Hierarchical Featured State Machines

Vanderson Hafemann Fragal[1,*], Adenilso Simao[1], Mohammad Reza Mousavi[2,**]

Abstract

Variants of the Finite State Machine (FSM) model have been extensively used to describe the behavior of reactive systems. In particular, several model-based testing techniques have been developed to support test case generation from FSMs and test case execution. Most of such techniques require several validation properties to hold for the underlying test models. Featured Finite State Machine (FFSM) is an extension of the FSM model proposed in our earlier publication that represents the abstract behavior of an entire Software Product Line (SPL). By validating an FFSM, we validate all valid products configurations of the SPL looking forward configurable test suites. However, modeling a large SPL using flat FFSMs may lead to a huge and hard-to-maintain specification. In this paper, we propose an extension of the FFSM model, named Hierarchical Featured State Machine (HFSM). Inspired by Statecharts and UML state machines, we introduce the HFSM model to improve model readability by grouping up FFSM conditional states and transitions into abstracted entities. Our ultimate goal is to use HFSMs as test models. To this end, we first define some syntactic and semantical validation criteria for HFSMs as prerequisites for using them as test models. Moreover, we implement an adapted graphical Eclipse-based editor from the Yakindu Project for modeling, derivation, and checking feature-oriented properties using Satisfiability Modulo Theory (SMT) solver tools. We investigate the applicability of our approach by applying it to an HFSM for a realistic case study (the Body Comfort System). The results indicate that HFSMs can be used to compactly represent and efficiently validate the behavior of parallel components in SPLs.

Keywords: Model Validation, Software Product Line, Featured Finite State Machine, Hierarchical Featured Finite State Machine.

*The work of V. Hafemann has been partially supported by the Science Without Borders project number 201694/2015-8.

**The work of M. R. Mousavi has been partially supported by the Swedish Research Council award number: 621-2014-5057 and the Swedish Knowledge Foundation project number 20140312.

Email addresses: vanderson.fragal@gmail.com (Vanderson Hafemann Fragal), adenilso@icmc.usp.br (Adenilso Simao), mm789@le.ac.uk (Mohammad Reza Mousavi)

[1]Institute of Math. and Computer Sciences - ICMC, University of Sï¿œo Paulo, Brazil.

[2]Department of Informatics, University of Leicester, UK.

1. Introduction

In the face of the increasing complexity, software industries moved from craftsmanship to industrialization, where components are customized and assembled to produce similar products with low cost and satisfying several customer demands [1].

In Software Product Line Engineering (SPLE), a family of related products (a Software Product Line - SPL) is built out of a common set of core assets, thus reducing development costs for each product [2]. In SPLE, products are built step-by-step, by incrementally adding or removing functionalities.

Similar to the development of single systems, the SPLE process also has several activities that are executed to ensure software quality. Testing (including verification and validation) is an example of such activities. Despite the systematic software artifact reuse that increases productivity, new challenges arise in testing activities for SPLE [3, 4].

Testing activities represent a large share of overall project costs [5] and are even more challenging in SPLE than in engineering single systems [6]. Unfortunately, several domains, such as embedded and safety-critical systems, do not strictly follow development standards (due to high test costs) to efficiently test several product configurations in a systematic manner. For example, the standard ISO 26262[3] for safety-critical automotive software recommends model-based techniques for various product configurations for the highest level of safety integrity.

Finite State Machines (FSMs) and their variants have been extensively used as a fundamental semantic model for various behavioral specification languages. In particular, several test case generation techniques have been developed for hardware and software testing based on FSMs; an overview of these techniques can be found in [7, 8, 9]. All FSM-based testing techniques require the underlying test models to satisfy some basic validation criteria such as connectedness and minimality.

There are recent attempts [10, 11] to extend the FSM-based testing techniques to SPLs, mostly using the delta-oriented approach to SPL modeling. We proposed Featured Finite State Machines (FFSMs) in the conference publication of this paper [12] with focus on the basic test model validation criteria for SPLs at the family-wide level. However, the scalability problem is the main issue of using the FFSM model for large and complex systems. Such a problem makes model analysis costly to execute and it also leads to a hard-to-maintain test model. To improve scalability in modeling, we propose in this paper an extension of FFSMs named Hierarchical Featured State Machines (HFSMs). The HFSM model is inspired by Statecharts [13], and its syntax is validated by checking well-formed states and transitions. To define a formal semantics for HFSMs and enable its formal analysis, we define a transformation from HFSMs to FFSMs.

Figure 1 shows an overview of the SPL validation workflow for HFSM with artifact dependencies represented by dashed arrows. In domain engineering, we

[3] `https://www.iso.org/standard/43464.html`

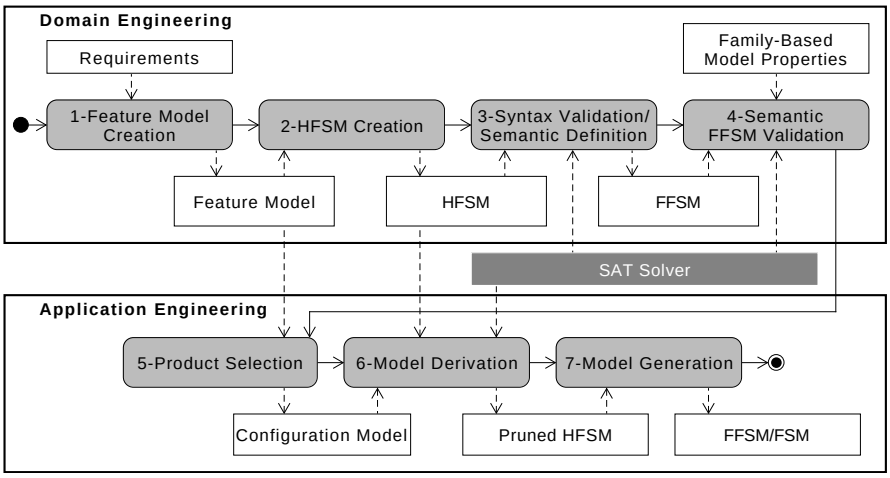

Figure 1: SPL validation workflow using HFSMs.

generate reusable and configurable artifacts while in application engineering those artifacts can be configured for a set of product configurations.

The main contributions of this paper are: (i) extending the FFSM model to HFSM. The introduction of FFSMs is also among our contributions, but due to our focus on hierarchy, we dispense with presenting the detailed results about the basic formalism which can be found in [12]; (ii) implementing a support tool for modeling, validation, and derivation of HFSMs using the Eclipse platform, Java language, and the Z3 solver tool [14]; and (iii) investigating the HFSM applicability and validation using a case study. The case study is from the automotive domain concerning the Body Comfort System [15]. The results indicate that HFSMs can be used to compactly represent and to efficiently validate the behavior of parallel components in SPLs.

The remainder of this paper is organized as follows. Section 2 presents some preliminary notions and concepts. Section 3 introduces the FFSM formalism. Section 4 introduces the HFSM formalism with detailed syntax and semantics. Section 5 presents the supporting tool for HFSM modeling, validation, and derivation. Section 6 illustrates a case study with an industrially-inspired HFSM. Section 7 provides an overview of the related work and a comparison among the relevant approaches in the literature. Finally, Section 8 concludes the paper and presents the directions of our future work.

2. Background

This section presents the basic concepts and definitions regarding SPL that we are going to use throughout the rest of the paper.

2.1. Feature Diagram

A *feature* is a prominent or distinctive user-visible aspect, quality, or characteristic of a software or a system [16]. A *feature diagram* [17] is a notational convention to describe constraint-based feature relations. The basic feature relations are mandatory, optional, inclusive-or (or), exclusive-or (alternative), include, and exclude [18]. A noteworthy feature modeling method for specifying feature constraints is the Feature-Oriented Domain Analysis (FODA) [16]. Subsequent feature modeling methods, such as the Orthogonal Variability Model (OVM) [19], extend the FODA to add new dependency relations.

Example 1. The Arcade Game Maker (AGM) [20] produces arcade games with different game rules. Figure 2 shows the feature diagram of AGM. There are three alternative features for the game rule (*Brickles*, *Pong*, and *Bowling*) and one optional feature (*Save*) to save the game. For each product, one and only one alternative feature must be selected, and the optional feature is left open for selection.

A feature diagram is developed in the domain engineering and is used as input to the application engineering level, where it is instantiated by a configuration model. A *configuration model* allows for selecting features to specify a product, and it is useful for integrating components in the product configuration process. The product configuration process (*binding*) derives a specific product using the reusable SPL architecture and a configuration model with selected features.

2.2. Feature Constraint

In general, due to the dependencies and constraints on feature combinations, only some combinations of features are valid. Assume a set of features F of a feature model. The set of all *valid products* P of an SPL is a subset of feature combinations from the power set $\mathcal{P}(F)$ that satisfies the constraints specified by the feature model [21].

A *feature constraint* χ is a propositional formula that interprets the elements of the feature set F as propositional variables. The set of all feature constraints is denoted by $B(F)$. The relation between features and their constraints can be modeled by a feature diagram and can be extracted as a logical formula using the formal semantics of feature diagrams [17]. A *product configuration* $\rho \in B(F)$

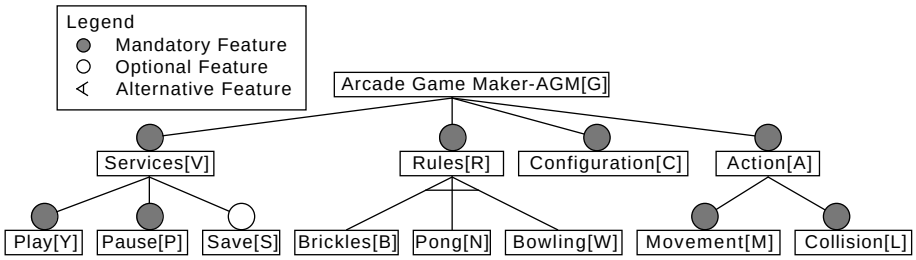

Figure 2: AGM Feature Diagram (adapted from [20]).

of a product $p \in P$ is a feature constraint of the form $\rho = (\bigwedge_{f \in p} f) \wedge (\bigwedge_{f \notin p} \neg f)$, i.e., the conjunction of all features present in p and the conjunction of the negation of all features absent from p. The set $\Lambda \subseteq B(F)$ denotes all valid product configurations of the SPL. Given a feature constraint $\chi \in B(F)$, a product configuration $\rho \in \Lambda$ *satisfies* χ, denoted by $\rho \vDash \chi$, if and only if feature constraint $\rho \wedge \chi$ is satisfiable.

Example 2. Given the feature diagram of Figure 2 the extracted feature set is $F = \{G, V, R, C, A, Y, P, S, B, N, W, M, L\}$ where $O = \{G, V, R, C, A, Y, P, M, L\} \subseteq F$ is the subset of mandatory features. The extracted feature constraint that represents the set of all valid products is:

$$\chi = ((\bigwedge_{f \in O} f) \wedge (S \implies V) \wedge (B \vee N \vee W) \wedge$$
$$\neg(B \wedge N) \wedge \neg(B \wedge W) \wedge \neg(N \wedge W)) \in B(F)$$

There are only six product configurations that satisfy χ, namely, those specified below:

$\rho_1 = (\bigwedge_{f \in O} f) \wedge B \wedge \neg N \wedge \neg W \wedge \neg S$, $\rho_2 = (\bigwedge_{f \in O} f) \wedge B \wedge \neg N \wedge \neg W \wedge S$,
$\rho_3 = (\bigwedge_{f \in O} f) \wedge \neg B \wedge N \wedge \neg W \wedge \neg S$, $\rho_4 = (\bigwedge_{f \in O} f) \wedge \neg B \wedge N \wedge \neg W \wedge S$,
$\rho_5 = (\bigwedge_{f \in O} f) \wedge \neg B \wedge \neg N \wedge W \wedge \neg S$, $\rho_6 = (\bigwedge_{f \in O} f) \wedge \neg B \wedge \neg N \wedge W \wedge S$.

In our textual notation, the logical operators on feature constraints are denoted by && (and), || (or), and ! (not).

2.3. Feature Model

A *Feature model* [2] specifies the structure of an SPL in terms of its feature and feature constraints. It can serve as the underlying structural model for other formalisms modeling the behavior of an SPL, e.g., for the purpose of testing.

Definition 2.1. A feature model FM is a tuple (F, χ), where F is the set of features and χ is the feature constraint.

3. Featured Finite State Machines

Variants of the FSM model have been extensively used to describe the behavior of different domains. In particular, several model-based testing approaches [3] have been developed to support test design and execution for SPLs. In this paper, we introduce Featured Finite State Machines (FFSMs) [12] and use them as the semantics model for our hierarchical model HFSM. The FFSM can represent the behavior of SPLs using a single model where product FSM properties are defined and checked in a family-wide level.

3.1. Basic Definitions

An FFSM combines states and transitions with feature constraints. The syntax of FFSMs is defined as follows.

Definition 3.1. An FFSM is a 6-tuple $(FM, C, c_0, Y, O, \Gamma)$, where

1. $FM = (F, \chi)$ is a feature model (Definition 2.1),
2. $C \subseteq S \times B(F)$ is a finite set of *conditional states*, where S is a finite set of state labels, $B(F)$ is the set of all feature constraints, such that C satisfies the following condition:

$$\forall_{(s,\varphi)\in C} \bullet \exists_{\rho\in\Lambda} \bullet \rho \vDash \varphi$$

3. $c_0 = (s_0, true) \in C$ is the *initial conditional state*,
4. $Y \subseteq I \times B(F)$ is a finite set of *conditional inputs*, where I is the set of input labels,
5. O is a finite set of *outputs*,
6. $\Gamma \subseteq C \times Y \times O \times C$ is the set of conditional transitions satisfying the following condition:

$$\forall_{((s,\varphi),(x,\varphi''),o,(s',\varphi'))\in\Gamma} \bullet \exists_{\rho\in\Lambda} \bullet \rho \vDash (\varphi \wedge \varphi' \wedge \varphi'')$$

The above-given two conditions ensure that every conditional state and every conditional transition is present in at least one valid product of the SPL. A conditional state $c = (s, \varphi) \in C$ is alternatively denoted by $s(\varphi)$. A conditional transition from conditional state c to c' with conditional input $y = x(\varphi'')$ and output o $t = (c, y, o, c')$ is alternatively denoted by $x(\varphi'')/o$ or $c \xrightarrow[o]{y} c'$. Omitted feature conditions mean that the condition is $true$, i.e., state s is equivalent to $(s, true) \in C$, and $\xrightarrow[o]{x}$ is equivalent to $\overset{(x,true)}{\underset{o}{\rightarrow}}$.

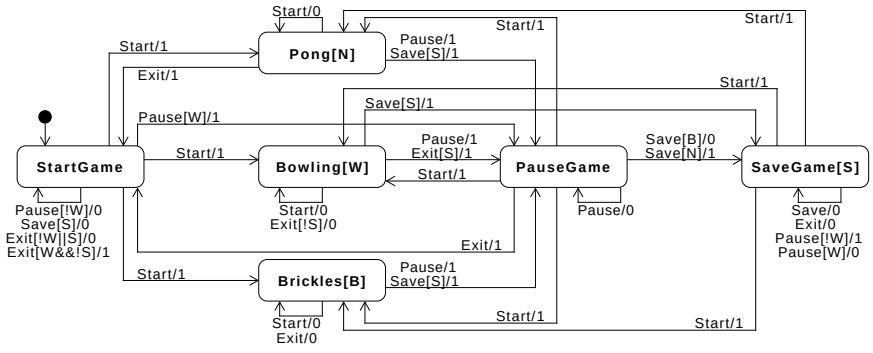

Figure 3: FFSM for the AGM SPL.

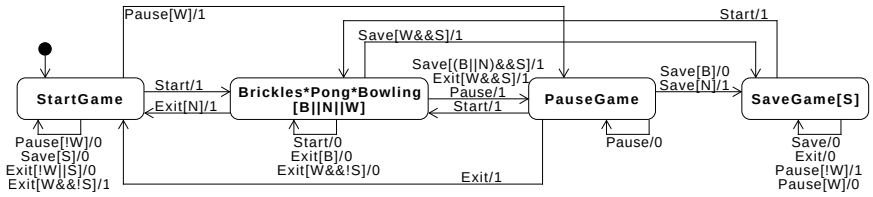

Figure 4: Alternative FFSM for the AGM SPL.

Example 3. Figure 3 shows the FFSM for the AGM SPL. Transitions have abstract input and output events. For inputs, their names are self-explanatory and for outputs, 0 denoted "nothing" and 1 denotes "beep". Feature constraints are put in brackets for conditional states and transitions. Feature constraints in conditional states constraint the feature constraints of all their incoming and outgoing conditional transitions. Both conditional transitions $StartGame \overset{(Exit,(W\&\&!S))}{\underset{1}{\rightarrow}} StartGame$ and $Bowling(W) \overset{(Exit,(!S))}{\underset{0}{\rightarrow}} Bowling(W)$ only exist in one product which have the W feature and do not have the S feature. Figure 4 shows an alternative FFSM for AGM which represents a mutually exclusive behavior combining *Brickles*, *Pong*, and *Bowling* in a single conditional state. States with alternative feature constraints are merged into a single state with a compound name (combining original names with "*") and with the disjunction of their corresponding constraints.

3.2. Model Derivation

To perform model derivation using an FFSM we use a specific feature constraint. To use a simplified feature constraint, we first need to define the equivalence relation between feature constraints for a given feature model FM.

Definition 3.2. Given an $FF = (FM, C, c_0, Y, O, \Gamma)$, where $FM = (F, \chi)$, a feature constraint ω_a is a *conditional prefix* of ω_b if: (i) there exists a valid configuration that satisfies both feature constraints, i.e. $\exists_{\rho \in \Lambda} \bullet \rho \vDash (\omega_a \wedge \omega_b)$; and (ii) the subset of configurations $\Lambda_a \subseteq \Lambda$ that satisfy ω_a is a subset of configurations $\Lambda_b \subseteq \Lambda$ that satisfy ω_b, i.e., $\Lambda_a \subseteq \Lambda_b$. When $\Lambda_a \subseteq \Lambda_b$ and $\Lambda_b \subseteq \Lambda_a$ we say that ω_a and ω_b are *equivalent* under FM.

Next, we define a model derivation operator, reminiscent of the operator in [22, 23], that is parameterized by feature constraints. Given a feature constraint, the product derivation operator reduces an FFSM into an FSM representing a selection of products.

Definition 3.3. Given an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$, and a valid product configuration $\rho \in \Lambda$ (or equivalently, a feature constraint ϕ corresponding to a product configuration) for a feature model FM (Definition 3.2), the product derivation operator Δ_ρ : induces an FSM $\Delta_\rho(FF) = (S, s_0, I, O, T)$, where:

1. $S = \{s | (s, \varphi) \in C \wedge \rho \vDash (\varphi \wedge \phi)\}$ is the set of states;

2. $s_0 = s$, $c_0 = (s, \varphi) \in C$ is the initial state;
3. $T = \{(s, x, o, s') | (s, \varphi) \overset{(x, \varphi'')}{\underset{o}{\rightarrow}} (s', \varphi') \in \Gamma \wedge \rho \vDash (\varphi \wedge \varphi' \wedge \varphi'' \wedge \phi)\}$ is the set of transitions.

By abusing the same notation, we also define how to reduce an FFSM into another FFSM that specifies a set of products (i.e., a product sub-line).

Definition 3.4. Given a feature constraint $\phi \in B(F)$ and an FFSM $FF = (FM, C, c_0, Y, O, \Gamma)$, if at least one product configuration $\rho \in \Lambda$ satisfies ϕ, i.e., $\exists_{\rho \in \Lambda} \bullet \rho \vDash \phi$, then the product derivation operator $\nabla_\phi : B(F) \rightarrow FFSM$ induces a reduced FFSM $\nabla_\phi(FF) = (FM', C', c_0, Y', O, \Gamma')$ comprising only those components (i.e., conditional states and transitions) that satisfy ϕ.

3.3. Validation Properties

To adopt FFSMs as test models, we need to validate the product-line-based specification with properties used for FSMs. These include the notions determinism, initially-connectedness and minimality initially defined for FSMs. In [12], we have lifted these notions to their featured counterparts, where the constraints for each of these notions involve the feature constraints on the considered states and transitions. Subsequently, in the following theorems, we showed and prove theat the FFSM-based properties coincide with the corresponding properties for all their valid FSM products.

Theorem 1. *An FFSM FF is deterministic if and only if all derived product FSMs $\Delta_\phi(FF)$ are deterministic.*

Theorem 2. *An FFSM is initially connected if and only if all derived product FSMs $\Delta_\phi(FF)$ are initially connected.*

Theorem 3. *An FFSM is minimal if and only if all derived product FSMs $\Delta_\phi(FF)$ are minimal.*

Further details (including formal definitions and proofs) on model validation properties for the FFSM model can be found in [12]. Since the focus of the present paper is on hierarchical models, we dispense with specifying these details here.

4. Hierarchical Featured State Machine

Inspired by Harel's Statecharts [13], several hierarchical state machine formalisms were defined to specify the behavioral aspects of reactive systems and extended to object-oriented software development methodologies such as the Unified Modeling Language (UML) [24]. The states represented in the hierarchical model can be simple states or contain an entire state machine. Systems can be specified by a stepwise refinement and visualized in different levels of granularity at the cost of complex syntax and semantics.

We introduced the Featured Finite State Machine (FFSM) formalism in [12] to extend the Finite State Machine (FSM) formalism to the Software Product

Line (SPL) context. Due to scalability problems, e.g., the combinatorial explosion while producing flat models of parallel behavior, large SPLs are hard to model and maintain using the FFSM model.

In this paper, we present the Hierarchical Featured State Machine (HFSM) formalism that extends the FFSM model by including hierarchy. The HFSM model improves model readability by grouping up FFSM conditional states and transitions into an abstracted view, which provides a better solution for modeling SPL-based models, e.g., as test models in a model-based testing (MBT) approach. (We refer to [25], where we lay the foundations of model-based testing using FFSMs.) Furthermore, similar to FFSMs, an HFSM model can also be projected into subsets of product configurations.

In our approach, we use the HFSM as a front-end for modeling and syntax check, while the semantics of an HFSM is represented by an FFSM. The syntax check verifies the well-formed state and transition structure. The semantic check verifies properties such as determinism, initially connectedness, and minimality at the SPL level. These properties are required for test-case generation methods guaranteeing full fault coverage (see [26, 25]). However, one of the main issues of lifting the FFSM formalism to HFSM is how to compose orthogonal regions that we also explain in this paper.

Next, we present the detailed syntax and semantics of HFSMs (based on [27]) followed by a support tool for syntax and semantic checks, and then a case study.

4.1. Syntax

There are many variations of hierarchical FSMs, upon which we can base our hierarchical extension of FFSM. Due to the popularity of UML and our prior experience with the formal semantics and verification of UML state diagrams, we based our definitions on those that form the basis of UML state diagrams [24], namely [13] and [27]. An HFSM comprises states that may have a further internal structure (hierarchy) and transitions among them. States and transitions have feature constraints that must be satisfied according to a feature model. The following definitions are based on the corresponding definitions in [13] and [27].

Definition 4.1. A *Hierarchical Featured State Machine* $HFSM$ is represented by a 5-tuple $(FM, \Upsilon, I, O, \mathcal{T})$, where:

1. $FM = (F, \chi)$ is a feature model (Definition 2.1),
2. Υ is a well-formed state structure,
3. I is a set of inputs events,
4. O is a set of outputs events,
5. $\mathcal{T}$ is a set of well-formed transitions.

A well-formed state structure (inspired by the same notion in [13]) is a tree that represent a valid state hierarchy. The set of well-formed transitions contains valid transitions that connect sets of states in the hierarchy. Input and output sets represent the observable behavior of the machine. Not every state structure or transition is valid, e.g., a simple state cannot have sub-states. Thus, some criteria are required to represent well-formed state structures and transitions. We

present the formal definitions of well-formed states and transitions in subsections 4.1.1 and 4.1.2, respectively.

4.1.1. Well-formed state structure

In this section, we define the syntactic state structure of HFSMs, their restrictions, and well-formedness conditions. First, we define the state structure as follows.

Definition 4.2. A *state structure* Υ is defined by a 6-tuple $(S, root, default, sub, type, feature)$, where:

1. S is a finite set of *states*;
2. $root \in S$ is the *root* state of the state structure;
3. $default : S \rightarrow \{true, false\}$, is a total function determining whether a state is *default* or not;
4. $sub : S \rightarrow \mathcal{P}(S)$ is a total function defining for each state, the set of its *sub-states*;
5. $type : S \rightarrow \{simple, compOr, compAnd, region\}$ is a total function determining for each state whether it is a *simple* (i.e., has no sub-states), a *compOr* (i.e., is composite and has only one default sub-state), a *compAnd* (i.e., is composite and has more than one default sub-state), or a *region* state (i.e., is a sub-state of a composite state);
6. $feature : S \rightarrow B(F)$, is a total function determining the feature constraint of a state.

A tree of conditional states represents the state structure. The root state is the unique state that encapsulates all states in the state structure. The sub-states of a state are their children in the tree structure. Composite states only have regions as their sub-states; sub-states of a region may be of any type but region (these constraints are to be enforced by our well-formedness conditions). For simplicity, we do not treat history, deep-history, join/forks, and entry/exit points connections in this paper. Albeit important, these concepts require a more elaborate semantics and are deferred to future work.

Example 4. Figure 5 shows the HFSM for the AGM SPL [20]. Unlike Example 3, we have given more structure to the alternative states (i.e., Brickles, Pong, and Bowling). We have put the alternative states in region states (in order to model independent and potentially parallel behavior). However, due to feature constraints applied to region states (i.e., B for R1, N for R2, and W for R3), they are exclusive and will never be composed in parallel. Alternatively, we could represent these alternative states by combining them as we did in Figure 4. However, we may have more than one state per alternative feature (i.e. 2 states with constraint B and 3 states with constraint N). Thus, as a design guideline, one may note that such groups of alternative states are better grouped by regions.

Regarding transitions, we can model transitions that start and finish in the same state (self-loop transitions) inside the state. By modeling self-loop

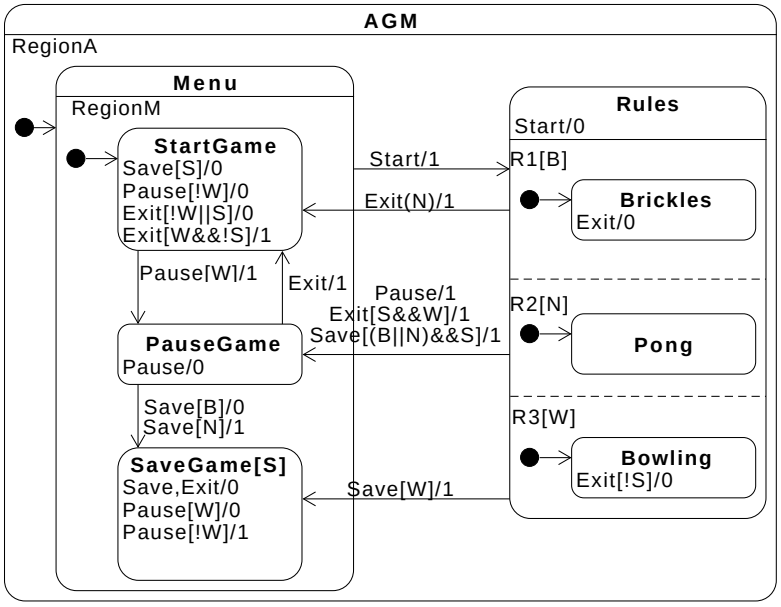

Figure 5: HFSM for AGM.

transitions inside a state, we can combine those which have equivalent feature constraints, e.g., instead of modeling inside $SaveGame[S]$ two self-loop transitions $Save/0$ and $Exit/0$, we can model $Save, Exit/0$.

A default state is a state that is automatically activated once the super-state (parent) is activated or it is the root itself. Default states are only activated when the transition does not explicitly target a state within the region. For example, the state *StartGame* is not activated after taking the transition from *Rules* to *SaveGame* using the input *Save*, which activates $Menu$ and $RegionM$. In Figure 5, the default states, besides region states, are represented by default connectors, i.e., a filled circle with an outgoing arrow pointing towards the default state.

Example 5. Figure 6 shows the state structure of the HFSM presented in Figure 5, where AGM is the root state, *Rules* is the compAnd state, all states but *PauseGame*, *SaveGame*, and *Rules* are default states and the sub-states of the *RegionA* state is $sub(RegionA) = \{Menu, Rules\}$.

The state structure is generic and allows for inconsistent (non-well-formed) specifications, e.g., sub-states of a region state being region states themselves. To define the concept of well-formed state structures, we need several auxiliary functions, defined below. First, we define hierarchy using descendants and ancestors. Descendants are recursively defined below to include the state itself, its sub-states, its sub-sub-states and so forth.

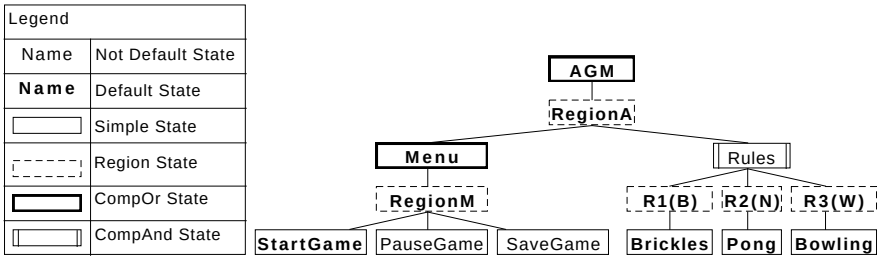

Figure 6: State structure for AGM HFSM.

Definition 4.3. Given a state $s \in S$, the set of *descendants* of s, denoted by $desc(s)$ is the smallest set satisfying the following two properties.

1. $s \in desc(s)$; and
2. $\forall_{a \in S} \bullet a \in desc(s) \implies \forall_{b \in sub(a)} \bullet b \in desc(s)$.

For a set of states $S' \subseteq S$, we define the notion of descendant by $Desc(S') = \bigcup_{s' \in S'} desc(s')$. Super-states and ancestors are the inverses of sub-states and descendants, respectively, and are defined as follows.

Definition 4.4. Given a state $s \in S$, a *super-state* (parent) of s is defined by a function $super : S \to S$ where $\forall_{s,s' \in S} \bullet s \in sub(s') \iff s' = super(s)$. Moreover, *ancestors* of s are denoted by $anc(s)$ and satisfy the following condition.

$$\forall_{s,s' \in S} \bullet s \in anc(s') \iff s' \in desc(s)$$

For a set of states $S' \subseteq S$, we define the notion of ancestor by $Anc(S') = \bigcap_{s' \in S'} anc(s')$.

Example 6. Following Figure 6 some descendants and ancestors in the state structure of the HFSM are:

- $desc(Menu) = \{Menu, RegionM, StartGame, PauseGame, SaveGame\}$;
- $anc(PauseGame) = \{PauseGame, RegionM, Menu, RegionA, AGM\}$;
- $Desc(\{RegionM, R1\}) = \{RegionM, R1, StartGame, PauseGame, SaveGame, Brickles\}$; and
- $Anc(\{PauseGame, Brickles\}) = \{RegionA, AGM\}$.

In the FFSM representation, the constraints of a conditional state further restrict the feature constraints of its incoming and outgoing transitions. In other words, transitions "inherit" the feature constraints of their source and target states. The feature constraint of a transition is composed and checked using the constraints of the involved states and the constraint of the transition itself.

Similarly, in the HFSM representation, the feature constraint of a state also applies to its descendants. Thus, we define below the composition of constraints that are used for checking states in the hierarchy.

Definition 4.5. Given a set of states $S' \subseteq S$, the *state feature composition*, denoted by $fcomp : \mathcal{P}(S') \rightarrow B(F)$ is the conjunction of feature constraints of S':

$$fcomp(S') = \bigwedge_{s \in S'} feature(s)$$

To check the feature constraint of an HFSM state $s \in S$, we compose the feature constraints of all ancestors of s (i.e., set $anc(s)$), denoted by $fcomp(anc(s))$.

Finally, we define the concept of well-formedness of state structures extended with feature constraints (inspired by the corresponding restrictions in [13] and [27]).

Definition 4.6. The state structure $\Upsilon = (S, root, default, sub, type, feature)$ is *well-formed*, when:

1. Simple-states have no sub-states.

   $$\forall_{s \in S} \bullet type(s) = simple \implies sub(s) = \varnothing$$

   This is a basic assumption as simple states should not have any further structure inside them.
2. All nodes, besides root, have a unique super-state.

   $$\forall_{s \in S \setminus \{root\}} \bullet \exists!_{s' \in S} \bullet s' = super(s)$$

   This is to make sure that the chain of ancestors reaches an end in the root and also to make sure that every internal behavior is encapsulated by regions states.
3. The descendant relation is asymmetric.

   $$\forall_{s \in S} \bullet s \notin sub(s) \wedge \forall_{s' \in S} \bullet s \in desc(s') \setminus \{s'\} \implies s' \notin desc(s) \setminus \{s\}$$

   This constraint disallows loops in the chain of ancestors and descendants.
4. The single sub-state of a compOr state is a region state.

   $$\forall_{s \in S} \bullet type(s) = compOr \implies \exists!_{s' \in S} \bullet s' \in sub(s) \wedge type(s') = region$$

   This constraint will be used to enforce that upon entering a compOr state, we enter the single region sub-state that represent its inner machine.
5. All sub-states of compAnd states are region states.

   $$\forall_{s \in S} \bullet type(s) = compAnd \implies |sub(s)| > 1 \wedge \forall_{s' \in sub(s)} \bullet type(s') = region$$

By definition, a compAnd state has more than one region sub-state. Upon entering a compAnd state, we enter in all region sub-states to support parallelism.

6. Region states are default states, their sub-states must not be region states, and only one of their sub-states is default.

$$\forall_{s \in S} \bullet type(s) = region \implies default(s) \wedge \forall_{s' \in sub(s)} \bullet type(s') \neq region \wedge \exists!_{s'' \in sub(s)} \bullet default(s'')$$

Upon entering a composite state, we automatically enter a region state that contains an inner machine. Region states are special states used to represent inner machines, and only composite states can have region states as sub-states. One sub-state of the region state must be default to represent the "initial state" of this region.

7. Root is of type compOr and all states are descendants of the root.

$$\exists!_{s \in S} \bullet s = root \wedge type(s) = compOr \wedge \forall_{s' \in S} \bullet s' \in desc(s)$$

The root state is the common ancestor state of all states in the state structure.

8. The feature constraint of every state is satisfied by at least one product (Definition 4.5).

$$\forall_{s \in S} \bullet \exists_{\rho \in \Lambda} \bullet \rho \vDash fcomp(anc(s))$$

A state has a valid feature constraint based on its ancestors. A simple state inherits all feature constraints of $anc(s)$. Thinking about an executable machine, once we activate a state (e.g., via transition), we activate all the ancestors as well. Thus, the hierarchy must not contain branches that are not satisfied by any product configuration.

4.1.2. Well-formed transitions

A transition connects states using input events that trigger output events when its constraints are satisfied. The syntax of a transition connects a pair of states of the well-formed state structure. (In principle, transitions in a hierarchical model can connect multiple states, i.e., have more than one source state and one target state. We initially experimented with such models, but their formal treatment here will substantially clutter the presentation and add very little insight. Hence, we do not formally allow for such complex transitions in this paper.) Next, we formalize the definition of transitions and their well-formedness criteria.

Definition 4.7. A *transition* t in an HFSM is defined by a 5-tuple (a, i, ω, o, b), where:

1. $a \in S$ is the source state;
2. $i \in I$ is the input event;
3. $\omega \in B(F)$ is the feature constraint of the transition;
4. $o \in O$ is the output event;
5. $b \in S$ is the target state.

A transition $t = (a, i, \omega, o, b)$ is denoted by $a \overset{(i,\omega)}{\underset{o}{\rightarrow}} b$. The source state is where the transition begins, while the target state is where it ends. The input and output events are the observable behavior of the transition. The feature constraint ω is a specific condition of the transition. When we omit the feature constraint, $\omega = true$. The feature constraints of the source and target states restrict the entire feature constraint of a transition. Thus, we compose the feature constraints of all elements of the transition as follows.

Definition 4.8. Given a transition $t = (a, i, \omega, o, b)$, the *transition feature composition*, denoted by $tcomp : \mathcal{T} \rightarrow B(F)$ is the conjunction of feature constraints of elements of t:

$$tcomp(t) = fcomp(anc(a)) \wedge \omega \wedge fcomp(anc(b))$$

A transition exists in a product configuration only if it satisfies its transition feature composition. The definition of transition is generic and allows for inconsistent specifications, e.g., transitions whose target state is a region state. As mentioned before, region states are default states that can represent an entire machine. The basic metamodels (e.g., of statecharts or UML) does not allow for transitions to connect region states.

To validate a transition, we define several auxiliary functions, and finally, define the notion of a well-formed transition. Thus, we define the subset of states that can be used as source and target in a transition.

Definition 4.9. Given the set S of states, the set $R \subset S$ is the set of all *transition-relevant states*, when they are neither region states nor the root state:

$$\forall_{s \in S} \bullet type(s) \neq region \wedge s \neq root \implies s \in R$$

To define orthogonality between two states, i.e., when they can both be entered after a transition, we use the following notion of least common ancestor.

Definition 4.10. Given a subset of states $S' \subseteq S$, their *least common ancestor*, denoted by $lca(S')$, is the bottommost ancestor which contains all states of S'.

$$\exists!_{a \in Anc(S')} \bullet \forall_{b \in \bigcap\limits_{s' \in S'} anc(s')} \bullet b \in anc(a) \Leftrightarrow lca(S') = a$$

Example 7. Consider the HFSM depicted in Figure 5; the set of transition relevant states and the least common ancestor of *PauseGame* and *Brickles* states are:

- $R = \{Menu, Rules, StartGame, PauseGame, Brickles, Pong, Bowling\}$;
- $lca(\{PauseGame, Brickles\}) = RegionA$.

Orthogonal states can be active at the same time. Transitions with source states that are orthogonal synchronize on common inputs and activate their target states simultaneously.

Definition 4.11. Two states $s, s' \in S$ are *orthogonal* to each other when their least common ancestor (Definition 4.10) is a compAnd state.

Finally, we define the concept of well-formedness of transitions extended with feature constraints (inspired by the corresponding definition in [13]).

Definition 4.12. A transition $t = (a, i, \omega, o, b)$ is *well-formed* when the following conditions hold.

1. The source and target states are transition-relevant states (Definition 4.9).

   $$a, b \in R$$

   This is a basic assumption as no transition should use regions or root as their source or target states.
2. The source and target states are not orthogonal to each other (Definition 4.11).

   $$type(lca(\{a, b\})) \neq compAnd$$

   There should not be any transition involving parallel regions. Communication mechanisms trigger transitions in parallel regions.
3. The transition t has to be satisfied by at least one product (Definition 4.8).

   $$\exists_{\rho \in \Lambda} \bullet \rho \vDash tcomp(t)$$

   Every transition must exist in at least one product configuration.

4.2. Semantics

The semantics of HFSMs is represented by FFSMs. We transform an HFSM with a valid syntax into an FFSM to represent its semantic. We use an algorithm for model transformation that composes parallel regions in order to create a transformed set of state configurations (TSC), i.e., the flattened representation of states that are simultaneously active.

Conditional states of the semantic FFSM are obtained from TSC, which is derived after composing all compAnd states. Then, conditional transitions are derived by processing the exit and enter sets, i.e., the set of states left and entered due to a transition. Next, we present the required definitions for our semantics.

4.2.1. State configurations

To transform an HFSM into an FFSM, valid conditional states and transitions are required. The set of conditional states of the FFSM is defined using the HFSM well-formed state structure.

Definition 4.13. Given an HFSM $H = (FM, \Upsilon, I, O, \mathcal{T})$ with valid syntax, a *state configuration* (or just *configuration*) $SC \subset S$ is a maximal orthogonal set of simple states. i.e., $\forall_{s,s' \in SC} \bullet (type(s) = type(s') = simple) \wedge (lca(\{s, s'\}) = compAnd)$.

The root state and at least one leaf of the state structure tree are always active. The initial configuration is identified using all descendants of root that are default.

Definition 4.14. Given a state $s \in S$, the set $ddesc(s) \subseteq S$ of *default descendants* of s is the set of states defined by the following condition:

$$\forall_{s,s' \in S} \bullet \forall_{s'' \in (anc(s') \setminus anc(s))} \bullet (s' \in desc(s)) \wedge default(s'') \iff s' \in ddesc(s)$$

Informally, if a state is a descendant of s and its ancestors up to s are all default, then it is in $ddesc(s)$.

The initial conditional state of a semantic FFSM is the composition of the default simple states among the descendants of root ($ddesc(root)$). The state set $Init \subset S$ denotes the *initial configuration* of the semantic FFSM and is defined by the following constraint: $\forall_{s \in S} \bullet (s \in ddesc(root)) \wedge (type(s) = simple) \Leftrightarrow s \in Init$.

Example 8. The default descendants of the root state are $ddesc(root) = \{AGM, RegionA, Menu, RegionM, StartGame\}$, where $root = AGM$. The initial configuration of the state structure presented in Figure 6 is $Init = \{StartGame\}$. If the state *Rules* were a default state instead of $Menu$, then our initial configuration would have been $Init = \{Brickles, Pong, Bowling\}$. In the next section, we show how our flattening process maps a configuration into a single state, i.e., the initial state of the semantic FFSM becomes the flattened state $Brickles * Pong * Bowling$.

State configurations may include *simple* states of several orthogonal state regions. To transform HFSM states into FFSM conditional states, we need to identify every valid state configuration. Next, we define the composition of orthogonal states that derives FFSM conditional states.

4.2.2. Composition of Orthogonal States

The composition of orthogonal states (state composition) is a mechanism that allows a flat representation of the parallel execution of the HFSM. We identify valid (reachable) state configurations by combining pairs of regions of compAnd states. To transform a state configuration into an FFSM conditional state, we merge all parallel states of the state configuration and put them in the set of transformed state configurations TSC, i.e., $\{a, b, c\} \in SC$ implies $\{a * b * c\} \in TSC$.

To perform state composition, we use Algorithm 1. The algorithm recursively computes the set of all flattened states, by starting from the root and performing a depth-first traversal until it reaches a simple state as a leaf. Afterwards the backtracking starts and all such visited states are added to the TSC; then, while traversing parallel regions, all those regions whose constraints do not contradict the hitherto accumulated constraints are composed to form a flattened state.

This functionality is achieved through 3 main functions: COMPOSE_STATES, PAIRWISE_MERGE, and COMPOSE.

At the highest level, in Line 29, function COMPOSE_STATES is called with the initial parameters root and the empty set. The first parameter of COMPOSE_STATES denotes the current state and the second parameter denotes the set of visited (simple or flattened) states. We assume that the components of the HFSM (such as its root and state structure) are stored in a global variable and are accessible in all functions. Function COMPOSE_STATES traverses the state structure (by considering the sub-states of the current state) and calculates a set of consistent (i.e., states that can co-exist in a product) through calling PAIRWISE_MERGE.

In turn, PAIRWISE_MERGE, fetches the identified states in an arbitrary order and calculates a flattened state by composing them one-by-one by calling COMPOSE.

Finally, COMPOSE, combines the state names (by putting an asterisk between each two of them) and the feature constraints (by taking their conjunction) and combines the calculated state name and constraint into a conditional state.

On Line 1, we execute the recursive function COMPOSE_STATES using the *root* state and the empty set TSC as the initial input parameters. On Line 2, we check all substates of a state s, starting with *root*. On Line 3, we make a recursive call using substates of s and TSC as input and resulting in the updated TSC set. On Line 4, we check whether the substate is *simple* type. On Line 5, we add the simple state into TSC. On Line 6, we check whether the substate is *compAnd* type. On Line 7, we initialize the *init* conditional state using the *compAnd* state and the disjunction of the feature constraints of all regions involved. On Line 8, we use the power set of regions to decide whether a subset of regions (called a segment) can be composed or not. On Line 9, we check whether there is a product configuration that can have a specific combination of regions. A feature constraint is created and verified using a conjunction of: (i) the feature constraints of all region states in R; and (ii) the negation of the feature constraints of all regions out of R (i.e., $get\_full\_conj\_const$). On Line 10, we call the *pairwise_merge* function to perform the composition process using pairs of region states in R.

On Line 12, we execute the *pairwise_merge* function using R, *init* and TSC as input. On Line 13, we remove the first region from R and initialize *comp*. On Line 14, we get every other region to compose with *comp*. On Line 15, we call the *compose* function that merge regions *comp* and r. The resulting composition is stored in *comp* to be composed with the next region of r. On Line 16, we set *init* as the initial state of the resulting *comp* composed (flattened) region. On Line 17, we remove from TSC all descendants of the regions of R. This is required for the return of the recursion (upper *compAnd* states). On Line 18, we update TSC with the resulting flat region *comp*.

On Line 20, we compose all states and transitions of a pair of regions. On Line 23, 24, and 25, for each substate pair we create the conditional state using their name, the combined feature constraint, and store in *state*. On Line 26, we store the composed elements of the flat region, including their conditional states *state* and conditional transitions. Every transition that leaves or reaches s_1 and s_2 are combined for *state*.

Algorithm 1 Composition of orthogonal states.

```
1: function COMPOSE_STATES(s, TSC)
2:     for s' ∈ sub(s) do
3:         TSC = COMPOSE_STATES(s', TSC)
4:         if type(s') = simple then
5:             TSC = TSC ∪ s'
6:         if type(s') = compAnd then
7:             init = (s', get_disjunction_constraint(sub(s')))
8:             for R ∈ powerset(sub(s')) do
9:                 if sat(get_full_conj_const(R, sub(s'))) then
10:                    TSC = PAIRWISE_MERGE(R, init, TSC)
11:    return TSC
12: function PAIRWISE_MERGE(R, init, TSC)
13:    comp = R.get(0); R.remove(0)
14:    for r ∈ R do
15:        comp = COMPOSE(comp, r)
16:    link_segment(comp, init)
17:    TSC = TSC\{Desc(R)}
18:    TSC = TSC ∪ comp.get_simple_states()
19:    return TSC
20: function COMPOSE(comp, r)
21:    for s1 ∈ sub(comp) do
22:        for s2 ∈ sub(r) do
23:            name = s1 +'' ∗ '' + s2
24:            feature = fcomp(anc(s1)) +' && '' + fcomp(anc(s2))
25:            state = create_state(name, Z3_cond(feature))
26:            comp = comp ∪ merge_transitions(state, s1, s2)
27:    return comp
28: function MAIN
29:    return COMPOSE_STATES(root, ∅);
```

Example 9. Figure 7 and Figure 8 show how the semantics of a compAnd state vary in terms of the feature model.

Figure 7.(a) shows a part of an HFSM, which is a compAnd state comprising three regions. In two of these regions, there are simple states with outgoing transitions synchronizing on input a. We have three feature constraints for each parallel region: F1 for R1; F2 for R2 and F3 for R3. Figure 7.(b) shows the semantic FFSM using Feature Model A. We perform the composition in pairs of regions, i.e, region R1 with R2, then the result with R3. The order of choosing regions in the composition does not change the final behavior, but changes the composed name of the configuration and how inconsistent pairs are removed (how efficiently we prune the structure). Dashed states represent unreachable state configurations that are not transformed to conditional states. Due to synchronous transitions, our semantic FFSM has 4 out of 8 state configurations.

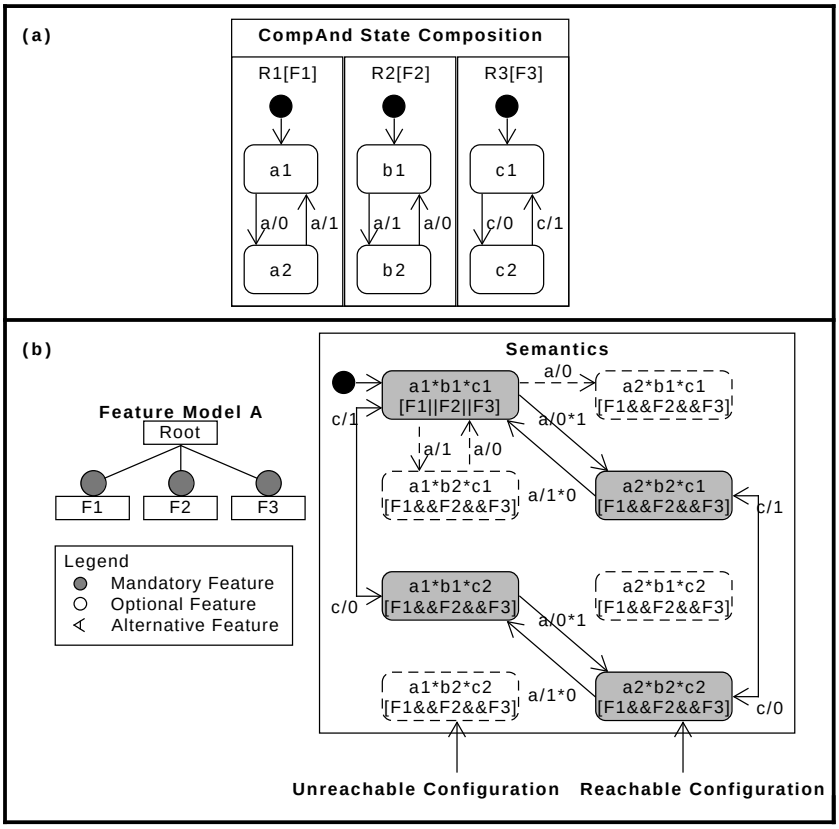

Figure 7: Semantic variation for composing a compAnd state (part 1).

In the HFSM, each region has a feature constraint that is logically equivalent to *true* based on feature model A; hence, our composition boils down to a simple flattening of the state structure.

Figure 8.(a) shows the semantic FFSM using Feature Model B. All pair of states are composed; since all features are optional, different combinations of feature presence or absence create different behavior segments. These combinations can lead to an exponential blow up in the number of valid state configurations in the worst case, in the number of features. Note that state configurations designated in gray are the same presented in Figure 7.(b). Note that there are groups of state configurations with the same feature constraint which belong to the same behavior segment. Thus, we have 7 (out of the total of 8) segments that represent the behavior of valid products. All segments are connected to the initial state configuration that uses as feature constraint the disjunction of all three region feature constraints.

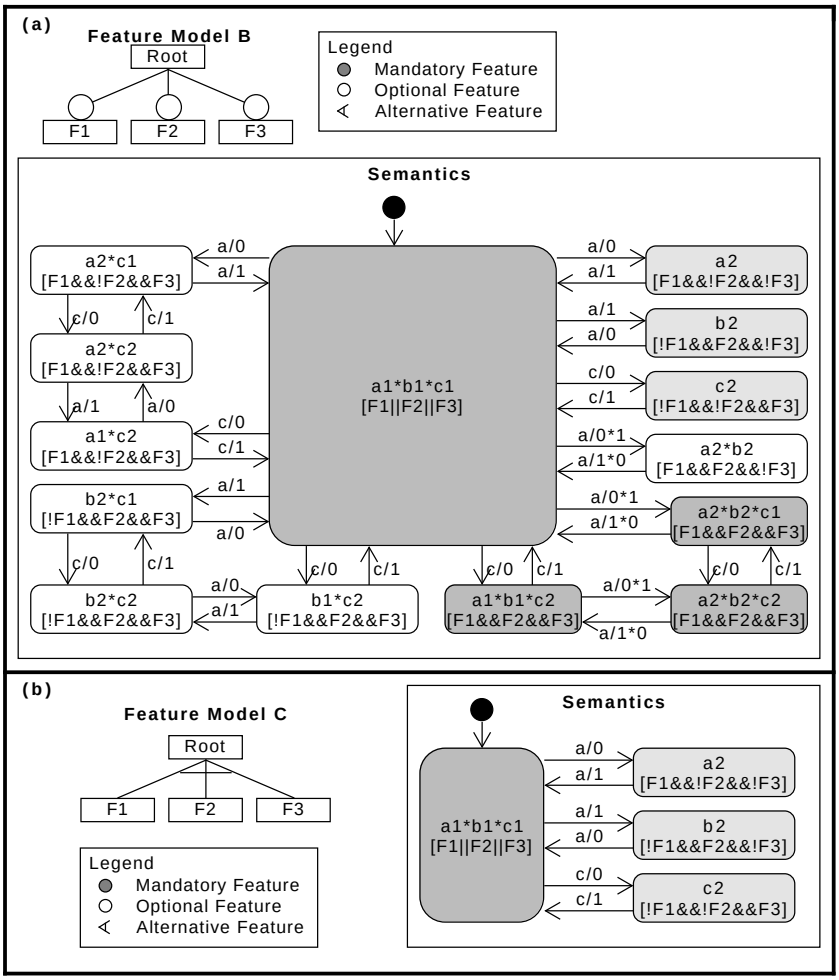

Figure 8: Semantic variation for composing a compAnd state (part 2).

Figure 8.(b) shows the semantic FFSM using Feature Model C. This scenario is where we greatly reduce the number of state configurations. In this case, the number of segments is the number of alternative features, i.e., 3 segments with one state configuration designated in light gray.

After executing Algorithm 1, there is a post-processing involved. Namely, we add the rest of state configurations that contain a single state into TSC, i.e., $\forall_{s \in S} \bullet (\forall_{s' \in anc(s)} \bullet type(s') \neq compAnd) \wedge type(s) = simple \implies s \in TSC$. Finally, for each transformed state configuration TSC, we create an FFSM conditional state.

Definition 4.15. Given a set of transformed state configurations (TSC) after executing the state composition (Algorithm 1), the set of conditional states C of an FFSM is defined by: $\forall_{s \in TSC} \bullet (s, fcomp(anc(s))) \in C$.

4.2.3. Creating conditional transitions

Semantically, transitions of the HFSM connect several states of the well-formed state structure; in particular, all ancestors of states of a state configuration are active.

Definition 4.16. Given a transition $t = (a, i, \omega, o, b)$ and a state configuration SC, the set of active states AC is all ancestors of SC, i.e., $\forall_{s \in SC} \bullet anc(s) \subseteq AC$.

Some states are activated/deactivated once a transition is performed. To identify those states we define the scope of a transition.

Definition 4.17. Given a transition $t = (a, i, \omega, o, b)$, the *scope* of t, denoted by $scope(t)$, is the lowest state in the state hierarchy that is a common ancestor of source and target states, i.e., $scope(t) = lca(\{a, b\})$ (Definition 4.10).

When a transition is performed, some states are deactivated while some are activated. The set of states that are activated after executing the transition is called enterSet.

Definition 4.18. Given a transition $t = (a, i, \omega, o, b)$, the *enterSet* of t is the set of states comprising: all default descendants of b and all ancestors of b except the ancestors of the scope, i.e, $enterSet = \{ddesc(b) \cup \{anc(b) \setminus anc(scope(t))\}\}$.

We do not include the ancestors of the scope of t because they are already active. The set of states that are deactivated after executing a transition is called exitSet, defined below.

Definition 4.19. Given a transition $t = (a, i, \omega, o, b)$ and a state configuration SC, the *exitSet* of t is the set of states comprising all descendants of a that are in SC and all ancestors of a except the ancestors of the scope, i.e., $exitSet = \{\{desc(a) \cap Anc(SC)\} \cup \{anc(a) \setminus anc(scope(t))\}\}$.

We do not include the ancestors of the scope of t because we do not want to exit those states only to enter them again. Once the transition is taken, the current state configuration may change. Simple states of the exitSet are removed and simple states of the enterSet are included in the new state configuration.

Example 10. Given the transition $t = (Menu, Start, true, 1, Rules)$ (Figure 5) and the current state configuration $SC_1 = \{PauseGame\}$, the scope of t is $scope(t) = RegionA$, the enterSet is {Rules,R1,R2,R3,Brickles,Pong,Bowling} ({Rules,R1*R2*R3,Brickles*Pong*Bowling} after orthogonal composition - section 4.2.2) and the exitSet is {Menu,RegionM,PauseGame}. After performing the transition, the new state configuration is $SC_2 = \{Brickles * Pong * Bowling\}$. In Algorithm 1, the *merge_transition* call combines the transitions involving different region states. Thus, the transitions that leave or reach *Brickles*, *Pong*, and *Bowling* are combined for the flattened state $Brickles * Pong * Bowling$. The single resulting FFSM conditional transition is created using the name of simple states and their feature constraints: $((PauseGame, fcomp(SC_1)), Start, true, 1, (Brickles * Pong * Bowling, fcomp(SC_2)))$.

For transition $t_2 = (Rules, Save, W, 1, SaveGame)$ and the current state configuration $SC_2 = \{Brickles * Pong * Bowling\}$, the scope of t_2 is $RegionA$, the enterSet is {Menu,RegionM,SaveGame} and the exitSet is {Rules,R1*R2*R3,Brickles*Pong*Bowling}. After taking the transition, the new state configuration is $SC_3 = \{SaveGame\}$. The single resulting FFSM conditional transition is: $((Brickles * Pong * Bowling, fcomp(SC_2)), Save, W, 1, (SaveGame, fcomp(SC_3)))$.

5. Tool Support

We implemented a tool[4] (under Eclipse Public Licence) that has a graphical editor based on the Eclipse platform. Our tool extends the Yakindu Project[5] (publicly available under Eclipse Public Licence) and is integrated with FeatureIDE [28] (publicly available under Lesser General Public Licence - LGPL), and the Z3 SMT Solver [14] (publicly available under MIT license) for constructing feature models and analyzing feature constraints, respectively. Our tool supports modeling, validation, and derivation of HFSM models with the aid of a semantic FFSM. Figure 9 shows how the HFSM of Figure 5 is modeled in our tool.

Our tool parses HFSMs provided in a simple textual format and generates a flattened version, after having analyzed the corresponding feature constraints according to Algorithm 1. The resulting FFSM is stored for further analysis (please see below) in a textual format with transitions of the following shape:
"source@z3condition −− input@z3condition/output − > target@z3condition"

Example 11. The FFSM generated from the HFSM of Figure 9 has 4 states and 26 transitions, which is equivalent to the manually modeled FFSM of Figure 4. The source of the first transition is the root state. The first and the last FFSM transitions are:
"StartGame@true −− Exit@(and W (not S))/_1() − > StartGame@true"

[4]Publicly available from: `https://github.com/vhfragal/ConFTGen-tool`
[5]Open Source Yakindu Project `https://github.com/Yakindu/statecharts`

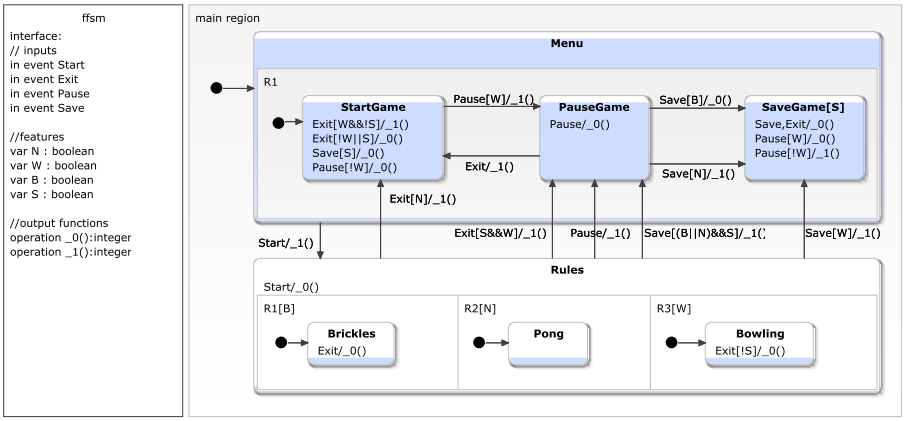

Figure 9: HFSM for AGM SPL on the implemented tool.

"Brickles*Pong*Bowling@(or B N W) −− Exit@(and S W)/_1() − > PauseGame@true"

5.1. HFSM Syntax Validation

Syntax validation is performed automatically by the implemented tool. To validate the syntax of an HFSM, we first extract the feature constraint χ of a feature model FM, and then generate assertions (corresponding to the validation properties) in the Z3 format. We execute Z3 externally by passing on the constraints in an SMT file. The generated SMT file has three parts: (i) type definitions; (ii) assertion of the feature constraint of a given feature model; (iii) assertions of the validation properties.

Example 12. To validate the syntax of the HFSM of Figure 9, we extract the feature constraint χ of the feature diagram of Figure 2. Then, we prepare the file header using type definitions and assert χ (in Z3 format):
(define-sort Feature () Bool)
(declare-const G Feature) (declare-const A Feature) (declare-const M Feature)
(declare-const L Feature) (declare-const C Feature) (declare-const R Feature)
(declare-const B Feature) (declare-const N Feature) (declare-const W Feature)
(declare-const V Feature) (declare-const Y Feature) (declare-const P Feature)
(declare-const S Feature)
(assert (and G (= A G) (= M A) (= L A) (= C G) (= R G) (= (or B N W) R)
(not (and B N)) (not (and B W)) (not (and N W)) (= V G) (= Y V) (= P V)
(=> S V)))

To validate assertions, we include several checks using assertion blocks. In Z3, *push* and *pop* commands can temporarily set the context (e.g., with assertions), and once a verification goal is discharged, the context can be reset. The $(check - sat)$ command evaluates all assertions present in the SMT file so far, and returns *sat* or *unsat*. We can complete our SMT file to check one or

more feature constraints following the structure:
(push)(assert Z3_CONSTRAINT)(check-sat)(pop)

Assume that we need to check the consistency of two different transitions x and y with the feature constraint of our FM, where the $fcomp(x) = (B \wedge N)$ and $fcomp(y) = (W \vee \neg S)$, respectively. Inside a command block which begins with *push* and ends with *pop*, we can write several *assert* commands. However, our simple check only requires one assertion for each $fcomp$. Thus, we create push-pop command blocks such as:
(push)(assert (and B N))(check-sat)(pop)
(push)(assert (or W (not S)))(check-sat)(pop)
These assertions result in *unsat* and *sat*, respectively[6]. The *unsat* result means that there is no product configuration that satisfy $(B \wedge N)$. The *sat* result means that there is at least one product configuration that satisfies $(W \vee \neg S)$, in this case $\rho_1, \rho_3, \rho_5, \rho_6$, which is a combination of subsets (due to $\vee$ operator), such that ρ_1, ρ_3, ρ_5 satisfy $\neg S$, and ρ_5, ρ_6 satisfy W (please see Example 2).

For complex validation properties such as minimality, we combine several such checks into a single block.

5.1.1. Well-formed validation

To check the constraints of a well-formed state structure, most of the items (1-7) from Definition 4.2 are covered by the Yakindu implementation based on its metamodel. The metamodel ensures that those items are always valid by construction. Hence, only validation of items 6 and 8 were added in our tool. Regarding item 6, we check whether every region state has a substate that is default. Thus, we do not allow empty regions. Regarding item 8, we check that every state is satisfied by at least one product configuration.

Example 13. Figure 10 (left) shows a state region with an invalid feature constraint which also invalidates all of its descendants (*Brickles*). The *Brickles* state inherits the feature constraints of its ancestors, in this case, $B\&\&N$ of $R1$ and true for *Rules*, *RegionA*, and AGM(root). All HFSM states are checked using their feature constraint. The resulting $fcomp(anc(R1))$ is equivalent to (and B N) in Z3 format, which is unsat (Example 12) according to our FM for AGM.

[6] Z3 online tool `https://rise4fun.com/z3`

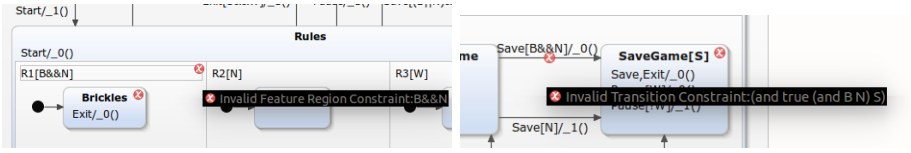

Figure 10: Invalid states (left) and transition (right) in HFSM for AGM SPL.

To check well-formed transitions, only item 1 of Definition 4.12 is covered by the metamodel and it is valid by construction. Item 2 is checked by Yakindu and we implemented item 3 in our extended tool.

Example 14. Figure 10 (right) shows an invalid transition due to its feature constraint. The transition $t = (PauseGame, Save, (B \wedge N), \_0(), SaveGame)$ has an invalid feature constraint that results *unsat* in our Z3 check.

5.2. Semantic Validation

Once we derive a semantic FFSM of our HFSM, validation properties (determinism, initially-connectedness, and minimality) can be checked as briefly introduced in Section 3.3 (we refer [12] to for full details). All these properties are checked automatically after saving the HFSM model.

To check determinism, we check all conditional states and their transitions. We select a conditional state $c = (s, \phi)$ and then select a conditional input i. First, we identify the set of satisfiable product configurations for the selected state, i.e., $\forall_{\rho \in \Lambda} \bullet \rho \vDash \phi \implies \rho \in \Lambda_s$. If c has more than one transition leaving the state with the input i, i.e., $t_1 = (c, i, \phi_1, o, c')$ and $t_2 = (c, i, \phi_2, o', c'')$, then we pairwise check whether the resulting feature constraint of those transitions ($tcomp(t_1)$ and $tcomp(t_2)$) have an intersection of product configurations, i.e, $\exists_{\rho \in \Lambda} \bullet \rho \vDash tcomp(t_1) \wedge tcomp(t_2)$. If they do, then our FFSM is not-deterministic. In other words, a deterministic FFSM cannot have a product configuration enabling two transitions leaving the same state with the same input.

Figure 11 shows an example of determinism error. The deterministic check fails when we change the feature constraint W to $(W||N)$ of the transition from *StartGame* to *PauseGame*; in that case, it will be in conflict with the self-loop transition of StartGame with Pause input and $\neg W$ feature constraint. The conflict occurs due to the non-empty intersection of two product configurations that have feature N. Thus, checking the feature model we see that both transitions are valid for products configurations with feature N.

After checking determinism, we check initially connectedness and minimality, respectively. To check minimality and initially connectedness, we use other checks to establish whether there are reaching paths for each and every state and distinguishing sequences for each and every pair of states, respectively.

Figure 12 shows an example of initially connectedness error. The initially connectedness check fails when for some product, there is no path from the initial state to some valid state. The conditional state $(SaveGame, S)$ has the feature constraint S which is satisfied by ρ_2, ρ_4 and ρ_6. To reach this conditional state in

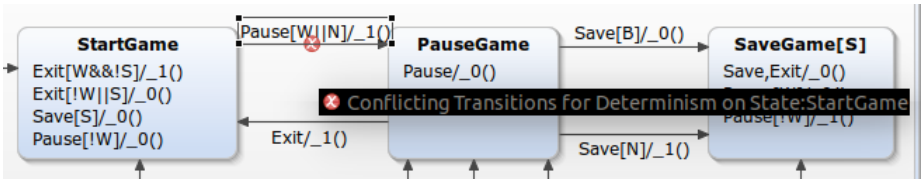

Figure 11: HFSM parts for AGM SPL with a deterministic error.

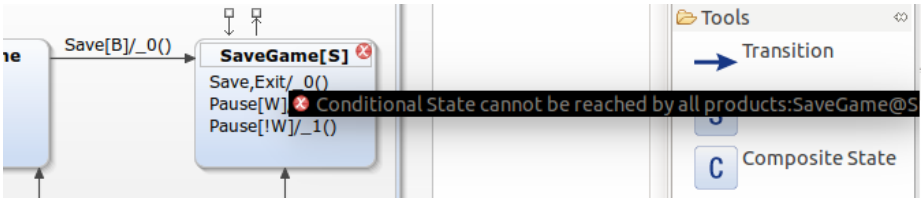

Figure 12: HFSM parts for AGM SPL with an initially connected error.

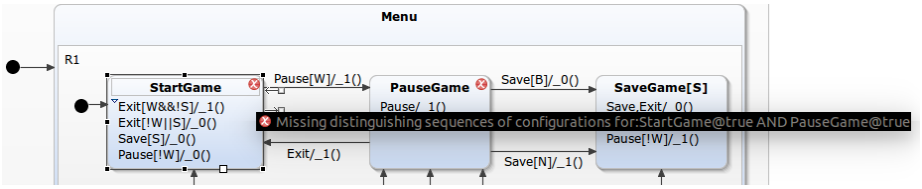

Figure 13: HFSM parts for AGM SPL with minimal error(bottom).

all products, we need three distinct paths (from B, N, and W). Once we remove the conditional transition $((PauseGame, true), Save, N, \_1(), (SaveGame, S))$, there is no path that reaches $SaveGame$ and is satisfied by ρ_4 anymore.

Figure 13 shows an example of minimality error. The minimality check fails when we cannot distinguish all pairs of conditional states. Consider the conditional states $StartGame$ and $PauseGame$; both states have the $true$ feature constraint and hence, they must be distinguishable in all valid product configurations. The $Exit$ input can distinguish the aforementioned pair of conditional states in configurations $\rho_1, \rho_2, \rho_3, \rho_4, \rho_6$. There is no input, however, that can distinguish this pair of conditional states in ρ_5.

5.3. Model Derivation

Once the HFSM is modeled, we can use a configuration file to select product configurations for AGM. Using the product configuration ρ_5 that is equivalent to $W \wedge \neg S$ (Example 2) the tool can derive a pruned HFSM. Figure 14 shows the reduced HFSM with only satisfiable elements. The semantic FFSM of the reduced HFSM is:
"StartGame@true −− Start@true/_1() − > Bowling@W"
"StartGame@true −− Exit@(and W (not S))/_1() − > StartGame@true"
"StartGame@true −− Pause@W/_1() − > PauseGame@true"
"PauseGame@true −− Start@true/_1() − > Bowling@W"
"PauseGame@true −− Pause@true/_1() − > PauseGame@true"
"PauseGame@true −− Exit@true/_1() − > StartGame@true"
"Bowling@W −− Start@true/_1() − > Bowling@W"
"Bowling@W −− Pause@true/_1() − > PauseGame@true"
"Bowling@W −− Exit@(not S)/_1() − > Bowling@W"

The semantic FFSM of the reduced HFSM can be derived into an FSM for ρ_5 by removing the feature constraints of its elements.

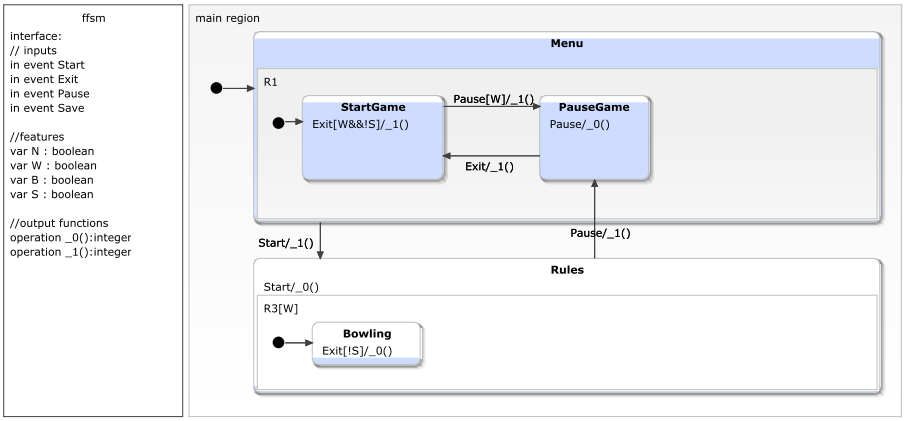

Figure 14: Derived HFSM for AGM SPL.

6. Body Comfort System Case Study

We illustrate and evaluate our approach in a prototypical implementation using a case study from the automotive domain, a simplified Body Comfort System (BCS) for the VW Golf SPL [15]. The FeatureIDE tool [28] was used to elaborate Feature Models and their configurations. The original BCS system has 19 non-mandatory features and can have 11616 configurations.

Figure 15 presents an adapted version of the feature model used to handle a part of the features with 4 non-mandatory features and 6 possible configurations for 4 components: *Finger_Protection_FP* (FP) blocking the window movement when a finger is clamped in a window, *Manual_ManPW* (ManPW) or alternatively *Automatic_AutPW* (AutPW), and *Central_Locking_System_CLS* (CLS) with optional *Automatic_Locking_AL* (AL) when the car is driving. In Example 3, we show that states with alternative features can be composed for FFSMs. The behavior of *ManPW* and *AutPW* components are similar and exclusive and hence, we can combine them in a single region by adding product-specific conditional transitions.

The behavior of components can be checked individually or in groups. In groups, they can be composed of parallel regions using hierarchical models or elaborated individually using flat models. Figure 16 presents the HFSM of four selected components of BCS. The original behavioral model of each component can be found in [15]. Inputs, outputs, and used features are presented on the left-hand side of Figure 16.

Two alternative components were modeled in the *PowerWindow* region, and product-specific transitions represent the behavior in each case. The only non-mandatory region is *CentralLockingSystem* which means that for different products, we have to compose either all three regions (first segment), and the first two on the left (second segment). Region composition is explained in Figure 7 and Figure 8.

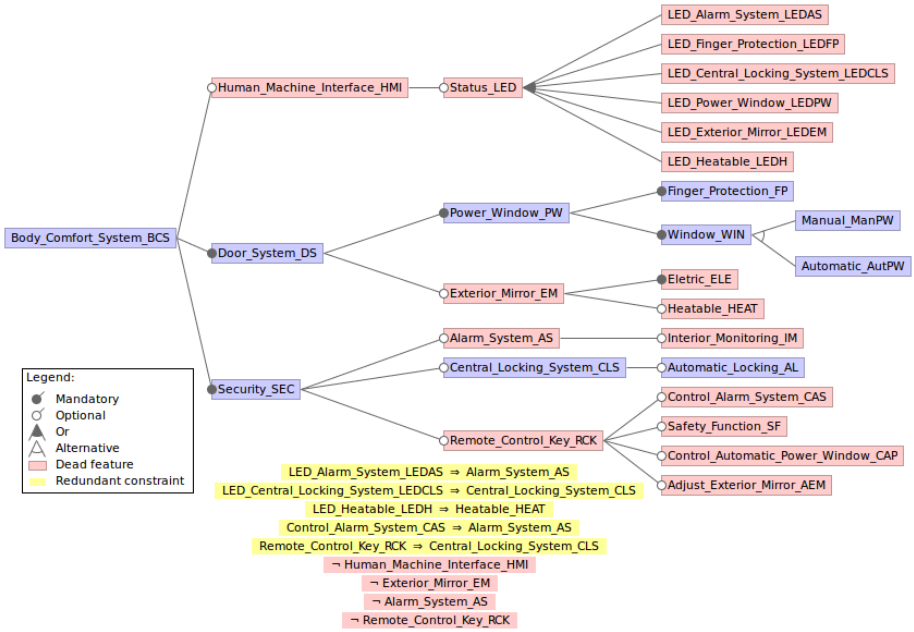

Figure 15: Adapted Feature Model of the Body Comfort System [15].

6.1. Results

The HFSM presented in Figure 16 was validated regarding both syntactic and semantic properties. To check the basic syntax and derive the semantic FFSM it took less than 2 seconds. The resulting semantic FFSM of BCS for the selected four components has 17 conditional states and 171 conditional transitions, and it took approximately 2 minutes to perform semantic checks for all three validation properties. The running environment used Ubuntu 15.04 (64 bit) operating system on an Intel processor i7-5500U at 2.40GHz with 12GB of RAM. Additional experimental results about the validation time of such properties in FFSMs were presented in [12].

Once the HFSM was validated, we chose a product configuration to derive and validate partial specifications. We pruned our HFSM for a subgroup of product configurations. For example, in a feature model configuration file we selected the Automatic Power Window component and left the Central Locking System and Automatic Locking features unchecked, resulting 3 out of 6 possible configurations. By using a feature constraint that ignores the uncheck features we derived an HFSM for those 3 product configurations. Figure 17 shows the resulting HFSM for 3 product configurations. We also derived another specification for a single product configuration by excluding other features, similar to the example presented in Section 5.3. It took 1 second to derive such models.

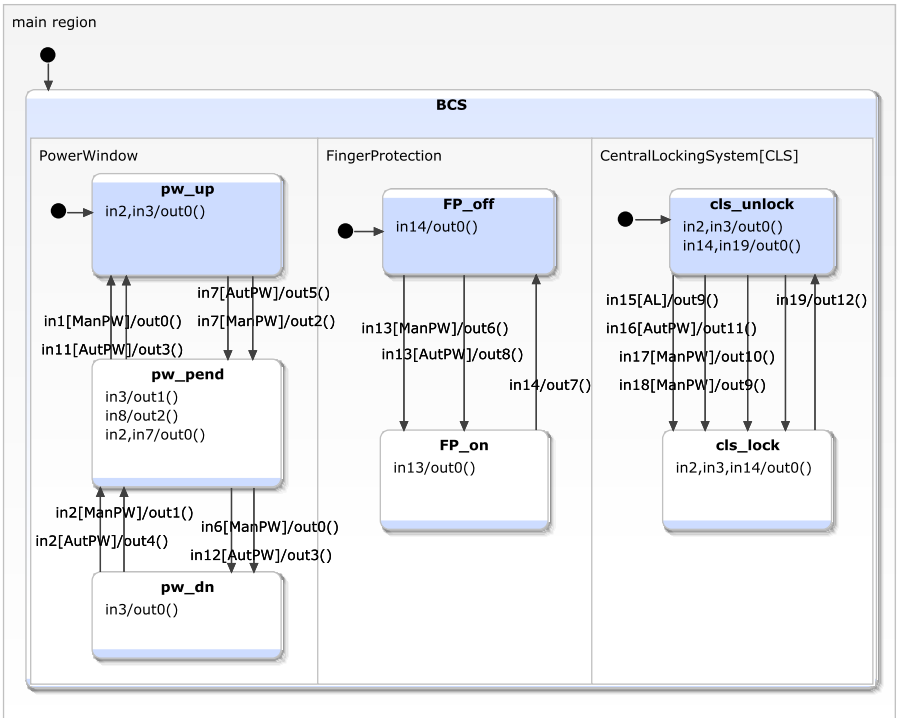

Figure 16: HFSM of 4 components of BCS.

6.2. Discussion of the Results

Scalability is the main issue of our approach as for any hierarchical model. The impact in terms of scalability concerns semantic validation that may increase as we add more parallel states. The BCS is no different, as it has several parallel regions containing 1 to 9 states. Composing all regions results in more than 50000 states which is a challenge for our semantic checks, i.e. the initially connected which may require checking several paths to each state. Also, we show in Section 4.2.2 that composing parallel regions with features may increase the number of semantic states. This is a threat to validity for the applicability for real-world cases.

Feature models and HFSMs are straightforward to model and easy to understand. The introduction of hierarchy in modeling, because it eases maintenance (by modularizing the design) and leads to a compact representation of the SPL behavior. However, large real-world specifications require a substantial amount of time to validate. A common approach to alleviate the complexity of analysis is to exploit compositionality, which seems a viable approach to consider further on.

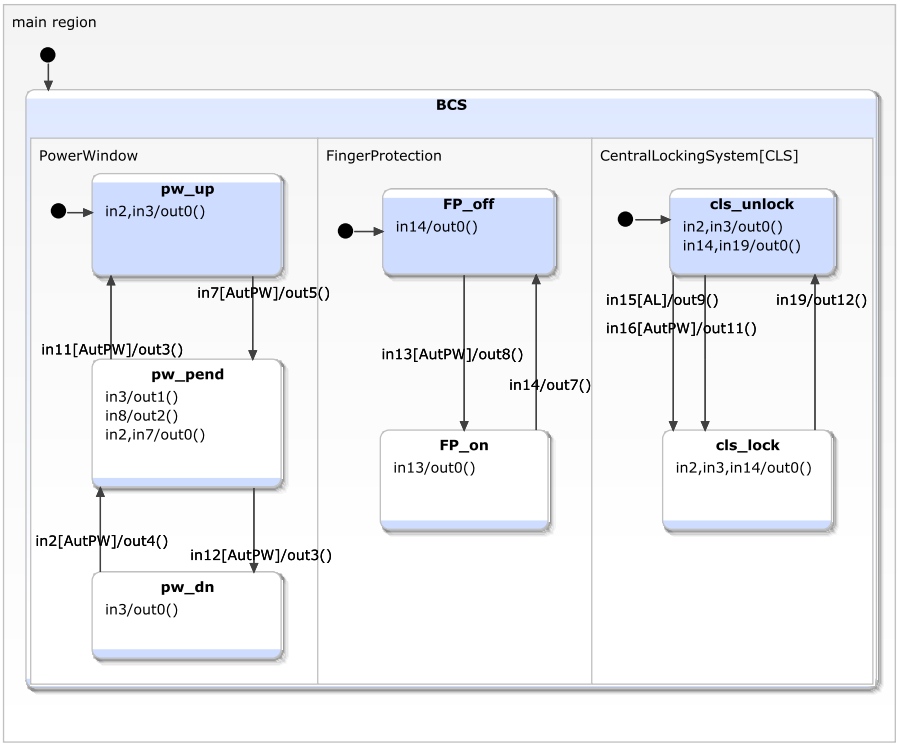

Figure 17: HFSM derived for 3 configurations with 3 components.

7. Related Work

Usually, an SPL can generate several similar products where only a few features vary from one to another. A major challenge in SPL engineering is verification of products using a simplified behavioral model that takes advantage of the similarity among products. There are proposals [29, 30] that provide a concise hierarchical formalism for representing SPL behavior in one model. However, many recent attempts are focused on formal (static and dynamic) verification, techniques model checking [30] or test generation for simple test criteria such as boundary tests [29]; we refer to [31] for an overview. In this paper, we lift the definition of an FFSM model to handle hierarchy with an HFSM. Subsequently, we provide syntactic and semantic checks to pave the way for using such hierarchical behavioral models as inputs for automatic test generation methods.

Regarding configurable models for SPLs, most modeling concepts for variability can be classified into three main approaches: annotative, compositional and transformational variability modeling [32]. Compositional approaches for modeling variability capture variation by selecting specific component variants. Compositional variability modeling [33] allows a modular description of vari-

ability but limits the impact of changes to the applied composition technique. Transformational approaches represent variability by transformation of a base architectural model. Model transformation rules guide the derivation of products by performing additions, modifications or removals using variability. For example, delta modeling [34] can represent variability in model transformation which a core system is developed, and subsequent products are derived by executing such transformations rules. Annotative approaches use variant annotations (also called 150%-models), e.g., UML stereotypes in UML models to define which model elements belong to specific product variants. In the orthogonal variability model (OVM) [35], a separate variability representation with links to the architecture model replaces direct annotations. Some approaches [36, 37] propose a pruning-based approach to UML 150% test model for SPLs, separating variability from the base models using mapping models.

Using an annotative 150% statechart, test model and transition coverage criteria, Cichos et al.'s approach [29] presents SPL test design for complete test model coverage with subsequent product subset selection for test suite execution. Weissleder et al. [38] propose an approach for automatic test suite derivation based on reusable UML state machine test models and OCL expressions. As in Featured Transition Systems [30], model fragments are annotated with presence conditions, i.e., Boolean expressions that define to which products a fragment belongs.

There a number of attempts to extend formal models to the SPL level; examples of such work include approaches based on Labeled Transition Systems [10, 39, 11] and feature-oriented approaches [22, 40]. Common to other formal feature-oriented approaches [22, 40], our proposed approach for configurable HFSMs is based on a specification that uses features of an SPL as feature constraints. However, the approach proposed in [22, 23] exploits non-deterministic models, and semantic validation of models is not considered in their approach. We are not aware of any prior study that uses formal models with hierarchy in the SPL context to validate properties such as determinism, initially connectedness, and minimality.

8. Conclusions

In this paper, we presented the Hierarchical Featured State Machine (HFSM) formalism for representing behavioral test models in the Software Product Line (SPL) context. The HFSM improves the modeling of SPL behavior compared to their underlying flat model, i.e., Featured Finite State Machines (FFSMs), by grouping states and transitions into hierarchies.

Inspired by statecharts and UML state diagrams, we defined the syntax and semantics of HFSMs. In the syntactic part, we defined well-formed state structures and transitions. In the semantical part, we used FFSMs as the semantic model of HFSMs. State configurations were transformed into conditional states. We also addressed the composition of orthogonal states, which can potentially lead to a combinatorial explosion of states. We showed how using feature constraints in regions can tame this combinatorial explosion in some cases.

We showed how basic validation properties (as prerequisites for most testing techniques) can be checked on the semantical FFSM models. To mechanize the validation of HFSMs, we implemented a tool by adapting the Yakindu project. We added several checks regarding syntactic and semantical validation properties and used the Z3 SMT solver to verify the generated feature constraints. The tool performs all the checks automatically. Moreover, the tool provides model derivation commands that are useful to create partial HFSM models for a single or a group of product configurations.

Finally, we used the Body Comfort System as a case study. We noticed that we could not analyze the whole specification due to the well-known state explosion problem: the resulting flat FFSM would have more than 50000 states. Thus, we selected some parts of the original specification. The results indicate that our HFSM is able to represent the parallel SPL behavior of four components each having a few states, the semantical FFSM models have up to 17 states and 171 transitions.

As future work, we plan to use HFSMs to extend our recent FFSM-based test-case generation method [25]. We also plan to include history, deep-history, join/forks, and entry/exit points connections in the HFSM model. Moreover, we plan to explore the state explosion problem identified in the HFSM used in the case study and use well-known reduction techniques and adapt them to the context of model-based testing.

References

[1] J. Greenfield, K. Short, Software factories: Assembling applications with patterns, models, frameworks and tools, in: Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications, OOPSLA '03, ACM, New York, NY, USA, 2003, pp. 16–27. doi:10.1145/949344.949348.

[2] K. Pohl, G. Böckle, F. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer-Verlag New York, Inc., 2005.

[3] S. Oster, A. Wubbeke, G. Engels, A. Schürr, A Survey of Model-Based Software Product Lines Testing, in: Model-Based Testing for Embedded Systems, CRC Press, 2012, pp. 338–381. doi:10.1145/2362536.2362545.

[4] E. Engström, P. Runeson, Test overlay in an emerging software product line - An industrial case study, Information and Software Technology 55 (3) (2013) 581–594. doi:10.1016/j.infsof.2012.04.009.

[5] G. J. Myers, C. Sandler, T. Badgett, T. M. Thomas, The Art of Software Testing, 2nd Edition, Vol. 15, John Wiley & Son, 2004. doi:10.1002/stvr.322.

[6] A. Tevanlinna, J. Taina, R. Kauppinen, Product family testing, ACM SIGSOFT Software Engineering Notes 29 (2) (2004) 12. doi:10.1145/979743.979766.

[7] D. Lee, M. Yannakakis, Principles and Methods of Testing Finite State Machines - A Survey, Proceedings of the IEEE 84 (8) (1996) 1090–1123. doi:10.1109/JPROC.1996.533955.

[8] M. Broy, B. Jonsson, J. Katoen, M. Leucker, A. Pretschner, Model-Based Testing of Reactive Systems: Advanced Lectures, Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[9] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, Others, Using formal specifications to support testing, ACM Computing Surveys (CSUR) 41 (2) (2009) 9. doi:10.1145/1459352.1459354.

[10] M. Lochau, J. Kamischke, Parameterized Preorder Relations for Model-Based Testing of Software Product Lines, in: Proceedings of the 5th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation. Technologies for Mastering Change (ISoLA), 2012, pp. 223–237. doi:10.1007/978-3-642-34026-0_17.

[11] M. Varshosaz, H. Beohar, M. R. Mousavi, Delta-Oriented FSM-Based Testing, in: Proceedings of the International Conference on Formal Engineering Methods (ICFEM), Springer, 2015, pp. 366–381. doi:10.1007/978-3-319-25423-4_24.

[12] V. H. Fragal, A. Simao, M. R. Mousavi, Validated Test Models for Software Product Lines: Featured Finite State Machines, in: Proceedings of the 13th International Conference on Formal Aspects of Component Software (FACS), Springer, 2016, pp. 210–227. doi:10.1007/978-3-319-57666-4_13.

[13] D. Harel, A. Naamad, The STATEMATE semantics of statecharts, ACM Transactions on Software Engineering and Methodology (TOSEM) 5 (4) (1996) 293–333. doi:10.1145/235321.235322.

[14] L. de Moura, N. Bjørner, Z3: An Efficient SMT Solver, in: Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, 2008, pp. 337–340. doi:10.1007/978-3-540-78800-3_24.

[15] S. Lity, R. Lachmann, M. Lochau, I. Schaefer, Delta-oriented Software Product Line Test Models - The Body Comfort System Case Study, Tech. rep., TU Braunschweig (2013).

[16] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, A. S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study, Tech. rep., Carnegie-Mellon University Software Engineering Institute (Nov. 1990).

[17] P. Y. Schobbens, P. Heymans, J. C. Trigaux, Feature Diagrams: A Survey and a Formal Semantics, in: Proceedings of the 14th IEEE International Conference on Requirements Engineering (RE), IEEE, 2006, pp. 139–148. doi:10.1109/RE.2006.23.

[18] S. Kang, J. Lee, M. Kim, W. Lee, Towards a Formal Framework for Product Line Test Development, in: Proceedings of the 7th IEEE International Conference on Computer and Information Technology (CIT), IEEE, 2007, pp. 921–926. doi:10.1109/CIT.2007.40.

[19] F. Linden, K. Schmif, E. Rommes, Software Product Lines in Action, Springer, 2007.

[20] SEI, A framework for software product line practice (2011).
URL http://www.sei.cmu.edu/productlines/tools/framework/

[21] D. Batory, Feature Models, Grammars, and Propositional Formulas, in: Proceedings of the 9th international conference on Software Product Lines (SPLC), 2005, pp. 7–20. doi:10.1007/11554844_3.

[22] H. Beohar, M. R. Mousavi, Input-output Conformance Testing Based on Featured Transition Systems, in: Proceedings of the 29th Annual ACM Symposium on Applied Computing, 2014, pp. 1272–1278. doi:10.1145/2554850.2554949.

[23] H. Beohar, M. R. Mousavi, Spinal Test Suites for Software Product Lines, in: Model-Based Testing (MBT), Vol. 141, 2014, pp. 44–55. arXiv:1403.7260, doi:10.4204/EPTCS.141.4.

[24] OMG, OMG Unified Modeling Language. Version 2.5, Tech. rep. (2015).

[25] V. H. Fragal, A. Simao, M. R. Mousavi, U. C. Turker, Extending hsi test generation method for software product lines, The Computer Journal-doi:10.1093/comjnl/bxy046.

[26] G. Luo, A. Petrenko, R. Petrenko, G. V. Bochmann, Selecting Test Sequences For Partially-Specified Nondeterministic Finite State Machines, in: Proceedings of The International Federation for Information Processing (IFIP), 1994, pp. 91–106. doi:10.1007/978-0-387-34883-4_6.

[27] E. Mikk, Y. Lakhnech, C. Petersohn, M. Siegel, On formal semantics of statecharts as supported by statemate, in: Proceedings of the 2Nd BCS-FACS Conference on Northern Formal Methods, Springer-Verlag, 1997, pp. 1–12.

[28] T. Thüm, C. Kästner, F. Benduhn, J. Meinicke, G. Saake, T. Leich, FeatureIDE: An Extensible Framework for Feature-Oriented Software Development, Science of Computer Programming 79 (2014) 70–85.

[29] H. Cichos, S. Oster, M. Lochau, A. Schürr, Model-Based Coverage-Driven Test Suite Generation for Software Product Lines, in: Proceedings of the 14th international conference on Model driven engineering languages and systems (MODELS), 2011, pp. 425–439. doi:10.1007/978-3-642-24485-8_31.

[30] A. Classen, M. Cordy, P.-Y. Schobbens, P. Heymans, A. Legay, J.-F. Raskin, Featured Transition Systems: Foundations for Verifying Variability-Intensive Systems and Their Application to LTL Model Checking, IEEE Transactions on Software Engineering 39 (8) (2013) 1069–1089. doi:10.1109/TSE.2012.86.

[31] T. Thüm, S. Apel, C. Kästner, I. Schaefer, G. Saake, A Classification and Survey of Analysis Strategies for Software Product Lines, ACM Computing Surveys (CSUR) 47 (6) (2014) 1–45. doi:10.1145/2580950.

[32] I. Schaefer, R. Rabiser, D. Clarke, L. Bettini, D. Benavides, G. Botterweck, A. Pathak, S. Trujillo, K. Villela, Software diversity: state of the art and perspectives, International Journal on Software Tools for Technology Transfer 14 (5) (2012) 477–495. doi:10.1007/s10009-012-0253-y.

[33] A. Haber, H. Rendel, B. Rumpe, I. Schaefer, F. van der Linden, Hierarchical Variability Modeling for Software Architectures, in: Proceedings of the 15th International Software Product Line Conference (SPLC), IEEE, Munich, Germany, 22-26 August, 2011, pp. 150–159. doi:10.1109/SPLC.2011.28.

[34] D. Clarke, M. Helvensteijn, I. Schaefer, Abstract delta modeling, in: ACM SIGPLAN Notices, Vol. 46, 2011, p. 13. doi:10.1145/1942788.1868298.

[35] K. Pohl, A. Metzger, Software product line testing, Communications of the ACM 49 (12) (2006) 78–81. doi:10.1145/1183236.1183271.

[36] K. Czarnecki, M. Antkiewicz, Mapping Features to Models: A Template Approach Based on Superimposed Variants, in: Proceedings of the 4th international conference on Generative Programming and Component Engineering (GPCE), 2005, pp. 422–437. doi:10.1007/11561347_28.

[37] H. Grönninger, H. Krahn, C. Pinkernell, B. Rumpe, Modeling Variants of Automotive Systems using Views, in: Tagungsband Modellierungs-Workshop MBEFF: Modellbasierte Entwicklung von eingebetteten Fahrzeugfunktionen, TU Braunschweig, Berlin, Germany, 2008, pp. 1–14. arXiv:1409.6629.

[38] S. Weißleder, D. Sokenou, B.-H. Schlingloff, Reusing state machines for automatic test generation in product lines, in: Model-Based Testing in Practice (MoTiP), 2008, p. 10.

[39] M. Lochau, S. Lity, R. Lachmann, I. Schaefer, U. Goltz, Delta-oriented model-based integration testing of large-scale systems, Journal of Systems and Software 91 (2014) 63–84. doi:10.1016/j.jss.2013.11.1096.

[40] X. Devroey, G. Perrouin, M. Papadakis, A. Legay, P.-Y. Schobbens, P. Heymans, Featured model-based mutation analysis, in: Proceedings of the 38th International Conference on Software Engineering (ICSE), ACM, 2016, pp. 655–666. doi:10.1145/2884781.