in Figure 3.6b can only wipe at a single speed, *Rain*. When a permanent wiper state is chosen in the button automaton in Figure 3.6e, the wiper can wipe permanently. The component plant automaton for the low quality sensor

is coupled with FsL in the feature model. The high quality sensor is coupled with FsH, the low quality wiper with FwL, the high quality wiper with FwH and the wipers and button are coupled with Fp.

The entire model with static feature configuration, hereafter named static wiper model, is listed in the Appendix. In this divided code, the different parts of the system are clearly identifiable. The component automata from Figure 3.6 are listed in Listing 2. The feature model consists of the combination of Listing 1 in which the feature plant automata, algebraic booleans and $Validity$ automaton are defined and Listing 3 in which the presence check automata are defined. The plant is thus the combination of Listing 1, 2 and 3. Finally, in Listing 4, the requirements are defined.

The wiper system with a dynamic feature configuration, hereafter mentioned as the dynamic wiper system, is also used later in the thesis. This model can be found in Listing 5 in the Appendix. During the reconfiguration, the two different wipers could be present at the same time. In order to prevent the unsafe situation that they are wiping at the same time, an extra requirement is made that states that when both wipers are present, at least one of them is not wiping. Also, re-initialization has been added to the component automata for both wipers. This means that when a wiper leaves the system, the wiper returns to the $NoRain$ location. These adaptations come from [33].



(a) *Low quality sensor automaton.* **(b)** *Low quality wiper automaton.* **(c)** *High quality sensor automaton.*

(d) *High quality wiper automaton.* **(e)** *Button automaton.*

Figure 3.6: *Component automata for the wiper system [33].*

3.3 Body Comfort System

The wiper system was only a small system, meaning that to validate the methods even further, a larger system is modelled. The Body Comfort System, BCS, is a model of a product line of a vehicle in which features can be chosen by customers [35]. In Figure 3.7, the feature model of the BCS is shown. The BCS is a system in which components work together to be a final product. In order to clarify the system a bit, the functionalities are described below [34].

- The Power Window is an electronic window that can detect a clamped finger, called Finger

Protection.

- The Adjustable and Heatable Exterior Mirror is an exterior mirror that can be adjusted and that is possibly heated for visibility improvement.
- The Human Machine Interface is a series of optional LEDs that indicate whether certain systems are active.
- The Central Locking System could lock all doors simultaneously. The Automatic Locking, locks the doors during driving.
- With the Remote Control Key option, the vehicle can be locked remotely. The Safety Function locks the car again after a certain time interval. The Control Automatic Power Window enables the control of the window with the Remote Control Key. There is also an option to control the mirrors by the Key.

Also for the BCS models exist with a static and dynamic feature configuration which are hereafter mentioned as the static and dynamic BCS and listed in Listing 6 and Listing 7 in the Appendix respectively.

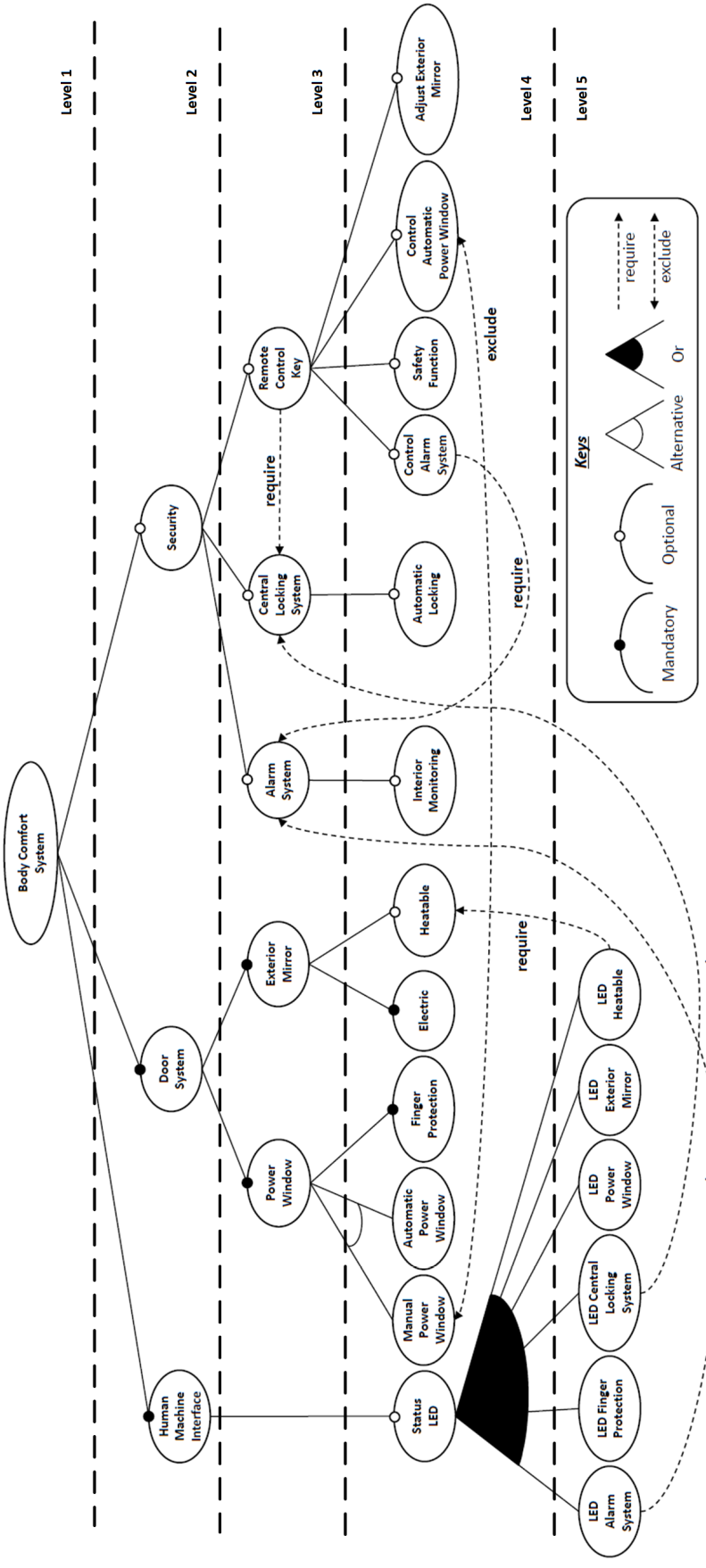


Figure 3.7: *The feature model of the BCS [35].*

4 Nonmonolithic synthesis methods

During monolithic supervisor synthesis, one supervisor is synthesized to control the entire system. However, for large systems a state space explosion occurs during the synthesis process. This means that for larger systems the computation of a supervisor is infeasible due to memory or time constraints. In order to prevent or reduce the required effort during the synthesis of a supervisor, several methods have been introduced in which the control task has been divided over multiple supervisors to control the system, which is called modular supervisor synthesis. However, the term modular supervisor synthesis is also used as a particular subcollection of nonmonolithic synthesis methods. To clearly distinguish the two different meanings for modular syntheses, the term nonmonolithic synthesis is used for the collection of all methods in this chapter, whereas modular synthesis is used for the subcollection of methods from Section 4.1.

In order to find a method to reduce the effort for feature models, several nonmonolithic methods are briefly described in this chapter. These methods already exist in literature. The contribution of this chapter is Table 4.1. This table is a clear overview of the methods that are discussed and the properties that belong to the methods and their resulting controlled systems. In Chapter 5, one of the methods is chosen and used on feature models.

4.1 Modular supervisor synthesis

In [5] a method for modular supervisor synthesis is proposed. Instead of synthesizing a supervisor from the global plant and one global specification, the authors use multiple specifications. A supervisor is made for each requirement using that requirement and the global plant. This means that multiple supervisors are computed. Events are only enabled when it is enabled by all supervisors. The authors strive to synthesize nonblocking supervisors. However, nonblockingness is not guaranteed in this method and can only be checked with a nonconflicting check, meaning that the synchronous product of the supervisors is nonblocking [5]. The authors also provide conditions for a maximally permissive supervisor for when a nonblocking supervisor can be computed.

Modular supervisor synthesis is also used in [6, 36]. Here, the authors consider a plant G that is composed of asynchronous subplants, meaning that the subplants do not share events. Local requirements are used, meaning that the requirements only refer to a specific part of the system. Instead of using the local requirement and the global plant to synthesize a supervisor, a local plant is used which is the parallel composition of all subplants that share events with the local requirement. Then the supremal controllable languages can be computed, which is the largest controllable sublanguage of a specification. The synthesis results in local supervisors which are nonblocking. Under the condition that the local supervisors are locally nonconflicting, the supervisors are nonblocking and maximally permissive.

In [7], a modular supervisor synthesis method is presented for extended finite state machines. It starts by checking whether the specification is controllable. When this is the case, the specification can serve as a supervisor directly. When this is not the case, the algorithm seeks suitable subsets of plants to use during synthesis. Changing variables are abstracted in a so called chaos Extended Finite State Machines, EFSMs, which includes all possible variable changes. The plants that are not included during the synthesis are replaced by chaos EFSMs. When the supervisor is controllable with respect to the abstraction of the plant and all uncontrollable events, then the algorithm is finished which results in a maximal permissive supervisor. The supervisors are not guaranteed nonblocking.

4.2 Decentralized supervisor synthesis

Decentralized supervisory control is first proposed in [8] and used in [9–12].
In [8] the authors use local requirements for which the conjunction makes the global specification. However, instead of using the corresponding parts of the plant as in modular synthesis, the authors now use a projection of the plant on the alphabet of the local requirement, resulting in local supervisors. These supervisors are then used to control the plant. Using the decentralized supervisors

result in the optimal supremal controllable language of the safe behavior. The structure is given in Figure 4.1 where the projections Q and P of plant G lead to local models G_0 and G_1 for with which supervisors f and g are computed.

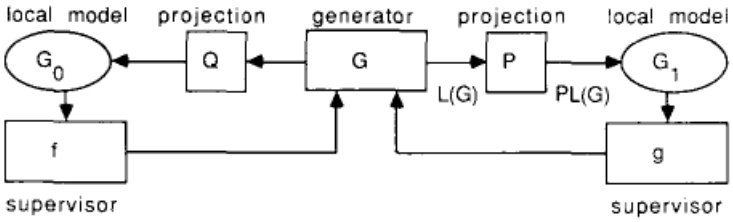


Figure 4.1: *Structure of a decentralized system.*

In [9], two situations are considered. In the first situation, the specifications are local in order to control the system locally. Meaning that, given a plant and specifications with local event sets, local supervisors can be constructed that only observe the observable local events and that only control the controllable local events. In the second situation, global specifications are considered, meaning that there is only one specification with which local supervisors can be constructed. In order to solve this, the global specification is reduced to local specifications as in the first situation by decomposing the specification. The global specification is decomposable with respect to the plant and the projections if the local versions of the specification, permit the global specification to be reconstructed. The solution is always nonblocking for the second situation, since the authors state that the conjunction of supervisors must be a proper supervisor. The solution does not guarantee maximally permissiveness, since the authors strive to find a supervisor results in at least a predefined minimally adequate behavior within the safe behavior.

[10] comes with a new approach. In previous approaches, the conditions that guarantee that the behavior is the same for a centralized and decentralized control are dependent on the specification. When a specification is changed, the conditions have to be verified again. In the new approach, the conditions are verified only once for a structure of the system after which decentralized control can be made for a set of specifications. This approach is called structural decentralized control and is shown in Figure 4.2. In the figure, the alphabets for which the supervisors are computed, are the alphabets of the corresponding requirements. The authors introduce two conditions for which, when satisfied, local syntheses do not lose optimality compared to global synthesis and subsystems do not incur blocking on the other subsystems.

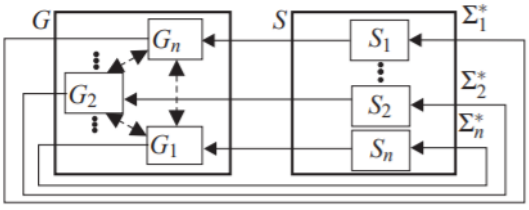


Figure 4.2: *A structural decentralized control system [15].*

4.3 Hierarchical supervisor synthesis

Hierarchical supervisory control is another method that is used to reduce the effort required to synthesize a supervisor [13–16].

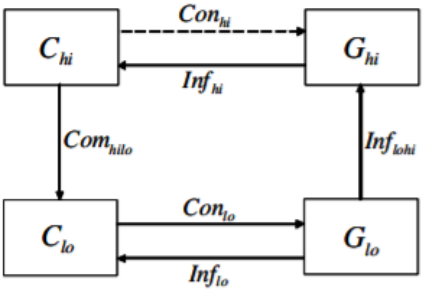


Figure 4.3: *Hierarchical control structure from [14].*

In [14], the setup is a two level hierarchy using a low level plant G_{lo} which can be perceived as the actual real world plant. This plant is controlled by the low level controller C_{lo}. G_{hi} is an abstracted simplified model of G_{lo}, $L(G_{hi}) = \theta(L(G_{lo}))$ that is used to base the decisions on that the high level controller C_{hi} makes and it gets its information from Inf_{hi}. Inf_{lohi} gives the information from the low level plant to the high level plant like state changes in G_{lo} and Inf_{lo} gives low level feedback from G_{lo} to C_{lo}. In the low level, the controller provides conventional control Con_{lo} to the plant. However the control that C_{hi} provides to G_{hi}, Con_{hi}, is only virtual, since the behavior of G_{hi} is determined by the behavior of G_{lo}. Com_{hilo} conveys the commands from the high level controller to the low level controller which translates the control in a corresponding low level control Con_{lo}. An important condition, hierarchical consistency is also introduced. This condition ensures that the task that is specified in the high level controller is actually achieved through the low level. In [13], extra conditions are made in order to preserve the nonblockingness between the levels.

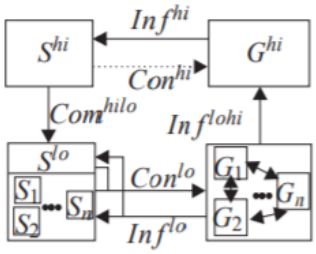


Figure 4.4: *Combination of hierarchical and decentralized structure [15].*

In [15], the authors are combining the decentralized structure from [10] where the system is modelled by the synchronous product of individual subsystems and hierarchical supervisor synthesis by making the abstraction based on the shared events. The structure is shown in Figure 4.4. This method results in a system that is hierarchically consistent, nonblocking and maximally permissive.

Another paper in which the authors combine two existing methods is [37]. In this paper, they seek to combine hierarchical control with modular supervisory control. First, all unnecessary local events are abstracted away. Next, a certain specification is picked for which all plant submodules with which it shares events are grouped. These submodules are asynchronous. For every group the synchronous product is made from all submodules, meaning that for a specification, there is a group that has one plant. For these new modules, again an abstraction is made after which a supervisor is constructed for each module and its specification. To lift the supervisors to the global level, the previously removed events are now added again. Now, the procedure moves up to the next level in which the automata representing the closed loop behavior of the modules from the previous level are used as subplants on the new level. These steps are repeated until no specifications are left. The procedure thus generates a set of nonmonolithic supervisors that are nonconflicting by construction. The result is a nonblocking system that satisfies the specifications. Maximal permissiveness is not guaranteed.

4.4 Multilevel supervisor synthesis

Another method of nonmonolithic synthesis is multilevel supervisory control. A control architecture of multilevel form is a system that consists of a network of subsystems which are divided over multiple levels of the system. Each subsystem has a parent in the higher level and has to satisfy the restrictions that are imposed from the parent. A subsystem may have children at a lower level of which it has to control the interaction. In [17], a top-down control synthesis approach is proposed. This approach is also used in [20] with three levels, but it can be extended to more levels. In Figure 4.5, the structure is shown. The plant and requirements are divided over the levels and form subsystems. The overall plant is the synchronous product of all plants. Next, a coordinated multilevel system is defined by adding the shared events of all tuples of the children to the alphabet of a parent.

In order to control the system, a set of supervisors is made with supervisors for every subsystem at each level. The synchronous product of these supervisors forms the supervisor satisfying the overall specification. The authors propose several conditions for the specification for which, when satisfied, the supervisors can be constructed, beginning at the top level and working its way down until all supervisors are constructed. In the papers, maximal permissiveness and nonblockingness are not guaranteed.

In [18], a bottom-up method is proposed. The same organisation technique is used as in [17]. However, this new method constructs supervisors for the low level groups first and uses the restricted plants from the low levels in the high level plants. The advantage of this is that the conditions of the specifications proposed in [17] are loosened. The procedure first finds low level coordinator alphabets and then computes the low level group coordinators and computes the supremal conditionally controllable sublanguage. If this is blocking, a low level coordinator is made to solve that. Next, it is repeated for higher levels. Coordinators are constructed separately to make the system not only safe, but also nonblocking. The same is repeated for the higher levels until all levels are processed. The supervisor can be imposed in a maximal permissive way if the specification satisfies an imposed condition.

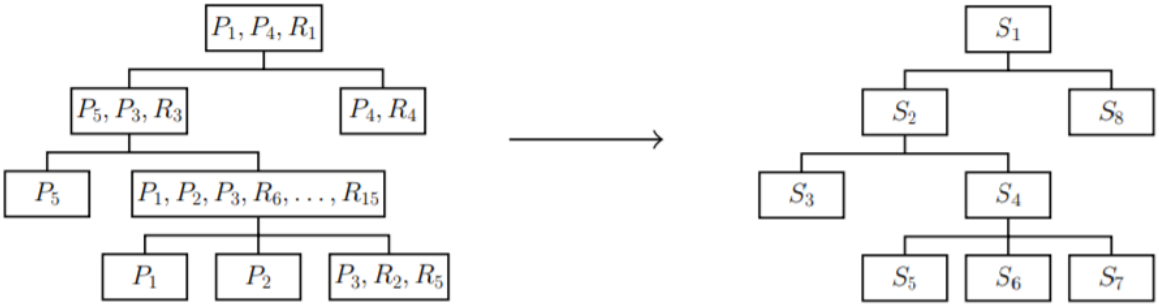


Figure 4.5: *Structure of a multilevel system [38]*

In [19], the top-down and bottom-up approaches are combined by first computing the coordinators with a top-down approach after which a posteriori supervisors and nonblocking coordinators are computed in a bottom-up manner. The combination's main advantage is the low complexity from the possibility to compute local supervisors for individual subsystems at only the lowest level from the top-down approach and the generality from the bottom-up approach that provides a safe and nonblocking solution.

4.5 Comparison

In Table 4.1, the previously discussed nonmonolithic methods and their properties are collected in an overview. In the *Type* column, the type of nonmonolithic synthesis belonging to the source in the first column is denoted. Next, it is denoted whether the method uses a division of the plant G and a division of the specifications K. Then, it is noted whether the resulting controlled system is safe, controllable, nonblocking and maximal permissive. Finally, in the last column, it is denoted whether the nonmonolithic method uses more than one level in its structure. This overview is the result of the literature research and gives a clear idea of what one can expect from using different nonmonolithic synthesis methods.

When properties are guaranteed by using a certain method, the element in the table contains a **Y**. Certain properties are in some papers not guaranteed for all systems, but only for systems that satisfy certain conditions. When this is the case, the corresponding element in the table is still filled in with a **Y**, since the paper does guarantee the property for certain systems. Only when the properties are not mentioned or when they are clearly not guaranteed an **N** is filled in.

While there are more papers and other sources that deal with nonmonolithic supervisor synthesis a selection has been made based on what methods could perhaps be used on the case models that are used later in the thesis. This means that for example papers for nonmonolithic synthesis on systems with particularly partial observability or uncertainties in the information channels are deliberately not added to the table, since the case models have full observability and no uncertainties.

Source	Type	Divide G	Divide K	Safe	Controllable	Nonblocking	Max. Permissive	More levels
[1]	Monolithic	N	N	Y	Y	Y	Y	N
[5]	Modular	Y	Y	Y	Y	Y	Y	N
[6, 36]	Modular	Y	Y	Y	Y	Y	Y	N
[7]	Modular	Y	Y	Y	Y	N	Y	N
[8]	Decentralized	Y	Y	Y	Y	N	Y	N
[9]	Decentralized	Y	Y	Y	Y	Y	N	N
[10]	Decentralized	Y	Y	Y	Y	Y	Y	N
[14]	Hierarchical	N	N	Y	Y	N	N	Y
[13]	Hierarchical	N	N	Y	Y	Y	N	Y
[15]	Hierarchical+ Decentralized	Y	Y	Y	Y	Y	Y	Y
[37]	Hierarchical+ Modular	Y	Y	Y	Y	Y	N	Y
[17, 20]	Multilevel	Y	Y	Y	Y	N	N	Y
[18]	Multilevel	Y	Y	Y	Y	Y	Y	Y
[19]	Multilevel	Y	Y	Y	Y	Y	Y	Y

Table 4.1: *Comparison of different nonmonolithic supervisory control methods.*

5 Modular supervisor synthesis in feature models

In this chapter, modular supervisor synthesis is used on feature models to reduce the required effort during supervisor synthesis. This is an already existing way to reduce effort. In order to measure and compare the effort that is required for supervisor synthesis, the monolithic as well as modular synthesis is used on both the static and dynamic model of the wiper system and the static and dynamic model of the BCS that were introduced in Section 3.2 and Section 3.3 respectively.

In the previous chapter, multiple nonmonolithic synthesis methods were listed and can be used on the wiper system and BCS. Multilevel synthesis is not a good candidate, since when a large part of the system is interconnected, a lot of the model would still be in the root node of the tree, meaning that synthesis would still need to be performed over a large part of the model. Hierarchical synthesis is also not used, because for large systems, a lot of effort would need to go to the abstraction phase for the high level before synthesis can be performed. In decentralized supervisor synthesis, the specification can be decomposed according to the plant structure. In modular synthesis, the specification does not need to be decomposable and is therefore used rather than decentralized synthesis. Modular synthesis is therefore used as a nonmonolithic method for the feature models in this thesis.

5.1 Applying modular synthesis

In this section, the procedure of applying modular synthesis on the models is explained using the static wiper system as an example. The application is however used on the dynamic model of the wiper system and both static and dynamic BCS models as well.

In CIF, supervisor synthesis can be done. However, the efficiency of the synthesis does not only depend on the method of synthesis. Event order and variable order influence the efficiency as well [39]. The event order is therefore fixed by adding a monitor automaton to every model with a self loop containing all events in the same order for every experiment. Since the event order also influences the required effort, every synthesis is done 25 times with random variable orders. Performing more experiments results in a more accurate result. However, due to the running time, the number of syntheses are limited to 25.

Since for modular synthesis, multiple supervisors are synthesized, there are multiple peak used BDD nodes and BDD operation counts. In order to compare the monolithic and modular synthesis process efforts, it is preferred to have only one peak used BDD nodes and one BDD operation count per total process. Since the peak used BDD nodes is the maximal number of nodes that is used during synthesis, the value of the modular synthesis with the highest peak used BDD nodes is also used as the peak used BDD nodes for the entire modular synthesis process. The BDD operation count, counts the amount of operations for one modular synthesis process. The BDD operation count for the entire modular synthesis process is the sum of the BDD operation counts of the individual modular processes, since it is the intention to count all BDD operations in the entire process to give a measure of time. If it was possible to run all modular synthesis processes simultaneously, then also the maximum BDD operation count could be used. However, this is practically not realistic for larger systems because of the large number of modular syntheses.

The modular synthesis method that was applied on the cases comes from [6,36] which was mentioned in Section 4.1. In this method, the supervisor synthesis is performed with a divided requirement model, meaning that there is one supervisor and synthesis process per requirement and that the plant during synthesis only consists of the parts of the plant model that are used in the requirement and all automata that share events with those parts. In order to see which automata should be used in the synthesis a dependency table is made. Next will follow an explanation of the dependency table for the static wiper system, but the same principle has been used for the dynamic wiper system and static and dynamic BCS as well.

In Figure 5.2 a dependency table is created for the static wiper system that serves as an example. The components of the wiper system are explained in Section 3.1 and Section 3.2. The model components

that are used in the table are:

- the feature plant automata *Fr-FwH*
- the algebraic booleans which represent the feature constraints *r1-r5*
- the algebraic boolean *sys_valid* and the plant automaton *Validity*
- the component automata *button-wiperHQ*
- the requirements *R1-R19*
- the presence check automaton *presence check*

The dependency table was made using the Pseudo-Algorithm 2.

Algorithm 2 Pseudo-algorithm Dependency Table

Input: Empty Dependency Table with rows x, with x=*Fr,*$\cdots$*,R19* and columns y, with y=*Fr,*$\cdots$*,presence check*.
Output: Complete Dependency Table.

1: **for all** rows x **do**
2: When a component from a row x shares events with a component on the column y, mark the element xy corresponding to that row x and column y.
3: When a component from row x has a component of column y in a guard or predicate, mark the element xy.
4: Mark element xy with $x = y$.
5: **end for**
6: **repeat**
7: **for all** rows x with at least one marked element **do**
8: **for all** marked elements xy in row x **do**
9: Find the column y corresponding to the marked element.
10: Find the row x_2 for which $x_2 = y$.
11: **for all** marked elements x_2y_2 in row x_2 **do**
12: Mark element xy_2.
13: **end for**
14: **end for**
15: **end for**
16: **until** all rows are processed without adding a new marked element.

Example 5.1. For example, $Validity$ refers to *sys_valid*, thus in step 1 in the row $Validity$, the elements corresponding to columns *sys_valid* and $Validity$ are marked. However, when repeating step 2-4, also the elements corresponding to the columns of the feature automata *Fr-FwH* and the algebraic booleans *r1-r5* are marked, since *sys_valid* refers to all algebraic booleans and they in turn refer to all feature automata.

As was previously mentioned, the modular synthesis is done by synthesizing a supervisor per requirement and the corresponding plant components as described in [36]. To find the plant components that should be used during synthesis first find the row that belongs to the requirement for which synthesis will be done. In that row, find the components on the columns corresponding to the marked elements. These components together is the asynchronous system that corresponds to the requirement and is used as the plant during synthesis.

In the way the presence check automata are defined in the original model, all events and their corresponding feature automata are present in the presence check automata. This means that almost all components are linked through the presence check automaton and that all those components should be part of the plant during modular synthesis. Fortunately, the presence check automata can also be defined where there is one presence check automaton for every component automaton. In

Figure 5.1, an example of the division of one presence check automaton into three presence check automata is shown. This solves the problem that every component is linked through the presence check automaton.

```
plant automaton PRESENCE_UNCONTROLLED :
    location:
        initial; marked;
        edge button.u_permOn when Fp.present ;
        edge sensorLQ.u_noRain when FsL.present ;
        edge sensorLQ.u_rain when FsL.present ;
        edge sensorHQ.u_noRain when FsH.present ;
        edge sensorHQ.u_littleRain when FsH.present ;
        edge sensorHQ.u_heavyRain when FsH.present ;
end
```

(a) *Original presence check automaton.*

```
plant automaton PRESENCE_UNCONTROLLED1 :
    location:
        initial ; marked ;
        edge button.u_permOn when Fp.present ;
end

plant automaton PRESENCE_UNCONTROLLED2 :
    location:
        initial ; marked ;
        edge sensorLQ.u_noRain when FsL.present ;
        edge sensorLQ.u_rain when FsL.present ;
end

plant automaton PRESENCE_UNCONTROLLED3 :
    location:
        initial ; marked ;
        edge sensorHQ.u_noRain when FsH.present ;
        edge sensorHQ.u_littleRain when FsH.present ;
        edge sensorHQ.u_heavyRain when FsH.present ;
end
```

(b) *Divided presence check automata.*

Figure 5.1: *The division of a presence check automaton.*

Since there are no cross references from and to requirements in this case, the requirements are not placed in the columns of the dependency table for readability. Also, the presence check automaton has been left out of the row components, since it would have an entirely marked row and eventually it has been divided as was mentioned above. However, it is still used in the dependency table in the column components as a reminder that the presence check automata of the corresponding components have been added to the plant during synthesis. The divided presence check automata could have been added separately, but due to readability problems, this has been omitted.

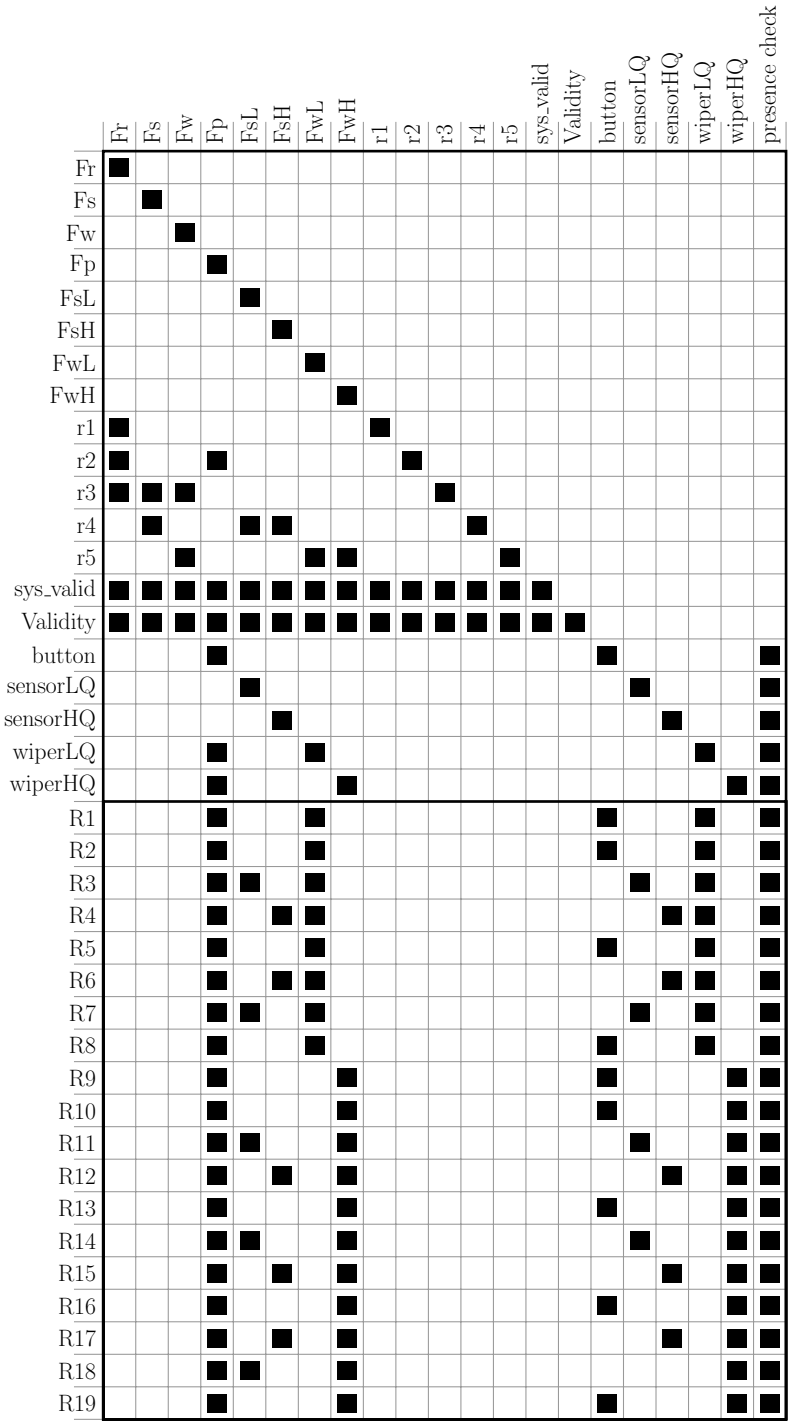


Figure 5.2: *Dependency table of the wiper system.*

5.2 Results

The effort that is required for the syntheses for the models are displayed in Figure 5.3 for the wiper models and Figure 5.4 for the BCS. Compared to the monolithic synthesis method there is a large reduction in peak used BDD nodes when modular synthesis is used. The decrease in BDD operation count is relatively not as large as the decrease in peak used BDD nodes. For the static and dynamic wiper model, the BDD operation count has even increased using modular synthesis compared to the monolithic results. This is due to the multiple syntheses per model since the BDD operation count

is summed to compute the BDD operation count for the synthesis process of the entire model. The BDD operation count that is added because of performing more than one synthesis can counter the BDD operation count won by using a modular method.

Using a modular supervisor synthesis can thus reduce the effort that is required for synthesis. Especially the peak used BDD nodes can decrease significantly. Though it seems that the peak used BDD nodes always decreases, the BDD operation count might increase for some systems. It would therefore be interesting to find a way in which a decrease of peak used BDD nodes could be created in combination with a significant BDD operation count decrease.

Figure 5.3: *Monolithic versus modular effort plot of the wiper system.*

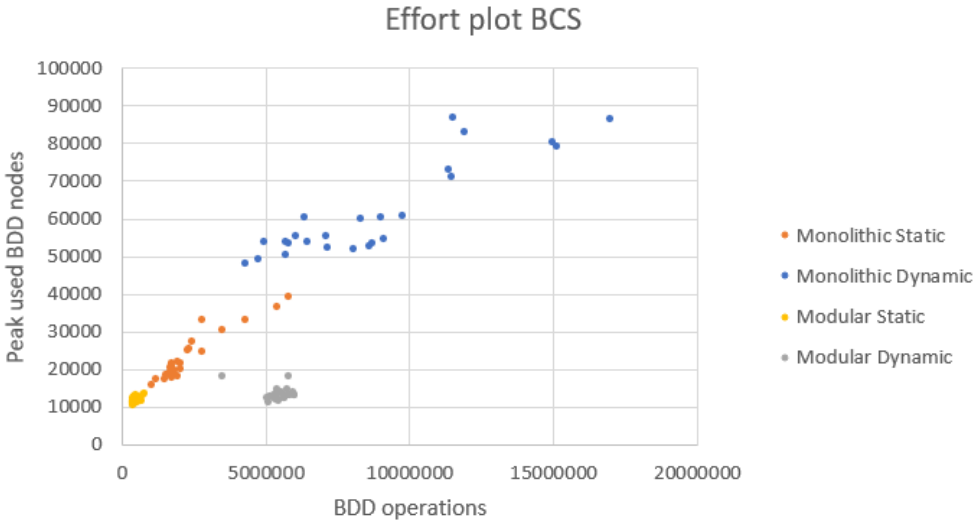


Figure 5.4: *Monolithic versus modular effort plot of the BCS.*

6 Removing feature model from synthesis

In the previous chapter while using modular supervisor synthesis, the effort was greatly reduced by only using the parts of the plant that had shared events with the components that were used in the requirement or that were referred to in the components from the requirement. By doing so, the feature model itself was deconstructed. All feature constraints were removed from the plant and only a selection of features was used. In this chapter, it is explained that this is allowed and a new method is created.

The feature model influences the supervisor synthesis process described in Section 2.2 and therefore the system by setting initial states and blocking events with the presence check automata. The initial states that are partly set by the feature model are only used in the last step of the algorithm in the definition of the initial states of the supervisor. Now assume that the initial states coming from the feature model are not used in the conjunction that constructs the initial states for the supervisor by for example leaving out the feature model in the plant during synthesis. This means that the initial states for the supervisor are the conjunction of the initial states set by the component plant automata and the safe nonblocking states. However, when this supervisor is applied in the model with the original plant, meaning that the feature model is included, the initial states of the controlled plant are the conjunction of the safe and nonblocking states, initial states of the component automata and the initial states set by the feature model, which is the same as it was when using the feature model in the synthesis.

When the features and presence check automata are removed, the disablement of certain events by the presence check is also removed. However, when the supervisor is used on the original plant again, these edges are again blocked within the safe state space of the system. Leaving out features during the synthesis process thus does not effect the safety of the system and does not create an unsafe controlled system when the supervisor is used on the original system. The consequences that removing features and presence check automata have on nonblocking, controllability and maximal permissiveness are discussed later in the thesis.

6.1 Removing the entire feature model

For now assuming that there are no references to the feature model in the requirements and keeping the previous section in mind, the entire feature model has thus no essential contribution to the supervisor synthesis. This is discussed in the next section. The feature model can thus be removed entirely from the plant that is used for synthesis, this means removing feature automata as well as presence check automata, feature constraints and the $Validity$ automaton.

When a feature is referred to in a requirement, then only that particular feature is kept in the plant. Even though a feature is used and present in the model, the presence check automata using that feature can still be removed, for the same reasons as described above.

This means that a new method, Feature Model Removal (FMR), for reducing the effort for supervisor synthesis for feature models has been created for both monolithic and modular synthesis which is as follows:

1. First remove the feature model from the model. This means that feature plant automata, the algebraic booleans representing the feature constraints, the *Validity* automaton and the presence check automata are removed. When a certain feature is referred to in for example a predicate of a requirement or a guard of a plant component, that particular feature plant automaton is not removed.
2. Next, perform synthesis using the requirements and the plant resulting from the previous step.
3. Finally, for the controlled system, use the original plant including the feature model and the supervisor resulting from step 2.

In Figure 6.1, the original synthesis process is shown. The plant and specifications are used to make a supervisor which controls the plant. In the FMR method, the red arrow is (partly) removed from the process, so the supervisor is synthesized with only a part of the plant and the specifications, but controls the entire plant.

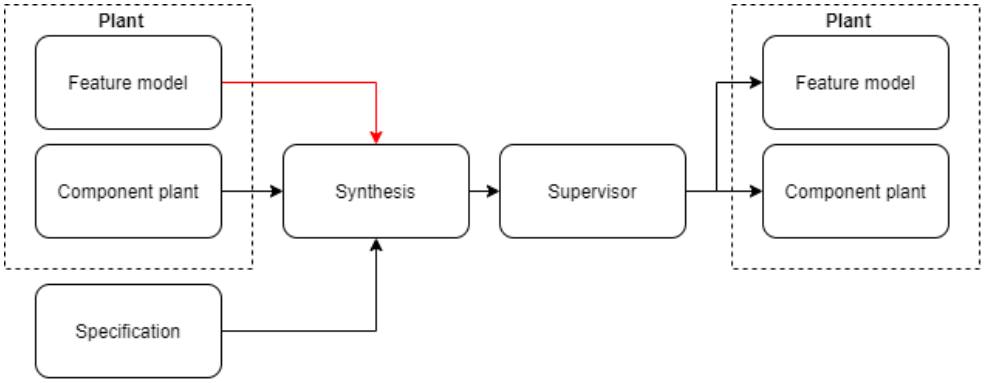


Figure 6.1: *Original and FMR synthesis process.*

6.2 Consequences of removing the feature model

Using the FMR, the plant that is combined with the supervisor in the controlled system is not the same plant that was used to make the supervisor with. This means that the controlled system does not necessarily adopt the controllable, nonblocking and maximal permissive properties of the controlled system that was made using regular supervisor synthesis. In this section, the consequences of the FMR on these properties are discussed as well as an extra discussion about safety for static feature configurations, controllable dynamic feature configurations and uncontrollable dynamic feature configurations.

6.2.1 Consequences for static configurations

When for models with static configurations the feature model is removed from the plant that is used to compute the supervisor, the supervisor is made for a plant in which all behavior of the component automata is possible. The result of the synthesis is a safe state space for all component automata. This is the same as the situation in which all features are present. When the supervisor is used in combination with the original plant including the feature model, this safe state space is used in a conjunction with every initial state introduced by the feature model. The presence check automata from the original plant can disable edges within this safe state space, but since the resulting state space is still a part of the safe state space, the controlled system is always safe.

The controlled system is also controllable since the algorithm in CIF makes sure that synthesized supervisors in CIF do not disable uncontrollable events. The feature model or its absence thus does not influence the controllability of the system.

In the next example, it is shown that using the FMR does not guarantee nonblockingness.

Example 6.1. Take the system from Figure 6.2. The presence check automaton states that b is only allowed when feature F is present and a requirement states that a needs location *B.2*. When synthesis is performed on the plant including the feature model, the supervisor demands the initial state *F.present*. However, when synthesis is performed without the feature model, this initial state is not required in the supervisor. That means that when the supervisor is used in combination with the original plant, that F can also not be present in which case b can not be executed, *B.2* can not be reached and a can not be executed and automaton A stays in the blocking location *A.1*.

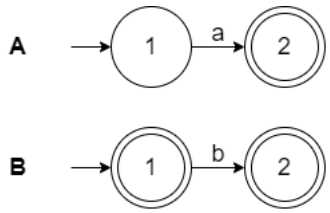


Figure 6.2: *Example 6.1.*

Condition 1. All automata have marked initial locations and that they do not share events.

Condition 1 means that when a feature corresponding with a certain component automaton is not present, all events in that automaton are disabled and the automaton stays in its marked initial location. This is however not enough to make the controlled system nonblocking as is shown in the following example.

Example 6.2. In the system in Figure 6.3, the initial locations are marked and there are no shared events. The requirement states that b needs $B.2$ and the presence check states that c and d are only enabled when feature F is present. The original supervisor states that $F.present$ is an initial state and that d is enabled when $A.3$. However, the supervisor when leaving the feature model out of the synthesis does not command this initial condition $F.present$. This means that when the supervisor is used on the original plant, when F is not present, c and d are not enabled, meaning that b is not enabled and that $A.2$ is a blocking state.

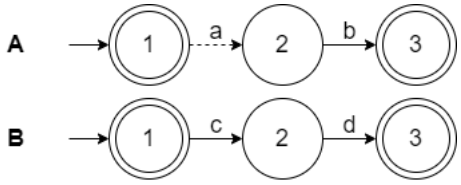


Figure 6.3: *Example 6.2.*

Condition 2. The availability of an edge from an unmarked to a marked state can not depend on a non-initial state in an automaton that corresponds to a feature that can be absent.

Condition 2 means that the availability of the edge only depends on component automata and not on the feature model. For a system plant satisfying the Conditions 1 and 2, the controlled system can not become blocking due to the feature model when the original controlled system is nonblocking.

The supervisor made with FMR applied to the original plant does not guarantee maximal permissiveness. This is shown in the following example.

Example 6.3. Imagine a system with two automata A and B as in Figure 6.4. In this system there is also a feature F. Event c is only enabled when F is present. There is one requirement invariant `not (A.3 and B.2)`. b and c are uncontrollable.

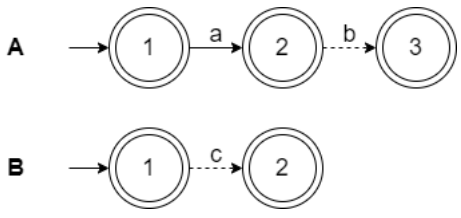


Figure 6.4: *Example 6.3.*

When synthesis is performed on this system including the feature model, the supervisor enables a when `B.1 and not F.present`. However, when the feature model is removed from the system as is

done in the FMR, the supervisor never enables a at all, since c can always be executed and after a, b can always be executed. This leads to $A.3$ and $B.2$ which was forbidden by the requirement. a is now always disabled instead contrary to the supervisor computed without FMR. This shows that using the method does not guarantee a maximal permissive controlled system.

Condition 3. All events are controllable in the automata that are used in the requirements and the automata that share events with those automata. Also, all events that lead to an unmarked state are controllable.

The events from the automata used in requirements and the automata that share events with them could be controlled by the supervisor to make the system safe. The events that lead to an unmarked state could be controlled by the supervisor to make the system nonblocking. If Condition 3 holds, the supervisor can control every edge that is relevant for the supervisor to make the system safe and nonblocking, based on the component automata regardless of the feature model. The supervisor synthesized by including and excluding the feature model in the plant during synthesis will thus restrict the same events. The only thing that the feature model does when the supervisor is used on the original plant is making the states corresponding to those edges reachable or not. That means that the controlled system is guaranteed maximally permissive with Condition 3 for models for which the controlled system is safe, controllable and nonblocking.

6.2.2 Consequences for uncontrollable dynamic configuration

For systems in which the configuration is dynamic with uncontrollable *come* and *go* events in the feature automata that were introduced in Section 3.1, the consequences of using the FMR are different compared to the static systems. The first difference between static and dynamic configurations is the possibility of a reset edge, which for example brings an automaton to its initial state when the corresponding feature leaves the system, when a *come* or *go* event is used in a component automaton. This can also happen in a controllable dynamic configuration. The features corresponding to those events are not removed from the plant that is used during synthesis. The resets are thus incorporated in the safe state space resulting from the synthesis. Further, as was the case in the static configuration systems, when the feature model is removed, the plant that is used during synthesis results in the same behavior as in the case that all features are present. The addition of the feature model after synthesis then again only results in the removal of edges in the safe state space when certain events are disabled due to the absence of a feature. Since the *come* and *go* events are uncontrollable, features can come and go at random. Therefore corresponding edges can be enabled and disabled at random as well, meaning that the absence of a feature only results in a hold up in the current locations in its corresponding component automata. Since it can resume its execution of edges when the feature is made present again, the state space of the component automata does not change. The controlled safe state space of the component automata therefore is the same when using original synthesis and using the FMR. This also means that the controlled system is safe.

Since the controlled state space for the component automata is the same for original synthesis and using the FMR, when the original controlled system is nonblocking, the new controlled system is also nonblocking and maximal permissiveness and controllability is guaranteed.

6.2.3 Consequences for controllable dynamic configuration

Systems can also have a dynamic configuration in which the *come* and *go* events are controllable. For such systems, using FMR does not necessarily result in a nonempty supervisor when the original synthesis does result in a nonempty supervisor. This is shown in the following example.

Example 6.4. Consider the example in Figure 6.5. The presence check automaton states that b can only happen when F is present and there is a requirement `requirement not A.3`. When the supervisor is made including the feature model in the plant, the supervisor states that *come* and *go*

are disabled and that the initial state is *F.present*. However, when the feature model is omitted from the synthesis, the synthesis results in an empty supervisor which means that the supervisor has to disable so much of the behavior that no initial condition remains in the state space. The controlled system is therefore empty.

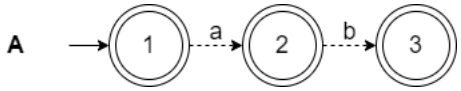


Figure 6.5: *Example of a potentially unsafe system.*

The problem is that the situation might be that only an uncontrollable string of events can lead to a bad state, in the example state 3, from the initial state of a component automaton in a system without a feature model. Since the supervisor can not control uncontrollable events, there is no valid initial state, which means that the controlled system is empty. The feature model however can disable certain uncontrollable events through the presence check automata with the absence of features, meaning that certain features should be included in the plant during synthesis. The FMR should therefore be adjusted for systems with controllable dynamic configurations. When a component automaton only has controllable events, then the supervisor can control all edges of that automaton by enabling or disabling the events from the component automaton without necessarily having to use the corresponding presence check and thus the *come* and *go* events. The feature and presence check can thus be removed. The following adaptation of the FMR can be used.

1. When a component plant automaton only has controllable events, then the corresponding presence check automaton is removed from the plant. When all plant automata corresponding to a certain feature only have controllable events, then also the feature plant automaton is removed from the plant.
   When a certain feature is referred to in for example a predicate of a requirement, a guard of a plant component or it shares an event with a component automaton, that particular feature plant automaton is not removed.

2. Next, perform synthesis using the requirements and the plant resulting from the previous step.

3. Finally, for the controlled system, use the original plant including the feature model and the supervisor resulting from step 2.

Now, there is only a feature model removal in the plant parts belonging to component automata of which is certain that the supervisor can control the behavior only by using component automaton events. For those parts, the addition of the feature model after synthesis will again act as a "pause" in the behavior of the automaton as in the uncontrollable dynamic configuration systems. Since the changed part of the system acts like the uncontrollable dynamic configuration system and the rest is unchanged, the controlled system is again safe, controllable and maximal permissive and it is nonblocking when the system after regular synthesis is nonblocking as well.

In Table 6.1 a summary is given for each type of configuration and which properties of the system after original synthesis without FMR is adopted by the system after synthesis with FMR.

	Safe	**Controllable**	**Nonblocking**	**Maximally Permissive**
Static	**Y**	**Y**	**N** **Y, for cond. 1&2**	**N** **Y, for cond. 3**
Uncontr. Dyn.	**Y**	**Y**	**Y**	**Y**
Contr. Dyn.	**N** **Y, for adapted FMR**	**Y**	**N** **Y, for adapted FMR**	**N** **Y, for adapted FMR**

Table 6.1: *Summary of the three configurations and whether they adopt the properties of the original synthesis when using FMR.*

6.3 Effect on the effort

In order to measure the effects on the required effort, the method has been applied on the static and dynamic models of the wiper system and the BCS. The effect on monolithic supervisor synthesis is used as a measure of how the FMR reduces the required effort on its own. The FMR and modular supervisor synthesis are also combined to see whether the FMR can reduce the required effort even though the effort has already been reduced by using modular synthesis.

In Figure 6.6 the required effort for the different synthesis methods are shown for the static wiper system. The average value of the peak used BDD nodes and the BDD operation count have decreased 31% and 47% respectively when applying the FMR on the monolithic supervisor synthesis. Comparing that to the modular method, the reduction of the peak used BDD nodes is far greater with using the modular method. However, the BDD operation count for the modular synthesis has significantly increased compared to the monolithic synthesis. Using the FMR together with the modular synthesis method results in the significant peak used BDD nodes decrease thanks to the modular synthesis and the BDD operation count increase has been compensated thanks to the FMR, which shows the relevant contribution of the FMR.

Figure 6.6: *Effort plot for supervisor synthesis on the static wiper system.*

In Figure 6.7 the effort plot is shown for the uncontrollable dynamic wiper system. Using the FMR during monolithic supervisor synthesis results in a decrease of 32% and 41% of the average values of the peak used BDD nodes and BDD operation count respectively. When using only modular supervisor synthesis, the peak used BDD nodes are decreased significantly again. The effort decrease thanks to the modular synthesis in BDD operation count does not outway the effort increase due to the increase of the number of syntheses, resulting in a net increase of effort. For modular synthesis, there is a trade-off between space and time effort. The modular synthesis in combination with FMR results in a smaller peak used BDD nodes than the monolithic synthesis in combination with FMR, while the BDD operation count is better for the monolithic version with FMR instead of the modular synthesis with FMR. The trade-off is thus apparent in the results. Adding the FMR to the modular synthesis reduces the BDD operation count while profiting from the decrease of peak used BDD nodes thanks to the modular synthesis.

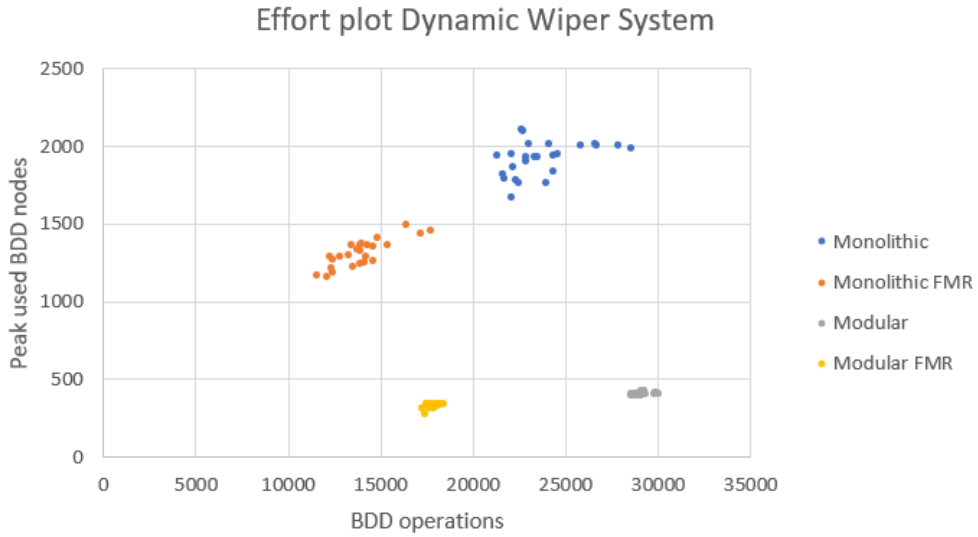


Figure 6.7: *Effort plot for supervisor synthesis on the dynamic wiper system.*

The results for the experiments using the static BCS system are plotted in Figure 6.8. This model is bigger than the wiper system and also has a larger feature model. Removing this from the model results in a decrease of the average value of both effort metrics with a decrease of the BDD operation count of no less than 79%. While modular synthesis also results in a large reduction of required effort, the addition of the FMR halves the BDD operation count further and decreases the peak used BDD nodes with a third. Comparing the original monolithic supervisor synthesis effort to the effort required by the modular supervisor synthesis with the FMR, there is a decrease of 96% and 96% of the average values of the peak used BDD nodes and BDD operation count respectively.

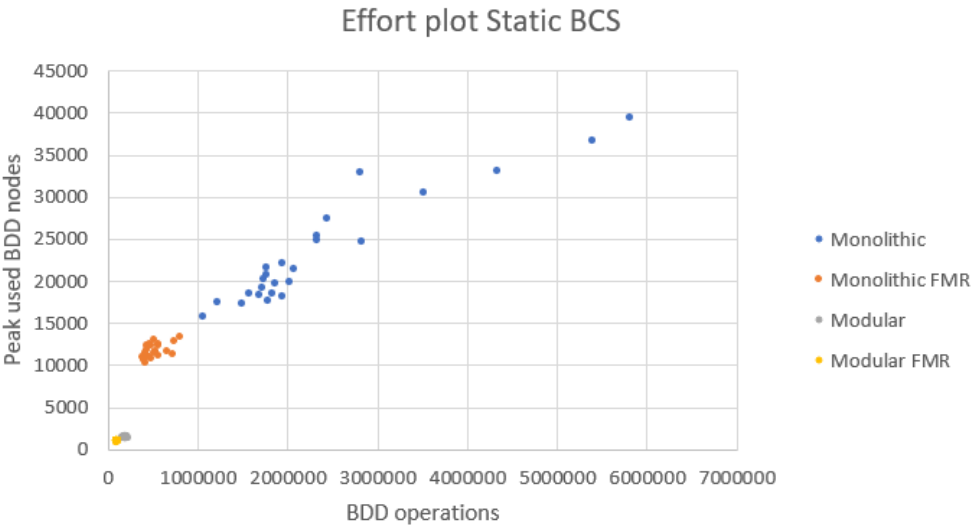


Figure 6.8: *Effort plot for supervisor synthesis on the static BCS.*

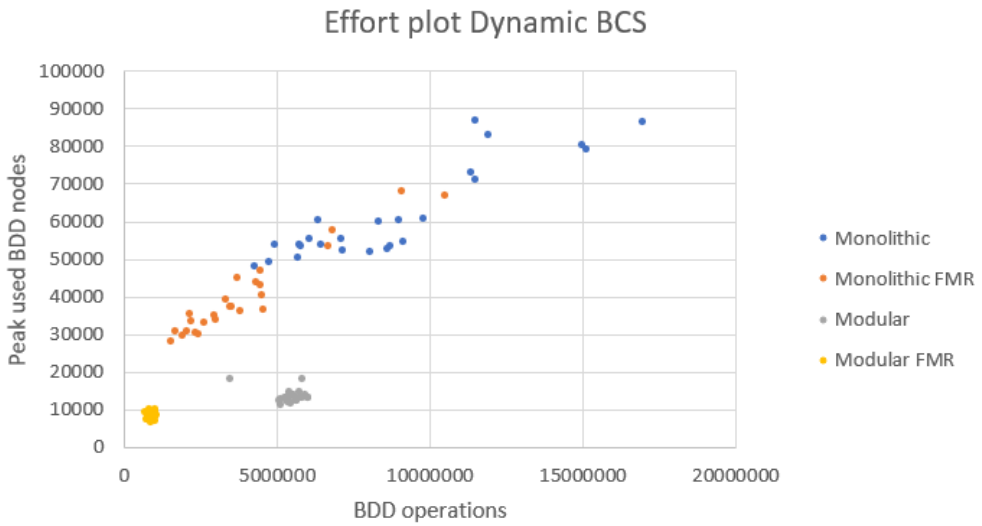


Figure 6.9: *Effort plot for supervisor synthesis on the dynamic BCS.*

The same conclusions can be read from the effort plot of the uncontrollable dynamic BCS in Figure 6.9. In this particular model the 54 modular syntheses using the FMR for the requirements that are also in the static model are good for just a small part of the total BDD operation count of the entire modular synthesis. Some of the added requirements to make the model dynamic refer to *sys_valid* which means that all algebraic booleans representing the feature constraints and therefore all feature plant automata can not be removed from the model. The last five syntheses, corresponding to the last five requirements are thus good for 90% of the BDD operation count for the entire modular synthesis using FMR. In models where references to the entire feature model can be avoided, both metrics can be reduced even more.

The results show that the FMR does reduce the effort significantly for both monolithic and modular supervisor synthesis. Though using modular supervisor synthesis can reduce the effort more than the FMR during the monolithic supervisor synthesis, adding the FMR to the modular supervisor synthesis does decrease the effort even more. In the case of modular synthesis of the dynamic wiper system, the BDD operation count was still more than the BDD operation count of the monolithic synthesis. In modular synthesis, there can be a trade-off between space and time effort. Still, applying FMR reduces the effort in all cases. The FMR thus proves to be a useful tool to reduce the effort on its own and in combination with modular synthesis.

Since the FMR is a useful tool in combination with modular synthesis, it is interesting to see whether the efficiency increase is also reached in combination with other nonmonolithic synthesis methods like multilevel or hierarchical methods. Since the FMR is now only tested on two cases, it might be useful to develop more cases for feature models of different sizes or configurations. There is for example not yet a case in which the dynamic configuration is controllable or where there is perhaps a mix of features with controllable and uncontrollable features.

7 Conclusions and recommendations

During monolithic supervisor synthesis, one supervisor is made to control the entire system. For large systems, this requires a lot of effort and the computation of a supervisor might become infeasible. In order to reduce the required effort, several methods can be used in which control is divided over multiple supervisors. Modular and decentralized supervisor synthesis divide the plant in modules that share events with the specifications. Hierarchical supervisor synthesis creates a higher level of the model which is abstracted for which high level supervisors are made that translates in the low level command. Multilevel supervisor synthesis divides the plant into a tree-structure for which supervisors are made per level or groups. The properties of the different methods are compared in a clear overview.

In order to reduce the required effort to synthesize a supervisor for feature models, first an existing method for modular supervisor synthesis was used on a small wiper system model and a large body comfort system model. Both static and dynamic configurations were used. In this method, the plant and requirements were divided into subsystems and synthesis was performed on those systems. In order to see more clearly which plant components should be used in the plant during synthesis for a specific requirement, dependency tables were created. The results showed that using modular synthesis greatly reduces the peak used BDD nodes compared to monolithic synthesis, but is not always effective for the BDD operation count due to the increased number of syntheses.

A new method to reduce the required effort for supervisor synthesis in feature models was introduced as Feature Model Removal, in which certain parts of the feature model in the plant are removed from the plant that is used during synthesis. The original plant is then used in combination with the synthesized supervisor which results in the controlled system.

This FMR method has some consequences. For feature models with a static configuration, the controlled system is safe and controllable, but only for systems with certain conditions can maximal permissiveness and nonblockingness be guaranteed.

For systems with uncontrollable dynamic configurations, the controlled state space of the component automata is the same for the original controlled system and the system using FMR. The safe, controllable, maximal permissive and nonblocking properties of the original system are therefore kept when using FMR.

For models with controllable dynamic configurations, FMR does not always work, which means that the method is altered such that the controlled system keeps the safe, controllable, maximal permissive and nonblocking properties of the original system.

Comparing the required effort for synthesis for both static and dynamic models of the wiper system and the BCS using monolithic and modular synthesis, both with and without FMR, shows that FMR reduces the effort significantly. Both on its own and in combination with modular synthesis, the FMR method is a useful tool to reduce synthesis effort for feature models.

For further research, it is interesting to find out whether the efficiency win of the FMR in combination with modular synthesis can also be won in combination with other nonmonolithic synthesis methods. Also, more cases should be developed, since there is a limited number of cases at the time of writing.

References

[1] C. Cassandras and S. Lafortune, *Discrete Event Systems Second Edition*. Springer, 2 ed., 2010.

[2] S. Miremadi, B. Lennartson, and K. Åkesson, "A BDD-based approach for modeling plant and supervisor by extended finite automata," *IEEE Transactions on Control Systems Technology*, vol. 20, no. 6, pp. 1421–1435, 2012.

[3] S. B. Akers, "Binary Decision Diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509–516, 1978.

[4] S. Thuijsman, D. Hendriks, R. Theunissen, M. Reniers, and R. Schiffelers, "Computational effort of BDD-based supervisor synthesis of extended finite automata," *IEEE International Conference on Automation Science and Engineering*, vol. 2019, pp. 486–493, 2019.

[5] W. M. Wonham and P. J. Ramadge, "A modular supervisory control design of discrete event systems," *Mathematics of Control, Signals and Systems*, vol. 1, pp. 13–30, 1988.

[6] M. H. Queiroz and J. E. R. Cury, "Modular Control of Composed Systems," *Proceedings Of The American Control Conference*, no. June, pp. 4051–4055, 2000.

[7] R. Malik and M. Teixeira, "Modular supervisor synthesis for extended finite-state machines subject to controllability," *2016 13th International Workshop on Discrete Event Systems, WODES 2016*, pp. 91–96, 2016.

[8] F. Lin and W. M. Wonham, "Decentralized supervisory control of discrete-event systems," *Information Sciences*, vol. 44, no. 3, pp. 199–224, 1988.

[9] K. Rudie and W. M. Wonham, "Think Globally, Act Locally: Decentralized Supervisory Control," *IEEE Transactions on Automatic Control*, vol. 37, no. 11, pp. 1692–1708, 1992.

[10] S.-H. Lee and K. C. Wong, "Structural Decentralised Control of Concurrent Discrete-event Systems," *European Journal of Control*, vol. 8, no. 5, pp. 477–491, 2002.

[11] T. Yoo and S. Lafortune, "A general architecture for decentralized supervisory control of Discrete Event Systems," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 12, pp. 335–377, 2002.

[12] W. M. Wonham and K. Cai, "Decentralized and distributed supervision of discrete-event systems," *Communications and Control Engineering*, pp. 147–203, 2019.

[13] K. C. Wong and W. M. Wonham, "Hierarchical control of discrete-event systems," *Discrete Event Dynamic Systems: Theory and Applications*, vol. 6, no. 3, pp. 241–273, 1996.

[14] H. Zhong and W. M. Wonham, "On the Consistency of Hierarchical Supervision in Discrete-event Systems," *IEEE Transactions on Automatic Control*, vol. 15, no. 10, 1990.

[15] K. Schmidt, J. Reger, and T. Moor, "Hierarchical control for structural decentralized DES," *IFAC Proceedings Volumes (IFAC-PapersOnline)*, vol. 37, no. 18, pp. 279–284, 2004.

[16] M. Z. Fekri and S. Hashtrudi-Zad, "Hierarchical robust supervisory control of discrete-event systems," *Proceedings of the American Control Conference*, pp. 1178–1183, 2008.

[17] J. Komenda, T. Masopust, and J. H. Van Schuppen, "Coordination control of distributed discrete-event systems," *Lecture Notes in Control and Information Sciences*, vol. 433, pp. 147–167, 2013.

[18] J. Komenda, T. Masopust, and J. H. Van Schuppen, "Bottom-up approach to multilevel supervisory control with coordination," *2014 European Control Conference, ECC 2014*, pp. 2715–2720, 2014.

[19] J. Komenda, T. Masopust, and J. H. Van Schuppen, "Multilevel coordination control of modular DES," *IEEE Conference on Decision and Control*, no. 52, pp. 384–389, 2013.

[20] J. Komenda, T. Masopust, and J. H. V. Schuppen, "Control of an Engineering-Structured Multi-level Discrete-Event System," *Proceedings of the 13th International Worshop on Discrete Event Systems*, pp. 103–108, 2016.

[21] W. M. Wonham, *Supervisory Control of Discrete-Event Systems*, vol. 8810. Kai Wong, 2010.

[22] 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*, vol. 8, no. 3, pp. 560–569, 2011.

[23] S. Thuijsman, M. Reniers, and D. Hendriks, "Efficiently enforcing mutual state exclusion requirements in symbolic supervisor synthesis," *2021 IEEE 17th International Conference on Automation Science and Engineering*, pp. 777–783, In press-August 2021.

[24] 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 and algorithms for the construction and analysis of systems : 20th International Conference*, vol. 8413, pp. 575–580, 2014.

[25] E. Randal, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams," *ACM Computing Surveys*, vol. 24, no. 3, pp. 293–318, 1992.

[26] T. van Dijk and J. van de Pol, "Sylvan: multi-core framework for decision diagrams," *International Journal on Software Tools for Technology Transfer*, vol. 19, no. 6, pp. 675–696, 2017.

[27] Y. K. Lopes, S. M. Trenkwalder, A. B. Leal, T. J. Dodd, and R. Groß, "Supervisory control theory applied to swarm robotics," *Swarm Intelligence*, vol. 10, no. 1, pp. 65–97, 2016.

[28] M. Reniers and J. Van de Mortel-Fronczak, *Supervisory Control*. Lecture notes 4CM30 Supervisory Control Theory, Eindhoven University of Technology, February 4, 2019.

[29] D. Benavides, P. Trinidad, and A. Ruiz-Cortés, "Automated reasoning on feature models," *Lecture Notes in Computer Science*, vol. 3520, no. January, pp. 491–503, 2005.

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

[31] J. Dervaux, P. Cormier, P. Moskovkin, O. Douheret, S. Konstantinidis, R. Lazzaroni, S. Lucas, and R. Snyders, "Modelling with FTS: A Collection of Illustrative Examples," tech. rep., University of Namur, 2010.

[32] M. H. ter Beek, M. A. Reniers, and E. P. de Vink, "Supervisory controller synthesis for product lines using CIF 3," *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, vol. 9952 LNCS, pp. 856–873, 2016.

[33] M. Reniers and S. Thuijsman, "Supervisory control for dynamic feature configuration in product lines," in *2020 Forum for Specification and Design Languages (FDL)*, pp. 1–8, 2020.

[34] M. Tuitert, "Supervisory controller synthesis for Dynamic Software Product Lines," Master's thesis, Eindhoven University of Technology, 2017.

[35] S. Lity, R. Lachmann, M. Lochau, and I. Schaefer, "Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study," Tech. Rep. 2012-07, Technische Universität Braunschweig, 2017.

[36] M. H. De Queiroz and J. E. Cury, "Modular Supervisory Control of Large Scale Discrete Event Systems," *Discrete Event Systems: Analysis and Control*, pp. 103–110, 2000.

[37] R. C. Hill and P. M. Tilbury, "Modular supervisory control of discrete-event systems with abstraction and incremental hierarchical construction," *Proceedings - Eighth International Workshop on Discrete Event Systems 2006*, pp. 399–406, 2006.

[38] M. Goorden, J. v. d. Mortel-Fronczak, M. Reniers, W. Fokkink, and J. Rooda, "Structuring multilevel discrete-event systems with dependence structure matrices," *IEEE Transactions on Automatic Control*, vol. 65, no. 4, pp. 1625–1639, 2020.

[39] S. Thuijsman, D. Hendriks, R. Theunissen, M. Reniers, and R. Schiffelers, "Computational effort of BDD-based supervisor synthesis of extended finite automata," *IEEE International Conference on Automation Science and Engineering*, pp. 486–493, 2019.

Appendix

Static Wiper System

Listing 1: *First part of the feature model*

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

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

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

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

Listing 2: *Component automata*

```
plant automaton button :
uncontrollable u_off , u_on , u_permOn ;
    location Off :
        initial ; marked ;
        edge u_permOn goto PermOn ;
        edge u_on goto On ;
    location On :
        edge u_permOn goto PermOn ;
        edge u_off goto Off ;
    location PermOn :
        edge u_off goto Off ;
        edge u_on goto On ;
end

plant automaton sensorLQ :
uncontrollable u_noRain , u_rain ;
    location NoRain :
        initial;marked ;
        edge u_rain goto Rain ;
    location Rain :
        edge u_noRain goto NoRain ;
end

plant automaton sensorHQ :
uncontrollable u_noRain  , u_littleRain , u_heavyRain ;
    location NoRain :
        initial ;marked ;
        edge u_littleRain goto LittleRain ;
        edge u_heavyRain goto HeavyRain ;
    location LittleRain :
        edge u_noRain goto NoRain ;
        edge u_heavyRain goto HeavyRain ;
```

```
    location HeavyRain :
        edge u_noRain goto NoRain ;
        edge u_littleRain goto LittleRain ;
end

plant wiperLQ :
controllable c_off , c_noRain , c_rain , c_permWipe ;
    location NoRain :
        initial ; marked ;
        edge c_rain goto Rain ;
        edge c_permWipe goto PermWipe ;
    location PermWipe :
        edge c_off goto NoRain ;
        edge c_rain goto Rain ;
    location Rain :
        edge c_noRain goto NoRain ;
        edge c_off goto NoRain ;
        edge c_permWipe goto PermWipe ;
end

plant wiperHQ :
controllable c_off , c_noRain , c_littleRain , c_heavyRain , c_permWipe ;
    location NoRain :
        marked ; initial ;
        edge c_permWipe goto PermWipe ;
        edge c_littleRain goto LittleRain ;
        edge c_heavyRain goto HeavyRain ;
    location PermWipe :
        edge c_off goto NoRain ;
        edge c_littleRain goto LittleRain ;
        edge c_heavyRain goto HeavyRain ;
    location LittleRain :
        edge c_noRain goto NoRain ;
        edge c_heavyRain goto HeavyRain ;
        edge c_off goto NoRain ;
        edge c_permWipe goto PermWipe ;
    location HeavyRain :
        edge c_noRain goto NoRain ;
        edge c_littleRain goto LittleRain ;
        edge c_off goto NoRain ;
        edge c_permWipe goto PermWipe ;
end
```

Listing 3: *Second part of the feature model*

```
plant automaton PRESENCE_CONTROLLED :
    location  :
        initial ; marked ;
        edge wiperLQ.c_off when FwL.present ;
        edge wiperLQ.c_noRain when FwL.present ;
        edge wiperLQ.c_rain when FwL.present ;
        edge wiperLQ.c_permWipe when Fp.present and FwL.present ;
        edge wiperHQ.c_off when FwH.present ;
        edge wiperHQ.c_noRain when FwH.present ;
        edge wiperHQ.c_littleRain when FwH.present ;
        edge wiperHQ.c_heavyRain when FwH.present ;
        edge wiperHQ.c_permWipe when Fp.present and FwH.present ;
end

plant automaton PRESENCE_UNCONTROLLED :
    location  :
        initial ; marked ;
        edge button.u_permOn when Fp.present ;
        edge sensorLQ.u_noRain when FsL.present ;
        edge sensorLQ.u_rain when FsL.present ;
```

```
        edge sensorHQ.u_noRain when FsH.present ;
        edge sensorHQ.u_littleRain when FsH.present ;
        edge sensorHQ.u_heavyRain when FsH.present ;
end
```

Listing 4: *Requirements*

```
requirement wiperLQ.c_off needs button.Off ;//R1
requirement wiperLQ.c_noRain needs button.On ;//R2
requirement wiperLQ.c_noRain needs ( FsL.present => sensorLQ.NoRain ) ;//R3
requirement wiperLQ.c_noRain needs ( FsH.present => sensorHQ.NoRain ) ;//R4
requirement wiperLQ.c_rain needs button.On ;//R5
requirement wiperLQ.c_rain needs ( FsH.present =>
               ( sensorHQ.LittleRain or sensorHQ.HeavyRain ) ) ;//R6
requirement wiperLQ.c_rain needs ( FsL.present => sensorLQ.Rain ) ;//R7
requirement wiperLQ.c_permWipe needs button.PermOn ;//R8
requirement wiperHQ.c_off needs button.Off ;//R9
requirement wiperHQ.c_noRain needs button.On ;//R10
requirement wiperHQ.c_noRain needs ( FsL.present => sensorLQ.NoRain ) ;//R11
requirement wiperHQ.c_noRain needs ( FsH.present => sensorHQ.NoRain ) ;//R12
requirement wiperHQ.c_littleRain needs button.On ;//R13
requirement wiperHQ.c_littleRain needs ( FsL.present => sensorLQ.Rain ) ;//R14
requirement wiperHQ.c_littleRain needs ( FsH.present => sensorHQ.LittleRain) ;//R15
requirement wiperHQ.c_heavyRain needs button.On ;//R16
requirement wiperHQ.c_heavyRain needs ( FsH.present => sensorHQ.HeavyRain ) ;//R17
requirement wiperHQ.c_heavyRain needs ( FsL.present => false ) ;//R18
requirement wiperHQ.c_permWipe needs button.PermOn ;//R19
```

Dynamic Wiper System

Listing 5: *Dynamic Wiper System model*

```
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

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

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

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

plant automaton button :
uncontrollable u_off , u_on , u_permOn ;
    location Off :
        initial ; marked ;
        edge u_permOn goto PermOn ;
        edge u_on goto On ;
    location On :
        edge u_permOn goto PermOn ;
        edge u_off goto Off ;
    location PermOn :
        edge u_off goto Off ;
        edge u_on goto On ;
end

plant automaton sensorLQ :
uncontrollable u_noRain , u_rain ;
    location NoRain :
        initial;marked ;
        edge u_rain goto Rain ;
    location Rain :
        edge u_noRain goto NoRain ;
end

plant automaton sensorHQ :
uncontrollable u_noRain  , u_littleRain , u_heavyRain ;
    location NoRain :
        initial ;marked ;
        edge u_littleRain goto LittleRain ;
        edge u_heavyRain goto HeavyRain ;
    location LittleRain :
        edge u_noRain goto NoRain ;
        edge u_heavyRain goto HeavyRain ;
    location HeavyRain :
        edge u_noRain goto NoRain ;
        edge u_littleRain goto LittleRain ;
end
```

```
plant wiperLQ:
controllable c_off , c_noRain , c_rain , c_permWipe ;
    location NoRain :
        initial ; marked ;
        edge c_rain goto Rain ;
        edge c_permWipe goto PermWipe ;
    location PermWipe :
        edge c_off goto NoRain ;
        edge c_rain goto Rain ;
        edge FwL.go goto NoRain;
    location Rain :
        edge c_noRain goto NoRain ;
        edge c_off goto NoRain ;
        edge c_permWipe goto PermWipe ;
        edge FwL.go goto NoRain;
end

plant wiperHQ:
controllable c_off , c_noRain , c_littleRain , c_heavyRain , c_permWipe ;
    location NoRain :
        marked ; initial ;
        edge c_permWipe goto PermWipe ;
        edge c_littleRain goto LittleRain ;
        edge c_heavyRain goto HeavyRain ;
    location PermWipe :
        edge c_off goto NoRain ;
        edge c_littleRain goto LittleRain ;
        edge c_heavyRain goto HeavyRain ;
        edge FwH.go goto NoRain;
    location LittleRain :
        edge c_noRain goto NoRain ;
        edge c_heavyRain goto HeavyRain ;
        edge c_off goto NoRain ;
        edge c_permWipe goto PermWipe ;
        edge FwH.go goto NoRain;
    location HeavyRain :
        edge c_noRain goto NoRain ;
        edge c_littleRain goto LittleRain ;
        edge c_off goto NoRain ;
        edge c_permWipe goto PermWipe ;
        edge FwH.go goto NoRain;
end

plant automaton PRESENCE_CONTROLLED :
    location  :
        initial ; marked ;
        edge wiperLQ.c_off when FwL.present ;
        edge wiperLQ.c_noRain when FwL.present ;
        edge wiperLQ.c_rain when FwL.present ;
        edge wiperLQ.c_permWipe when Fp.present and FwL.present ;
        edge wiperHQ.c_off when FwH.present ;
        edge wiperHQ.c_noRain when FwH.present ;
        edge wiperHQ.c_littleRain when FwH.present ;
        edge wiperHQ.c_heavyRain when FwH.present ;
        edge wiperHQ.c_permWipe when Fp.present and FwH.present ;
end

plant automaton PRESENCE_UNCONTROLLED :
    location  :
        initial ; marked ;
        edge button.u_permOn when Fp.present ;
        edge sensorLQ.u_noRain when FsL.present ;
        edge sensorLQ.u_rain when FsL.present ;
        edge sensorHQ.u_noRain when FsH.present ;
        edge sensorHQ.u_littleRain when FsH.present ;
        edge sensorHQ.u_heavyRain when FsH.present ;
end
```

```
requirement wiperLQ.c_off needs button.Off ;//R1
requirement wiperLQ.c_noRain needs button.On ;//R2
requirement wiperLQ.c_noRain needs ( FsL.present => sensorLQ.NoRain ) ;//R3
requirement wiperLQ.c_noRain needs ( FsH.present => sensorHQ.NoRain ) ;//R4
requirement wiperLQ.c_rain needs button.On ;//R5
requirement wiperLQ.c_rain needs ( FsH.present =>
                ( sensorHQ.LittleRain or sensorHQ.HeavyRain ) ) ;//R6
requirement wiperLQ.c_rain needs ( FsL.present => sensorLQ.Rain ) ;//R7
requirement wiperLQ.c_permWipe needs button.PermOn ;//R8
requirement wiperHQ.c_off needs button.Off ;//R9
requirement wiperHQ.c_noRain needs button.On ;//R10
requirement wiperHQ.c_noRain needs ( FsL.present => sensorLQ.NoRain ) ;//R11
requirement wiperHQ.c_noRain needs ( FsH.present => sensorHQ.NoRain ) ;//R12
requirement wiperHQ.c_littleRain needs button.On ;//R13
requirement wiperHQ.c_littleRain needs ( FsL.present => sensorLQ.Rain ) ;//R14
requirement wiperHQ.c_littleRain needs ( FsH.present => sensorHQ.LittleRain) ;//R15
requirement wiperHQ.c_heavyRain needs button.On ;//R16
requirement wiperHQ.c_heavyRain needs ( FsH.present => sensorHQ.HeavyRain ) ;//R17
requirement wiperHQ.c_heavyRain needs ( FsL.present => false ) ;//R18
requirement wiperHQ.c_permWipe needs button.PermOn ;//R19
requirement invariant (FwL.present and FwH.present) =>
                (wiperLQ.NoRain or wiperHQ.NoRain);//R20
```

Static BCS

Listing 6: *Static BCS model*

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


FBCS:FEATURE();
FHMI:FEATURE(); FDoor:FEATURE(); FSecu:FEATURE();
FPowerW:FEATURE(); FMir:FEATURE(); FAlarm:FEATURE();FCLS:FEATURE();FRCKey:FEATURE();
FLED:FEATURE(); FManPW:FEATURE(); FAutoPW:FEATURE(); FFingerP:FEATURE();
FMirE:FEATURE(); FMirHeat:FEATURE();
FInterMon:FEATURE(); FAutoL:FEATURE(); FCtrAlarm:FEATURE(); FSafe:FEATURE();
FCtrAutoPW:FEATURE(); FAdjMir:FEATURE();
FLEDAlarm:FEATURE(); FLEDFP:FEATURE();FLEDCLS:FEATURE();FLEDPW:FEATURE();
FLEDMir:FEATURE(); FLEDHeat:FEATURE();

alg bool r1=FBCS.present; // Root feature present
alg bool r2=FBCS.present <=> FHMI.present; // HMI mandatory
alg bool r3=FBCS.present <=> FDoor.present; // Door mandatory
alg bool r4=FSecu.present => FBCS.present; // Security optional
alg bool r5=FDoor.present <=> (FPowerW.present and FMir.present); // PW,EM mandatory
alg bool r6=FAlarm.present => FSecu.present; // AS optional
alg bool r7=FCLS.present => FSecu.present; // CLS optional
alg bool r8=FRCKey.present => FSecu.present; // RCK optional
alg bool r9=FLED.present => FHMI.present; // LED optional
alg bool r10=FPowerW.present <=> (FManPW.present <=> not FAutoPW.present);
                //Manual or automatic PW
alg bool r11=FPowerW.present <=> (FFingerP.present); // Finger Protection mandatory
alg bool r12=FMir.present <=> (FMirE.present); // Electric exterior mirror mandatory
alg bool r13=FMirHeat.present => FMir.present; // Mirror heating optional
alg bool r14=FInterMon.present => FAlarm.present; // Interior monitoring optional
alg bool r15=FAutoL.present => FCLS.present; // Automatic locking optional
alg bool r16=FCtrAlarm.present => FRCKey.present; // Control alarm optional
alg bool r17=FSafe.present => FRCKey.present; // Safety optional
alg bool r18=FCtrAutoPW.present => FRCKey.present;
                // Control automatic power window optional
alg bool r19=FAdjMir.present => FRCKey.present; // Safety optional
alg bool r20=(FLEDAlarm.present or FLEDFP.present or FLEDCLS.present or
FLEDPW.present or FLEDMir.present or FLEDHeat.present)<=> FLED.present;
alg bool r21=FLEDAlarm.present => FAlarm.present; //LED alarm requires Alarm
alg bool r22=FLEDCLS.present => FCLS.present; //LED central requires central locking
alg bool r23=FLEDHeat.present => FMirHeat.present;
                //LED heat mirror requires heated mirror
alg bool r24=not(FManPW.present and FCtrAutoPW.present);
                //Manual power windows excludes control autoPW
alg bool r25=FCtrAlarm.present => FAlarm.present;
                //Control alarm requires Alarm system
alg bool r26=FRCKey.present => FCLS.present;
                //Remote control key requires central locking system
alg bool sys_valid=r1 and r2 and r3 and r4 and r5 and r6 and r7 and r8 and r9 and
    r10 and r11 and r12 and r13 and r14 and r15 and r16 and r17 and r18 and r19
    and r20 and r21 and r22 and r23 and r24 and r25 and r26;

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

plant automaton LED_EM_top:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
```

```
    location On:
        edge c_off goto Off;
end

plant automaton LED_EM_left:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_EM_bottom:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_EM_right:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CHECK_HMI_LED_EM:
location : initial; marked;
        edge LED_EM_top.c_off when FLEDMir.present;
        edge LED_EM_top.c_on when FLEDMir.present;
        edge LED_EM_left.c_off when FLEDMir.present;
        edge LED_EM_left.c_on when FLEDMir.present;
        edge LED_EM_bottom.c_off when FLEDMir.present;
        edge LED_EM_bottom.c_on when FLEDMir.present;
        edge LED_EM_right.c_off when FLEDMir.present;
        edge LED_EM_right.c_on when FLEDMir.present;
end

requirement LED_EM_top.c_on needs positionEM.EM_top or positionEM.EM_top_left or
                positionEM.EM_top_right;//R1
requirement LED_EM_top.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_hor_left or positionEM.EM_hor_right;//R2
requirement LED_EM_left.c_on needs positionEM.EM_hor_left or positionEM.EM_top_left
                or positionEM.EM_bottom_left;//R3
requirement LED_EM_left.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_bottom or positionEM.EM_top;//R4
requirement LED_EM_bottom.c_on needs positionEM.EM_bottom or
                positionEM.EM_bottom_left or positionEM.EM_bottom_right;//R5
requirement LED_EM_bottom.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_hor_left or positionEM.EM_hor_right;//R6
requirement LED_EM_right.c_on needs positionEM.EM_hor_right or
                positionEM.EM_top_right or positionEM.EM_bottom_right;//R7
requirement LED_EM_right.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_bottom or positionEM.EM_top;//R8

plant automaton LED_PW_FP:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end
```

```
plant automaton PRESENCE_CONTROLLED_LED_FP:
location : initial; marked;
    edge LED_PW_FP.c_on when FLEDFP.present;
    edge LED_PW_FP.c_off when FLEDFP.present;
end

requirement LED_PW_up.c_on needs fingerPW.On;//R9
requirement LED_PW_up.c_off needs fingerPW.Off;//R10

plant automaton LED_PW_up:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_PW_dn:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CONTROLLED_PW_DN_UP:
location : initial; marked;
    edge LED_PW_up.c_on when FLEDPW.present;
    edge LED_PW_up.c_off when FLEDPW.present;
    edge LED_PW_dn.c_on when FLEDPW.present;
    edge LED_PW_dn.c_off when FLEDPW.present;
end

requirement LED_PW_up.c_on needs motorPW.Up;//R11
requirement LED_PW_up.c_off needs motorPW.Idle;//R12
requirement LED_PW_dn.c_on needs motorPW.Down;//R13
requirement LED_PW_dn.c_off needs motorPW.Idle;//R14

plant automaton LED_EM_heating:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CONTROLLED_EM_HEAT:
location : initial; marked;
    edge LED_EM_heating.c_on when FLEDHeat.present;
    edge LED_EM_heating.c_off when FLEDHeat.present;
end

requirement LED_EM_heating.c_on needs EM_heating.On;//R15
requirement LED_EM_heating.c_off needs EM_heating.Off;//R16

plant automaton LED_CLS:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end
```

```
plant automaton PRESENCE_CONTROLLED_LED_CLS:
location : initial; marked;
    edge LED_CLS.c_on when FLEDCLS.present;
    edge LED_CLS.c_off when FLEDCLS.present;
end

requirement LED_CLS.c_on needs CLS.Locked;//R17
requirement LED_CLS.c_off needs CLS.Unlocked;//R18

plant automaton LED_AS_active:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_AS_alarm:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_AS_alarm_det:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_AS_IM:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CONTROLLED_LED_AS:
location : initial; marked;
    edge LED_AS_active.c_on when FLEDAlarm.present;
    edge LED_AS_active.c_off when FLEDAlarm.present;
    edge LED_AS_alarm.c_on when FLEDAlarm.present;
    edge LED_AS_alarm.c_off when FLEDAlarm.present;
    edge LED_AS_alarm_det.c_on when FLEDAlarm.present;
    edge LED_AS_alarm_det.c_off when FLEDAlarm.present;
    edge LED_AS_IM.c_on when FLEDAlarm.present;
    edge LED_AS_IM.c_off when FLEDAlarm.present;
end

requirement LED_AS_active.c_on needs AlarmSystem.Activated;//R19
requirement LED_AS_active.c_off needs AlarmSystem.Deactivated;//R20
requirement LED_AS_alarm.c_on needs AlarmSystem.On;//R21
requirement LED_AS_alarm.c_off needs AlarmSystem.Activated;//R22
requirement LED_AS_alarm_det.c_on needs AlarmSystem.Alarm_detected;//R23
requirement LED_AS_alarm_det.c_off needs AlarmSystem.On;//R24
requirement LED_AS_IM.c_on needs InteriorMonitoring.On;//R25
requirement LED_AS_IM.c_off needs InteriorMonitoring.Off;//R26

plant automaton buttonPW:
    uncontrollable u_up, u_down, u_released;
```

```
    location Idle:
        initial; marked;
        edge u_up goto Up;
        edge u_down goto Down;
    location Down:
        edge u_released goto Idle;
        edge u_up goto Up;
    location Up:
        edge u_released goto Idle;
        edge u_down goto Down;
end

plant automaton sensorPW:
    uncontrollable u_up, u_down, u_inBetween;
    location InBetween:
        edge u_up goto Up;
        edge u_down goto Down;
    location Up:
        initial; marked;
        edge u_inBetween goto InBetween;
    location Down:
        edge u_inBetween goto InBetween;
end

plant automaton fingerPW:
    uncontrollable u_fingerDetected;
    controllable c_fingerReleased;
    location Off:
    initial;marked;
    edge u_fingerDetected goto On;
    location On:
    edge c_fingerReleased goto Off;
end

plant automaton motorPW:
    controllable c_up, c_down, c_off_dn, c_off_up;
    location Idle:
        initial;marked;
        edge c_up goto Up;
        edge c_down goto Down;
    location Up:
        edge c_up;
        edge c_off_up goto Idle;
        edge CLS.c_lock goto Up_CLS;
    location Up_CLS:
        edge CLS.c_unlock goto Up;
    location Down:
        edge c_off_dn goto Idle;
        edge c_down;
        edge CLS.c_lock goto Down_CLS;
    location Down_CLS:
        edge CLS.c_unlock goto Down;
end

plant automaton PRESENCE_UNCONTROLLED_PW:
location : initial; marked;
    edge buttonPW.u_up when FPowerW.present;
    edge buttonPW.u_down when FPowerW.present;
    edge buttonPW.u_released when FPowerW.present;
    edge sensorPW.u_up when FPowerW.present;
    edge sensorPW.u_down when FPowerW.present;
    edge sensorPW.u_inBetween when FPowerW.present;
    edge fingerPW.u_fingerDetected when FFingerP.present;
end

plant automaton PRESENCE_CONTROLLED_PW:
location : initial; marked;
    edge fingerPW.c_fingerReleased when FFingerP.present;
```

```
    edge motorPW.c_up when FPowerW.present;
    edge motorPW.c_down when FPowerW.present;
    edge motorPW.c_off_dn when FPowerW.present;
    edge motorPW.c_off_up when FPowerW.present;
end

requirement motorPW.c_up needs buttonPW.Up or RCK_PW.Up;//R27
requirement motorPW.c_up needs fingerPW.Off;//R28
requirement motorPW.c_up needs not sensorPW.Up;//R29
requirement motorPW.c_down needs buttonPW.Down  or RCK_PW.Down;//R30
requirement motorPW.c_down needs fingerPW.Off;//R31
requirement motorPW.c_down needs not sensorPW.Down;//R32
requirement motorPW.c_off_dn needs fingerPW.On or (FManPW.present => buttonPW.Idle)
        or sensorPW.Down or (FAutoPW.present => (buttonPW.Up or RCK_PW.Up));//R33
requirement motorPW.c_off_up needs fingerPW.On or (FManPW.present => buttonPW.Idle)
        or sensorPW.Up or (FAutoPW.present => (buttonPW.Down or RCK_PW.Down));//R34
requirement fingerPW.c_fingerReleased needs buttonPW.Down;//R35

plant automaton buttonEM:
    uncontrollable u_up, u_down, u_left, u_right, u_released;
    location Idle:marked;initial;
        edge u_up goto Up;
        edge u_down goto Down;
        edge u_left goto Left;
        edge u_right goto Right;
    location Up:
        edge u_released goto Idle;
    location Down:
        edge u_released goto Idle;
    location Left:
        edge u_released goto Idle;
    location Right:
        edge u_released goto Idle;
end

plant automaton positionEM:
    uncontrollable u_pos_top, u_pos_down, u_pos_left, u_pos_right;
    uncontrollable u_rel_top, u_rel_down, u_rel_left, u_rel_right;
    location EM_hor_pending:
        initial;marked;
        edge u_pos_top goto EM_top;
        edge u_pos_down goto EM_bottom;
        edge u_pos_left goto EM_hor_left;
        edge u_pos_right goto EM_hor_right;
    location EM_top:
        edge u_pos_left goto EM_top_left;
        edge u_pos_right goto EM_top_right;
        edge u_rel_top goto EM_hor_pending;
    location EM_top_left:
        edge u_rel_top goto EM_hor_left;
        edge u_rel_left goto EM_top;
    location EM_top_right:
        edge u_rel_top goto EM_hor_right;
        edge u_rel_right goto EM_top;
    location EM_bottom:
        edge u_rel_down goto EM_hor_pending;
        edge u_pos_left goto EM_bottom_left;
        edge u_pos_right goto EM_bottom_right;
    location EM_bottom_left:
        edge u_rel_down goto EM_hor_left;
        edge u_rel_left goto EM_bottom;
    location EM_bottom_right:
        edge u_rel_down goto EM_hor_right;
        edge u_rel_right goto EM_bottom;
    location EM_hor_left:
        edge u_rel_left goto EM_hor_pending;
    location EM_hor_right:
        edge u_rel_right goto EM_hor_pending;
```

```
end

plant automaton motorEM:
    controllable c_left, c_right, c_up, c_down, c_off;
    location Idle:initial; marked;
        edge c_left goto Left;
        edge c_right goto Right;
        edge c_up goto Up;
        edge c_down goto Down;
    location Left:
        edge c_off goto Idle;
    location Right:
        edge c_off goto Idle;
    location Up:
        edge c_off goto Idle;
    location Down:
        edge c_off goto Idle;
end

plant automaton PRESENCE_UNCONTROLLED_EM:
location : initial; marked;
    edge buttonEM.u_up when FMir.present;
    edge buttonEM.u_down when FMir.present;
    edge buttonEM.u_left when FMir.present;
    edge buttonEM.u_right when FMir.present;
    edge positionEM.u_pos_top when FMir.present;
    edge positionEM.u_pos_down when FMir.present;
    edge positionEM.u_pos_left when FMir.present;
    edge positionEM.u_pos_right when FMir.present;
    edge positionEM.u_rel_top when FMir.present;
    edge positionEM.u_rel_down when FMir.present;
    edge positionEM.u_rel_left when FMir.present;
    edge positionEM.u_rel_right when FMir.present;
end

plant automaton PRESENCE_CONTROLLED_EM:
location : initial; marked;
    edge motorEM.c_up when FMir.present;
    edge motorEM.c_down when FMir.present;
    edge motorEM.c_left when FMir.present;
    edge motorEM.c_right when FMir.present;
    edge motorEM.c_off when FMir.present;
end

requirement motorEM.c_left needs not(positionEM.EM_hor_left or
                positionEM.EM_top_left or positionEM.EM_bottom_left);//R36
requirement motorEM.c_right needs not(positionEM.EM_hor_right or
                positionEM.EM_top_right or positionEM.EM_bottom_right);//R37
requirement motorEM.c_up needs not(positionEM.EM_top or positionEM.EM_top_right
                or positionEM.EM_top_left);//R38
requirement motorEM.c_down needs not(positionEM.EM_bottom or
                positionEM.EM_bottom_right or positionEM.EM_bottom_left);//R39
requirement motorEM.c_left needs buttonEM.Left or RCK_EM.Left;//R40
requirement motorEM.c_right needs buttonEM.Right or RCK_EM.Right;//R41
requirement motorEM.c_up needs buttonEM.Up or RCK_EM.Up;//R42
requirement motorEM.c_down needs buttonEM.Down or RCK_EM.Down;//R43
requirement motorEM.c_off needs buttonEM.Idle or RCK_EM.Idle;//R44

plant automaton EM_temp_time:
    uncontrollable u_lowtemp, u_done;
    location Heating_off:initial;marked;
        edge u_lowtemp goto Heating_on;
    location Heating_on:
        edge u_done goto Heating_off;
end

plant automaton EM_heating:
    controllable c_on, c_off;
```

```
    location Off:initial;marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_UNCONTROLLED_MIRHEAT:
location : initial; marked;
    edge EM_temp_time.u_lowtemp when FMirHeat.present;
    edge EM_temp_time.u_done when FMirHeat.present;
end

plant automaton PRESENCE_CONTROLLED_MIRHEAT:
location : initial; marked;
    edge EM_heating.c_on when FMirHeat.present;
    edge EM_heating.c_off when FMirHeat.present;
end

requirement EM_heating.c_on needs EM_temp_time.Heating_on;//R45
requirement EM_heating.c_on needs EM_temp_time.Heating_off;//R46

plant automaton AlarmSystem:
    controllable c_on, c_off,c_deactivated, c_activated, c_IM_detected;
    uncontrollable u_detected, u_time_elapsed;
    location Deactivated:
        edge c_activated goto Activated;
    location Activated:
        initial;marked;
        edge c_on goto On;
        edge c_deactivated goto Deactivated;
    location On:
        edge c_off goto Activated;
        edge u_detected goto Alarm_detected;
        edge c_IM_detected goto Alarm_detected;
    location Alarm_detected:
        edge c_off goto Activated;
        edge u_time_elapsed goto On;
end

plant automaton InteriorMonitoring:
    uncontrollable u_detected, u_clear;
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
        edge u_detected goto Detected;
    location Detected:
        edge u_clear goto On;
        edge c_off goto Off;
end

plant automaton PRESENCE_UNCONTROLLED_AS:
location : initial; marked;
    edge AlarmSystem.u_detected when FAlarm.present;
    edge AlarmSystem.u_time_elapsed when FAlarm.present;
    edge InteriorMonitoring.u_detected when FInterMon.present;
    edge InteriorMonitoring.u_clear when FInterMon.present;
end

plant automaton PRESENCE_CONTROLLED_AS:
location : initial; marked;
    edge AlarmSystem.c_on when FAlarm.present;
    edge AlarmSystem.c_off when FAlarm.present;
    edge AlarmSystem.c_deactivated when FAlarm.present;
    edge AlarmSystem.c_IM_detected when FInterMon.present;
    edge InteriorMonitoring.c_on when FInterMon.present;
```

```
    edge InteriorMonitoring.c_off when FInterMon.present;
end

requirement AlarmSystem.c_on needs Key_lock.Locked or RCK_CLS.Locked;//R47
requirement AlarmSystem.c_off needs Key_lock.Unlocked or RCK_CLS.Unlocked;//R48
requirement AlarmSystem.c_deactivated needs Key_lock.Unlocked or
               RCK_CLS.Unlocked;//R49
requirement AlarmSystem.c_IM_detected needs InteriorMonitoring.Detected;//R50
requirement InteriorMonitoring.c_off needs Key_lock.Unlocked or
               RCK_CLS.Unlocked;//R51

plant automaton Key_lock:
    uncontrollable u_lock, u_unlock;
    location Unlocked:
    initial;marked;
        edge u_lock goto Locked;
    location Locked:
        edge u_unlock goto Unlocked;
end

plant automaton CLS:
    controllable c_lock, c_unlock;
    location Unlocked:
    initial;marked;
        edge c_lock goto Locked;
    location Locked:
    initial;marked;
        edge c_unlock goto Unlocked;
end

plant automaton Auto_lock:
    uncontrollable u_drive_lock, u_door_open_unlock;
    location Unlocked:
    initial;marked;
        edge u_drive_lock goto Locked;
    location Locked:
    initial;marked;
        edge u_door_open_unlock goto Unlocked;
end

plant automaton PRESENCE_UNCONTROLLED_CLS:
location : initial; marked;
    edge Key_lock.u_lock when FCLS.present;
    edge Key_lock.u_unlock when FCLS.present;
    edge Auto_lock.u_drive_lock when FAutoL.present;
    edge Auto_lock.u_door_open_unlock when FAutoL.present;
end

plant automaton PRESENCE_CONTROLLED_CLS:
location : initial; marked;
    edge CLS.c_lock when FCLS.present;
    edge CLS.c_unlock when FCLS.present;
end

requirement CLS.c_lock needs Key_lock.Locked or RCK_CLS.Locked
               or Auto_lock.Locked;//R52
requirement CLS.c_unlock needs Key_lock.Unlocked or RCK_CLS.Unlocked;//R53

plant automaton RCK_CLS:
    uncontrollable u_lock, u_unlock;
    location Unlocked:
    initial;marked;
        edge u_lock goto Locked;
    location Locked:
    initial;marked;
        edge u_unlock goto Unlocked;
end
```

```
plant automaton Safety_fcn:
    uncontrollable u_time_elapsed_lock, u_door_open;
    controllable c_locked, c_unlock;
    location Idle:
    initial;marked;
        edge c_unlock goto Unlocked_RCK;
    location Unlocked_RCK:
    marked;
        edge u_time_elapsed_lock goto AutoLocking;
        edge u_door_open goto Idle;
    location AutoLocking:
        edge CLS.c_lock goto Idle;
end

plant automaton RCK_PW:
    uncontrollable u_up, u_down, u_released;
    location Idle:
    initial;marked;
        edge u_up goto Up;
        edge u_down goto Down;
    location Up:
        edge u_released goto Idle;
    location Down:
        edge u_released goto Idle;
end

plant automaton RCK_EM:
    uncontrollable u_up, u_down, u_left, u_right, u_released;
    location Idle:marked;initial;
        edge u_up goto Up;
        edge u_down goto Down;
        edge u_left goto Left;
        edge u_right goto Right;
    location Up:
        edge u_released goto Idle;
    location Down:
        edge u_released goto Idle;
    location Left:
        edge u_released goto Idle;
    location Right:
        edge u_released goto Idle;
end

plant automaton PRESENCE_UNCONTROLLED_SAFETY_RCK:
location : initial; marked;
    edge Safety_fcn.u_time_elapsed_lock when FSafe.present;
    edge Safety_fcn.u_door_open when FSafe.present;
    edge RCK_CLS.u_lock when FCLS.present;
    edge RCK_CLS.u_unlock when FCLS.present;
    edge RCK_PW.u_up when FCtrAutoPW.present;
    edge RCK_PW.u_down when FCtrAutoPW.present;
    edge RCK_PW.u_released when FCtrAutoPW.present;
    edge RCK_EM.u_up when FAdjMir.present;
    edge RCK_EM.u_down when FAdjMir.present;
    edge RCK_EM.u_left when FAdjMir.present;
    edge RCK_EM.u_right when FAdjMir.present;
    edge RCK_EM.u_released when FAdjMir.present;
end

plant automaton PRESENCE_CONTROLLED_SAFETY_RCK:
location : initial; marked;
    edge Safety_fcn.c_unlock when FSafe.present;
    edge Safety_fcn.c_locked when FSafe.present;
end

requirement Safety_fcn.c_unlock needs RCK_CLS.Unlocked;//R54
```

Dynamic BCS

Listing 7: *Dynamic BCS model*

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


FBCS:FEATURE();
FHMI:FEATURE(); FDoor:FEATURE(); FSecu:FEATURE();
FPowerW:FEATURE(); FMir:FEATURE(); FAlarm:FEATURE();FCLS:FEATURE();FRCKey:FEATURE();
FLED:FEATURE(); FManPW:FEATURE(); FAutoPW:FEATURE(); FFingerP:FEATURE();
FMirE:FEATURE(); FMirHeat:FEATURE();
FInterMon:FEATURE(); FAutoL:FEATURE(); FCtrAlarm:FEATURE(); FSafe:FEATURE();
FCtrAutoPW:FEATURE(); FAdjMir:FEATURE();
FLEDAlarm:FEATURE(); FLEDFP:FEATURE();FLEDCLS:FEATURE();FLEDPW:FEATURE();
FLEDMir:FEATURE(); FLEDHeat:FEATURE();

alg bool r1=FBCS.present; // Root feature present
alg bool r2=FBCS.present <=> FHMI.present; // HMI mandatory
alg bool r3=FBCS.present <=> FDoor.present; // Door mandatory
alg bool r4=FSecu.present => FBCS.present; // Security optional
alg bool r5=FDoor.present <=> (FPowerW.present and FMir.present); // PW,EM mandatory
alg bool r6=FAlarm.present => FSecu.present; // AS optional
alg bool r7=FCLS.present => FSecu.present; // CLS optional
alg bool r8=FRCKey.present => FSecu.present; // RCK optional
alg bool r9=FLED.present => FHMI.present; // LED optional
alg bool r10=FPowerW.present <=> (FManPW.present <=> not FAutoPW.present);
                //Manual or automatic PW
alg bool r11=FPowerW.present <=> (FFingerP.present); // Finger Protection mandatory
alg bool r12=FMir.present <=> (FMirE.present); // Electric exterior mirror mandatory
alg bool r13=FMirHeat.present => FMir.present; // Mirror heating optional
alg bool r14=FInterMon.present => FAlarm.present; // Interior monitoring optional
alg bool r15=FAutoL.present => FCLS.present; // Automatic locking optional
alg bool r16=FCtrAlarm.present => FRCKey.present; // Control alarm optional
alg bool r17=FSafe.present => FRCKey.present; // Safety optional
alg bool r18=FCtrAutoPW.present => FRCKey.present;
                // Control automatic power window optional
alg bool r19=FAdjMir.present => FRCKey.present; // Safety optional
alg bool r20=(FLEDAlarm.present or FLEDFP.present or FLEDCLS.present or
FLEDPW.present or FLEDMir.present or FLEDHeat.present)<=> FLED.present;
alg bool r21=FLEDAlarm.present => FAlarm.present; //LED alarm requires Alarm
alg bool r22=FLEDCLS.present => FCLS.present; //LED central requires central locking
alg bool r23=FLEDHeat.present => FMirHeat.present;
                //LED heat mirror requires heated mirror
alg bool r24=not(FManPW.present and FCtrAutoPW.present);
                //Manual power windows excludes control autoPW
alg bool r25=FCtrAlarm.present => FAlarm.present;
                //Control alarm requires Alarm system
alg bool r26=FRCKey.present => FCLS.present;
                //Remote control key requires central locking system
alg bool sys_valid=r1 and r2 and r3 and r4 and r5 and r6 and r7 and r8 and r9 and
    r10 and r11 and r12 and r13 and r14 and r15 and r16 and r17 and r18 and r19
    and r20 and r21 and r22 and r23 and r24 and r25 and r26;

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

plant automaton LED_EM_top:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
```

```
    location On:
        edge c_off goto Off;
end

plant automaton LED_EM_left:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_EM_bottom:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_EM_right:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CHECK_HMI_LED_EM:
location : initial; marked;
        edge LED_EM_top.c_off when FLEDMir.present;
        edge LED_EM_top.c_on when FLEDMir.present;
        edge LED_EM_left.c_off when FLEDMir.present;
        edge LED_EM_left.c_on when FLEDMir.present;
        edge LED_EM_bottom.c_off when FLEDMir.present;
        edge LED_EM_bottom.c_on when FLEDMir.present;
        edge LED_EM_right.c_off when FLEDMir.present;
        edge LED_EM_right.c_on when FLEDMir.present;
end

requirement LED_EM_top.c_on needs positionEM.EM_top or positionEM.EM_top_left or
                positionEM.EM_top_right;//R1
requirement LED_EM_top.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_hor_left or positionEM.EM_hor_right;//R2
requirement LED_EM_left.c_on needs positionEM.EM_hor_left or positionEM.EM_top_left
                or positionEM.EM_bottom_left;//R3
requirement LED_EM_left.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_bottom or positionEM.EM_top;//R4
requirement LED_EM_bottom.c_on needs positionEM.EM_bottom or
                positionEM.EM_bottom_left or positionEM.EM_bottom_right;//R5
requirement LED_EM_bottom.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_hor_left or positionEM.EM_hor_right;//R6
requirement LED_EM_right.c_on needs positionEM.EM_hor_right or
                positionEM.EM_top_right or positionEM.EM_bottom_right;//R7
requirement LED_EM_right.c_off needs positionEM.EM_hor_pending or
                positionEM.EM_bottom or positionEM.EM_top;//R8

plant automaton LED_PW_FP:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end
```

```
plant automaton PRESENCE_CONTROLLED_LED_FP:
location : initial; marked;
    edge LED_PW_FP.c_on when FLEDFP.present;
    edge LED_PW_FP.c_off when FLEDFP.present;
end

requirement LED_PW_up.c_on needs fingerPW.On;//R9
requirement LED_PW_up.c_off needs fingerPW.Off;//R10

plant automaton LED_PW_up:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_PW_dn:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CONTROLLED_PW_DN_UP:
location : initial; marked;
    edge LED_PW_up.c_on when FLEDPW.present;
    edge LED_PW_up.c_off when FLEDPW.present;
    edge LED_PW_dn.c_on when FLEDPW.present;
    edge LED_PW_dn.c_off when FLEDPW.present;
end

requirement LED_PW_up.c_on needs motorPW.Up;//R11
requirement LED_PW_up.c_off needs motorPW.Idle;//R12
requirement LED_PW_dn.c_on needs motorPW.Down;//R13
requirement LED_PW_dn.c_off needs motorPW.Idle;//R14

plant automaton LED_EM_heating:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CONTROLLED_EM_HEAT:
location : initial; marked;
    edge LED_EM_heating.c_on when FLEDHeat.present;
    edge LED_EM_heating.c_off when FLEDHeat.present;
end

requirement LED_EM_heating.c_on needs EM_heating.On;//R15
requirement LED_EM_heating.c_off needs EM_heating.Off;//R16

plant automaton LED_CLS:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end
```

```
plant automaton PRESENCE_CONTROLLED_LED_CLS:
location : initial; marked;
    edge LED_CLS.c_on when FLEDCLS.present;
    edge LED_CLS.c_off when FLEDCLS.present;
end

requirement LED_CLS.c_on needs CLS.Locked;//R17
requirement LED_CLS.c_off needs CLS.Unlocked;//R18

plant automaton LED_AS_active:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_AS_alarm:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_AS_alarm_det:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton LED_AS_IM:
    controllable c_on, c_off;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_CONTROLLED_LED_AS:
location : initial; marked;
    edge LED_AS_active.c_on when FLEDAlarm.present;
    edge LED_AS_active.c_off when FLEDAlarm.present;
    edge LED_AS_alarm.c_on when FLEDAlarm.present;
    edge LED_AS_alarm.c_off when FLEDAlarm.present;
    edge LED_AS_alarm_det.c_on when FLEDAlarm.present;
    edge LED_AS_alarm_det.c_off when FLEDAlarm.present;
    edge LED_AS_IM.c_on when FLEDAlarm.present;
    edge LED_AS_IM.c_off when FLEDAlarm.present;
end

requirement LED_AS_active.c_on needs AlarmSystem.Activated;//R19
requirement LED_AS_active.c_off needs AlarmSystem.Deactivated;//R20
requirement LED_AS_alarm.c_on needs AlarmSystem.On;//R21
requirement LED_AS_alarm.c_off needs AlarmSystem.Activated;//R22
requirement LED_AS_alarm_det.c_on needs AlarmSystem.Alarm_detected;//R23
requirement LED_AS_alarm_det.c_off needs AlarmSystem.On;//R24
requirement LED_AS_IM.c_on needs InteriorMonitoring.On;//R25
requirement LED_AS_IM.c_off needs InteriorMonitoring.Off;//R26

plant automaton buttonPW:
    uncontrollable u_up, u_down, u_released;
```

```
    location Idle:
        initial; marked;
        edge u_up goto Up;
        edge u_down goto Down;
    location Down:
        edge u_released goto Idle;
        edge u_up goto Up;
    location Up:
        edge u_released goto Idle;
        edge u_down goto Down;
end

plant automaton sensorPW:
    uncontrollable u_up, u_down, u_inBetween;
    location InBetween:
        edge u_up goto Up;
        edge u_down goto Down;
    location Up:
        initial; marked;
        edge u_inBetween goto InBetween;
    location Down:
        edge u_inBetween goto InBetween;
end

plant automaton fingerPW:
    uncontrollable u_fingerDetected;
    controllable c_fingerReleased;
    location Off:
    initial;marked;
    edge u_fingerDetected goto On;
    location On:
    edge c_fingerReleased goto Off;
end

plant automaton motorPW:
    controllable c_up, c_down, c_off_dn, c_off_up, c_off_reconf;
    location Idle:
        initial;marked;
        edge c_up goto Up;
        edge c_down goto Down;
    location Up:
        edge c_off_reconf goto Idle;
        edge c_up;
        edge c_off_up goto Idle;
        edge CLS.c_lock goto Up_CLS;
    location Up_CLS:
        edge c_off_reconf goto Idle;
        edge CLS.c_unlock goto Up;
    location Down:
        edge c_off_dn goto Idle;
        edge c_down;
        edge CLS.c_lock goto Down_CLS;
    location Down_CLS:
        edge CLS.c_unlock goto Down;
end

plant automaton PRESENCE_UNCONTROLLED_PW:
location : initial; marked;
    edge buttonPW.u_up when FPowerW.present;
    edge buttonPW.u_down when FPowerW.present;
    edge buttonPW.u_released when FPowerW.present;
    edge sensorPW.u_up when FPowerW.present;
    edge sensorPW.u_down when FPowerW.present;
    edge sensorPW.u_inBetween when FPowerW.present;
    edge fingerPW.u_fingerDetected when FFingerP.present;
end

plant automaton PRESENCE_CONTROLLED_PW:
```

```
location : initial; marked;
    edge fingerPW.c_fingerReleased when FFingerP.present;
    edge motorPW.c_up when FPowerW.present;
    edge motorPW.c_down when FPowerW.present;
    edge motorPW.c_off_dn when FPowerW.present;
    edge motorPW.c_off_up when FPowerW.present;
end

requirement motorPW.c_up needs buttonPW.Up or RCK_PW.Up;//R27
requirement motorPW.c_up needs fingerPW.Off;//R28
requirement motorPW.c_up needs not sensorPW.Up;//R29
requirement motorPW.c_down needs buttonPW.Down  or RCK_PW.Down;//R30
requirement motorPW.c_down needs fingerPW.Off;//R31
requirement motorPW.c_down needs not sensorPW.Down;//R32
requirement motorPW.c_off_dn needs fingerPW.On or (FManPW.present => buttonPW.Idle)
        or  sensorPW.Down or (FAutoPW.present => (buttonPW.Up or RCK_PW.Up));//R33
requirement motorPW.c_off_up needs fingerPW.On or (FManPW.present => buttonPW.Idle)
        or sensorPW.Up or (FAutoPW.present => (buttonPW.Down or RCK_PW.Down));//R34
requirement fingerPW.c_fingerReleased needs buttonPW.Down;//R35
requirement not (not sys_valid and motorPW.Up ) ; //R55
requirement not (not sys_valid and motorPW.Up_CLS ) ;//R56
requirement motorPW.c_off_reconf needs not sys_valid ;//R57

plant automaton buttonEM:
    uncontrollable u_up, u_down, u_left, u_right, u_released;
    location Idle:marked;initial;
        edge u_up goto Up;
        edge u_down goto Down;
        edge u_left goto Left;
        edge u_right goto Right;
    location Up:
        edge u_released goto Idle;
    location Down:
        edge u_released goto Idle;
    location Left:
        edge u_released goto Idle;
    location Right:
        edge u_released goto Idle;
end

plant automaton positionEM:
    uncontrollable u_pos_top, u_pos_down, u_pos_left, u_pos_right;
    uncontrollable u_rel_top, u_rel_down, u_rel_left, u_rel_right;
    location EM_hor_pending:
        initial;marked;
        edge u_pos_top goto EM_top;
        edge u_pos_down goto EM_bottom;
        edge u_pos_left goto EM_hor_left;
        edge u_pos_right goto EM_hor_right;
    location EM_top:
        edge u_pos_left goto EM_top_left;
        edge u_pos_right goto EM_top_right;
        edge u_rel_top goto EM_hor_pending;
    location EM_top_left:
        edge u_rel_top goto EM_hor_left;
        edge u_rel_left goto EM_top;
    location EM_top_right:
        edge u_rel_top goto EM_hor_right;
        edge u_rel_right goto EM_top;
    location EM_bottom:
        edge u_rel_down goto EM_hor_pending;
        edge u_pos_left goto EM_bottom_left;
        edge u_pos_right goto EM_bottom_right;
    location EM_bottom_left:
        edge u_rel_down goto EM_hor_left;
        edge u_rel_left goto EM_bottom;
    location EM_bottom_right:
        edge u_rel_down goto EM_hor_right;
```

```
        edge u_rel_right goto EM_bottom;
    location EM_hor_left:
        edge u_rel_left goto EM_hor_pending;
    location EM_hor_right:
        edge u_rel_right goto EM_hor_pending;
end

plant automaton motorEM:
    controllable c_left, c_right, c_up, c_down, c_off;
    location Idle:initial; marked;
        edge c_left goto Left;
        edge c_right goto Right;
        edge c_up goto Up;
        edge c_down goto Down;
    location Left:
        edge c_off goto Idle;
    location Right:
        edge c_off goto Idle;
    location Up:
        edge c_off goto Idle;
    location Down:
        edge c_off goto Idle;
end

plant automaton PRESENCE_UNCONTROLLED_EM:
location : initial; marked;
    edge buttonEM.u_up when FMir.present;
    edge buttonEM.u_down when FMir.present;
    edge buttonEM.u_left when FMir.present;
    edge buttonEM.u_right when FMir.present;
    edge positionEM.u_pos_top when FMir.present;
    edge positionEM.u_pos_down when FMir.present;
    edge positionEM.u_pos_left when FMir.present;
    edge positionEM.u_pos_right when FMir.present;
    edge positionEM.u_rel_top when FMir.present;
    edge positionEM.u_rel_down when FMir.present;
    edge positionEM.u_rel_left when FMir.present;
    edge positionEM.u_rel_right when FMir.present;
end

plant automaton PRESENCE_CONTROLLED_EM:
location : initial; marked;
    edge motorEM.c_up when FMir.present;
    edge motorEM.c_down when FMir.present;
    edge motorEM.c_left when FMir.present;
    edge motorEM.c_right when FMir.present;
    edge motorEM.c_off when FMir.present;
end

requirement motorEM.c_left needs not(positionEM.EM_hor_left or
                positionEM.EM_top_left  or positionEM.EM_bottom_left);//R36
requirement motorEM.c_right needs not(positionEM.EM_hor_right or
                positionEM.EM_top_right or positionEM.EM_bottom_right);//R37
requirement motorEM.c_up needs not(positionEM.EM_top or
positionEM.EM_top_right or
                positionEM.EM_top_left);//R38
requirement motorEM.c_down needs not(positionEM.EM_bottom or
                positionEM.EM_bottom_right or positionEM.EM_bottom_left);//R39
requirement motorEM.c_left needs buttonEM.Left or RCK_EM.Left;//R40
requirement motorEM.c_right needs buttonEM.Right or RCK_EM.Right;//R41
requirement motorEM.c_up needs buttonEM.Up or RCK_EM.Up;//R42
requirement motorEM.c_down needs buttonEM.Down or RCK_EM.Down;//R43
requirement motorEM.c_off needs buttonEM.Idle or RCK_EM.Idle;//R44

plant automaton EM_temp_time:
    uncontrollable u_lowtemp, u_done;
    location Heating_off:initial;marked;
        edge u_lowtemp goto Heating_on;
```

```
    location Heating_on:
        edge u_done goto Heating_off;
end

plant automaton EM_heating:
    controllable c_on, c_off;
    location Off:initial;marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
end

plant automaton PRESENCE_UNCONTROLLED_MIRHEAT:
location : initial; marked;
    edge EM_temp_time.u_lowtemp when FMirHeat.present;
    edge EM_temp_time.u_done when FMirHeat.present;
end

plant automaton PRESENCE_CONTROLLED_MIRHEAT:
location : initial; marked;
    edge EM_heating.c_on when FMirHeat.present;
    edge EM_heating.c_off when FMirHeat.present;
end

requirement EM_heating.c_on needs EM_temp_time.Heating_on;//R45
requirement EM_heating.c_on needs EM_temp_time.Heating_off;//R46

plant automaton AlarmSystem:
    controllable c_on, c_off,c_off_reconf,c_deactivated, c_activated, c_IM_detected;
    uncontrollable u_detected, u_time_elapsed;
    location Deactivated:
        edge c_activated goto Activated;
    location Activated:
        initial;marked;
        edge c_on goto On;
        edge c_deactivated goto Deactivated;
        edge c_off_reconf goto Deactivated;
    location On:
        edge c_off_reconf goto Deactivated;
        edge c_off goto Activated;
        edge u_detected goto Alarm_detected;
        edge c_IM_detected goto Alarm_detected;
    location Alarm_detected:
        edge c_off_reconf goto Deactivated;
        edge c_off goto Activated;
        edge u_time_elapsed goto On;
end

plant automaton InteriorMonitoring:
    uncontrollable u_detected, u_clear;
    controllable c_on, c_off, c_off_reconf;
    location Off:
    initial; marked;
        edge c_on goto On;
    location On:
        edge c_off goto Off;
        edge u_detected goto Detected;
        edge c_off_reconf goto Off;
    location Detected:
        edge u_clear goto On;
        edge c_off goto Off;
        edge c_off_reconf goto Off;
end

plant automaton PRESENCE_UNCONTROLLED_AS:
location : initial; marked;
    edge AlarmSystem.u_detected when FAlarm.present;
    edge AlarmSystem.u_time_elapsed when FAlarm.present;
```

```
    edge InteriorMonitoring.u_detected when FInterMon.present;
    edge InteriorMonitoring.u_clear when FInterMon.present;
end

plant automaton PRESENCE_CONTROLLED_AS:
location : initial; marked;
    edge AlarmSystem.c_on when FAlarm.present;
    edge AlarmSystem.c_off when FAlarm.present;
    edge AlarmSystem.c_deactivated when FAlarm.present;
    edge AlarmSystem.c_IM_detected when FInterMon.present;
    edge InteriorMonitoring.c_on when FInterMon.present;
    edge InteriorMonitoring.c_off when FInterMon.present;
end

requirement AlarmSystem.c_on needs Key_lock.Locked or RCK_CLS.Locked;//R47
requirement AlarmSystem.c_off needs Key_lock.Unlocked or RCK_CLS.Unlocked;//R48
requirement AlarmSystem.c_deactivated needs Key_lock.Unlocked or
               RCK_CLS.Unlocked;//R49
requirement AlarmSystem.c_IM_detected needs InteriorMonitoring.Detected;//R50
requirement InteriorMonitoring.c_off needs Key_lock.Unlocked or
            RCK_CLS.Unlocked;//R51
requirement AlarmSystem.c_off_reconf needs not sys_valid;//R58
requirement InteriorMonitoring.c_off_reconf needs not sys_valid; //R59

plant automaton VALIDITY_UNCONTROLLED_AS :
location  : initial ; marked ;
        edge AlarmSystem.u_detected when sys_valid ;
        edge AlarmSystem.u_time_elapsed when sys_valid ;
        edge InteriorMonitoring.u_detected when sys_valid ;
        edge InteriorMonitoring.u_clear when sys_valid ;
end

plant automaton VALIDITY_CONTROLLED_AS :
location  : initial ; marked ;
        edge AlarmSystem.c_on when sys_valid ;
        edge AlarmSystem.c_off when sys_valid ;
        edge AlarmSystem.c_deactivated when sys_valid ;
        edge AlarmSystem.c_IM_detected when sys_valid ;
        edge InteriorMonitoring.c_on when sys_valid ;
        edge InteriorMonitoring.c_off when sys_valid ;
end

plant automaton Key_lock:
    uncontrollable u_lock, u_unlock;
    location Unlocked:
    initial;marked;
        edge u_lock goto Locked;
    location Locked:
        edge u_unlock goto Unlocked;
end

plant automaton CLS:
    controllable c_lock, c_unlock, c_unlock_reconf;
    location Unlocked:
    initial;marked;
        edge c_lock goto Locked;
    location Locked:
    initial;marked;
        edge c_unlock_reconf goto Unlocked;
        edge c_unlock goto Unlocked;
end

plant automaton VALIDITY_CONTROLLED_reconf :
location  : initial ; marked ;
        edge CLS.c_lock when sys_valid ;
        edge CLS.c_unlock when sys_valid ;
end
```

```
plant automaton Auto_lock:
    uncontrollable u_drive_lock, u_door_open_unlock;
    location Unlocked:
    initial;marked;
        edge u_drive_lock goto Locked;
    location Locked:
    initial;marked;
        edge u_door_open_unlock goto Unlocked;
end

plant automaton PRESENCE_UNCONTROLLED_CLS:
location : initial; marked;
    edge Key_lock.u_lock when FCLS.present;
    edge Key_lock.u_unlock when FCLS.present;
    edge Auto_lock.u_drive_lock when FAutoL.present;
    edge Auto_lock.u_door_open_unlock when FAutoL.present;
end

plant automaton PRESENCE_CONTROLLED_CLS:
location : initial; marked;
    edge CLS.c_lock when FCLS.present;
    edge CLS.c_unlock when FCLS.present;
end

requirement CLS.c_lock needs Key_lock.Locked or RCK_CLS.Locked
                or Auto_lock.Locked;//R52
requirement CLS.c_unlock needs Key_lock.Unlocked or RCK_CLS.Unlocked;//R53

plant automaton RCK_CLS:
    uncontrollable u_lock, u_unlock;
    location Unlocked:
    initial;marked;
        edge u_lock goto Locked;
    location Locked:
    initial;marked;
        edge u_unlock goto Unlocked;
end

plant automaton Safety_fcn:
    uncontrollable u_time_elapsed_lock, u_door_open;
    controllable c_locked, c_unlock;
    location Idle:
    initial;marked;
        edge c_unlock goto Unlocked_RCK;
    location Unlocked_RCK:
    marked;
        edge u_time_elapsed_lock goto AutoLocking;
        edge u_door_open goto Idle;
    location AutoLocking:
        edge CLS.c_lock goto Idle;
end

plant automaton RCK_PW:
    uncontrollable u_up, u_down, u_released;
    location Idle:
    initial;marked;
        edge u_up goto Up;
        edge u_down goto Down;
    location Up:
        edge u_released goto Idle;
    location Down:
        edge u_released goto Idle;
end

plant automaton RCK_EM:
    uncontrollable u_up, u_down, u_left, u_right, u_released;
    location Idle:marked;initial;
        edge u_up goto Up;
```

```
        edge u_down goto Down;
        edge u_left goto Left;
        edge u_right goto Right;
    location Up:
        edge u_released goto Idle;
    location Down:
        edge u_released goto Idle;
    location Left:
        edge u_released goto Idle;
    location Right:
        edge u_released goto Idle;
end

plant automaton PRESENCE_UNCONTROLLED_SAFETY_RCK:
location : initial; marked;
    edge Safety_fcn.u_time_elapsed_lock when FSafe.present;
    edge Safety_fcn.u_door_open when FSafe.present;
    edge RCK_CLS.u_lock when FCLS.present;
    edge RCK_CLS.u_unlock when FCLS.present;
    edge RCK_PW.u_up when FCtrAutoPW.present;
    edge RCK_PW.u_down when FCtrAutoPW.present;
    edge RCK_PW.u_released when FCtrAutoPW.present;
    edge RCK_EM.u_up when FAdjMir.present;
    edge RCK_EM.u_down when FAdjMir.present;
    edge RCK_EM.u_left when FAdjMir.present;
    edge RCK_EM.u_right when FAdjMir.present;
    edge RCK_EM.u_released when FAdjMir.present;
end

plant automaton PRESENCE_CONTROLLED_SAFETY_RCK:
location : initial; marked;
    edge Safety_fcn.c_unlock when FSafe.present;
    edge Safety_fcn.c_locked when FSafe.present;
end

requirement Safety_fcn.c_unlock needs RCK_CLS.Unlocked;//R54
```