Incremental and Commutative Composition of State-machine Models of Features

Sandy Beidu, Joanne M. Atlee, Pourya Shaker
David R. Cheriton School of Computer Science
University of Waterloo, Canada
{sbeidu,jmatlee,p2shaker}@uwaterloo.ca

***Abstract*—In this paper, we present a technique for incremental and commutative composition of state-machine models of features, using the FeatureHouse framework. The inputs to FeatureHouse are feature state-machines (or state-machine fragments) modelled in a feature-oriented requirement modelling language called FORML and the outputs are two state-machine models: (1) a model of the whole product line with optional features guarded by presence conditions; this model is suitable for family-based analysis of the product line; and (2) an intermediate model of composition that facilitates incremental composition of future features. We discuss the challenges and benefits of our approach and our implementation in the FeatureHouse.**

I. Introduction

We are interested in supporting feature modularity in requirements modelling of feature-rich systems - especially software product lines (SPLs), where families of similar products are understood, constructed, managed, and evolved in terms of their features. In previous work, we presented a feature-oriented requirements modelling language called FORML [1], in which behavioural requirements are expressed in a UML-like state-machine language. Features are specified as distinct modules, and the requirements of the SPL are derived by composing all of the feature modules into a single SPL model. An SPL model represents behaviours of all possible products derivable from a product line. This model enables family-based analysis where the whole product line is analyzed together instead of analyzing individual products. It has been shown that family-based model checking of an SPL is more efficient than model checking all products individually because the analysis can exploit the commonalities among different products [2].

Distinguishing aspects of FORML include (1) support for modelling new features as state-machine *fragments* that extend existing feature modules, (2) language constructs for explicitly modelling intended interactions among features (so that analyses that detect interactions can exclude the intended interactions), and (3) a composition operator that is commutative and associative. A commutative composition operator has considerable advantages. For one, engineers do not need to identify an order of composition to derive a product or an SPL model from a collection of features. Second, the order of composition does not affect analysis results.

In [1], we presented the requirements for a commutative and associative composition of state machine models of features into an SPL model. In this paper, we present the semantics of the feature composition and its implementation using FeatureHouse [3]. We explain how FeatureHouse composition, based on superimposition, applies to state-machine models – especially how technical challenges posed by fine-grained state-machine fragments and commutative composition are addressed. We conclude with a discussion of our experiences and our plans for future work.

II. Preliminaries

This section provides an overview of feature-oriented requirement modelling and a supporting language, FORML. Throughout this paper, we use example models from an automotive system comprising a Basic Driving Service (BDS) and a small set of optional features.

A. Feature-Oriented Requirements Modelling

Feature-oriented requirements modelling is an approach to manage the complexity in specifying and analyzing a software system by decomposing the system into features, where a *feature* is a coherent and identifiable bundle of system functionality [4]. In product-line development, a family of related software products (e.g. automobile models) share a common set of mandatory features, and products are differentiated by their variable (optional or alternative) features. The behavioural requirement for each feature is modeled independently as a *feature module*. An individual *product variant* of the product line is derived by composing a subset of features. A composed model of all the features in the product line is called a *150% model* (a.k.a *virtual product* or *metaproduct* or *product simulator*). In most cases this model is not a valid product because usually there are mutually-exclusive features. A 150% model can be converted into an *SPL model* where feature-specific behaviour is guarded by feature variables that represent the presence or absence of a feature in a particular product variant.

B. Overview of FORML

This sub-section provides an overview of a feature-oriented requirement modelling language called FORML. A FORML *behaviour model* comprises a set of feature modules, each of which is a set of state machines or state-machine fragments, called *feature machines* or *feature-machine fragments*, respectively. The behaviour of an SPL product is the parallel

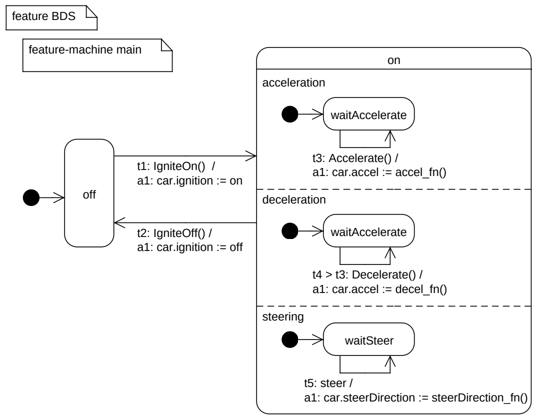
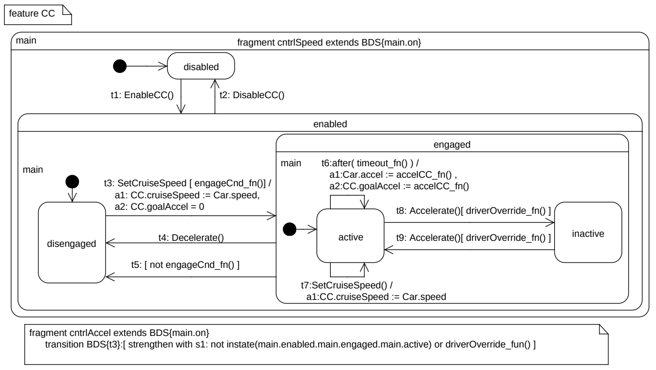
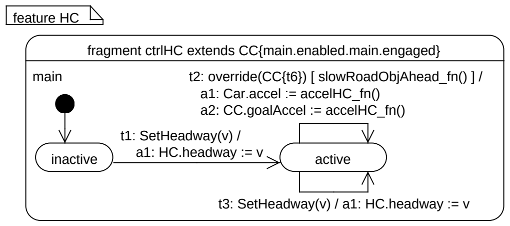


Fig. 1. Basic Driving Service (BDS) standalone feature machine


feature CC
main
fragment cntrlSpeed extends BDS{main.on}
disabled
t1: EnableCC()
t2: DisableCC()
enabled
main
disengaged
t3: SetCruiseSpeed [engageCnd_fn()] /
a1: CC.cruiseSpeed := Car.speed,
a2: CC.goalAccel = 0
t4: Decelerate()
t5: [not engageCnd_fn()]
engaged
main
t6:after(timeout_fn()) /
a1:Car.accel := accelCC_fn() ,
a2:CC.goalAccel := accelCC_fn()
active
t8: Accelerate()[driverOverride_fn()]
t9: Accelerate()[driverOverride_fn()]
inactive
t7:SetCruiseSpeed() /
a1:CC.cruiseSpeed := Car.speed
fragment cntrlAccel extends BDS{main.on}
transition BDS{t3}:[strengthen with s1: not instate(main.enabled.main.engaged.main.active) or driverOverride_fun()]
Fig. 2. Cruise Control (CC) feature-machine fragments


feature HC
fragment ctrlHC extends CC{main.enabled.main.engaged}
main
t2: override(CC{t6}) [slowRoadObjAhead_fn()] /
a1: Car.accel := accelHC_fn()
a2: CC.goalAccel := accelHC_fn()
t1: SetHeadway(v) /
a1: HC.headway := v
inactive
active
t3: SetHeadway(v) / a1: HC.headway := v
Fig. 3. Headway Control (HC) feature-machine fragment

execution of its feature machines extended by its fragments. Details of FORML can be found in Shaker's PhD thesis [5]. In addition, a FORML model includes a *feature model* that defines valid configurations of features – that is the product variants of the SPL.

1) Feature Machines: The notation for a FORML feature machine is based on UML state machines [6]. A feature machine consists of a set of states and transitions between the states. A state may be a basic state or a superstate containing one or more orthogonal *regions*, where each region models a concurrent sub state machine. For example, Figure 1 shows a feature machine for a Basic Driving Service (BDS) feature that models the behaviour of an automobile. The machine consists of a basic state off and a superstate on, which represents the vehicle's ignition being off or on. The superstate on has three orthogonal regions that represent the vehicles concurrent behaviour wiht respect to $acceleration$, $braking$ and $steering$, when the ignition is on. Transitions between states are triggered by events and conditions over variables in the SPL's environment, and transition actions are assignments to environmental variables. There are no internal variables.

FORML provides a rich transition label with several extension points that allow for fine-grained extensions of feature machines. A transition between states has a label of the form:

$$id : te\ [gc]\ /\ id_1 : [c_1]a1, \ldots, id_n : [c_n]a_n$$

where id is the name of the transition; te is an optional triggering event; gc is a boolean guard condition; and $a_1...a_n$ are concurrent actions, each with its own name id_i and guard condition c_i. An execution step consists of the execution of all concurrently enabled transitions and their actions.

2) Feature-Machine Fragments: An SPL evolves by incrementally adding new features. This is modelled by adding a new feature machine that executes in parallel with existing machines or by adding a fragment that extends existing feature machine or fragment[1]. Specifically, a new feature can *add* new behaviours, or can *remove* or *replace* behaviours at specified locations of an existing feature.

[1]When a feature B extends a feature A, then the presence of feature *B* in a product requires the presence of feature *A*. This dependency must be specified in the *feature model*.

Adding behaviours: A feature-machine fragment can add new elements to an existing feature machine at specific extension points:

- A *new region* that extends an existing state
- A *new transition* (and possibly new source and destinations states) that extends an existing feature machine.
- A *new action* that extends an existing transition
- A *weakening clause* that extends the guard condition of an existing transition or action with a disjunct, thereby weakening the guard condition.

Figure 2 shows the machine fragment for a new feature, Cruise Control (CC) that extends the feature machine of BDS. CC is modelled as a new orthogonal region that extends the state on of BDS; the location of each extension is specified at the top of the fragment.

Removing behaviours: A new feature can restrict the behaviour of an existing feature by extending the guard condition of one of its transitions or actions with a conjunct, called a *strengthening clause*. Strengthening a guard condition results in the guard condition being satisfied less frequently, and therefore leads to removed behaviours. For example, feature CC in Figure 2 adds a strengthening clause (specified as UML note) to the guard condition of transition $t3$ of feature BDS. The clause prohibits BDS from processing *Accelerate* input events when CC is active, so that CC can handle those events.

Replacing behaviours: A new feature can replace existing behaviour by specifying new transitions that are enabled under similar conditions as an existing transition but that have different actions. This can be be specified using the following language constructs:

- $t2 > F\{t1\}$: ...
  specifies that a new transition $t2$ has *priority* over an existing transition $t1$ in a feature machine F, whenever

both $t2$ and $t1$ are simultaneously enabled.

- $t2 : override(F\{t1\})[c]\ /:...$
  specifies a new transition $t2$ that *overrides* an existing transition $t1$ in feature machine F. An override differs from a transition priority in that the enabling condition of $t2$ implicitly has the same enabling conditions as $t1$, but could be strengthened with an additional guard c.

For example, the Headway Control (HC) feature in Figure 3 adds a new transition $t2$ that overrides transition $t6$ of CC whenever a car gets too close to the car ahead of it.

III. Composing Features in An SPL

An SPL model is derived by composing the feature machines and fragments into a single all-inclusive model, with feature specific behaviour guarded by feature variables.

A. FeatureHouse Integration

We implemented our feature-machine composition using FeatureHouse [3]. FeatureHouse provides a generic framework for structural composition of software artefacts. The framework includes the following tools:

- *FSTGenerator* that automates the generation of a lexical analyser, a parser, and pretty printer based on a provided BNF grammar called *FeatureBNF*. The parser reads an input model and produces an abstract syntax tree, called a feature structure tree (FST) [7] for every feature it parses.
- *FSTComposer* that composes FSTs using superimposition.

Our inputs to FeatureHouse are feature machines (or feature-machine fragments) modelled in FORML, and the outputs are two state-machine models: (1) an intermediate model of composition that facilitates incremental composition of future features; and (2) a model of the whole product line with optional features guarded by presence conditions. The latter model is suitable for family-based analysis of the product line. There were some technical challenges that we had to address in implementing our state-machine composition in FeatureHouse. These included: (1) expanding fragments and qualifying model elements to ensure uniqueness of model element names, which is necessary for superimposition; (2) ensuring that the composition result is insensitive to the composition order and (3) guarding each feature fragment (which could be as small as a clause within a complex guard condition) with an appropriate presence condition and ensuring that transition priorities and overrides are preserved in the composition.

We discuss and address each of these technical challenges in the rest of this section.

B. Expanding Fragments and Qualifying Model Elements

A feature-machine fragment specifies the extension point within another feature where it should be attached. This eases the specification of fragments and avoids duplicating parts of other feature machines which have already been specified. It was challenging to create the BNF grammar of fragments because we had to ensure that superimposition matches the context of a fragment with the correct extension point of existing feature. Prior to composition, each fragment is expanded to ensure this correct matching. The fragment expanded machine includes only model elements from the extended feature-machine that is necessary to reach the extension point. For example feature CC in Figure 2 is a fragment whose context is state $BDS\{main.on\}$ from BDS in Figure 1. This fragment must be expanded into a feature machine which has region $main$ and state on from feature machine BDS. Other model elements like state off, regions $acceleration$, $deceleration$ and $steering$ from BDS are not necessary in the CC's expanded machine, and are not included.

Also in order for superimposition to work in FeatureHouse, all model element names must be unique, to ensure that model elements can be easily referenced and extended. The problem is that different features can introduce elements with the same name. For example, both features CC and HC introduce a distinct region $main$ in CC's $engaged$ state. To eliminate such name clashes, we pre-process the feature machines and fragments, and qualify the name of each model element n with the name of the feature F that *introduces it*. For instance, weakening and strengthening clauses are named and qualified with the name of the feature that introduced them so that they can themselves be strengthened or weakened. In Figure 2, the strengthening clause introduced by feature CC is named $s1$ and can be referenced by another feature as $CC\{s1\}$.

C. Superimposition of state-machines

An SPL model is derived by composing feature machines and fragments using superimposition. *Superimposition* is the process of composing software artifacts of different components by merging their common substructures.

behaviour-model ::= *state-machine*+ **macro***
state-machine ::= *state** **initial-state**? *transition**
state ::= *region**
region ::= *state*+ **initial-state**?
transition ::= **src**? **dest**? (**trigger** | **override-spec**)?
        targets* condition? action*
action ::= **override-spec**? **targets*** *condition*? **WCA**?
condition ::= **predicate**? *clause**
clause ::= *predicate*? **clause-type**? *clause**

Fig. 4. Grammar definition of an FST: *nonterminal* and **terminal** symbols are denoted with different fonts. + denotes one or more, * denotes zero or more, and ? denotes zero or one repetitions of the preceding syntactic element.

Feature machines and fragments are represented by an abstract-syntax-tree called feature structure trees (FSTs) [7]. Figure 4 shows a grammar definition of an FST. FST nodes specify the name and type of a model's elements, including state-machines, regions, states, transitions, conditions, etc. An FST node may be a *non-terminal* such as a transition, in which case it represents a composite element whose structure is exposed as subtrees of the node. The non-terminal nodes correspond to the extension points of feature modules. Whereas, a *terminal* node, such as transition's triggering event, represents a model element whose structure is atomic and thus cannot be extended by a feature fragment.

Fig. 5. Superimposition of FSTs of features BDS (unshaded area with continuous outline), CC (shaded area without outline) and HC (pattern-filled area with a dashed outline). FST non-terminal nodes are rectangles with a dashed outline whereas terminal nodes are rectangles with continuous outlines. The $\cdots$ represents omitted information.

A fragment is located in the FST of the feature that *introduces* the fragment. The context of a fragment (i.e. the extension point in another feature where the fragment is attached) is mirrored in the fragment FST as a path from the FST root to the root of the fragment. For example, in Figure 5, the context of CC's strengthening clause $CC\{s1\}$ is the path $BDS\{main\}.t3.t3$: $BDS\{main\}$ machine, transition $t3$ in that machine, guard condition $t3$ of that transition. Composition of the feature machines and fragments is the superimposition of the features' FSTs. Superimposition of FSTs is effectively an *overlay* of the FSTs, where elements with the same name and type are merged. Figure 5 shows the superimposition of the FSTs of the feature machines of BDS and the machine fragments of CC and HC. The superimposition of a set of (qualified and expanded) feature machines $F'_1 \cdots F'_n$ is denoted by the expression

$$F'_1 \bullet \cdots \bullet F'_n$$

where $\bullet$ is the *superimposition operator* which takes two feature modules as operands and returns a composed feature module. The superimposition operator is commutative, which means that the order in which feature machines are superimposed does not affect the results. The superimposition is also associative, which means that all parenthesizations of the above expression have the same result. The proofs [2] are based on the commutativity and associativity of the operations used to combine model elements that are not merged:

- the *union* of parallel machines in an integrated model
- the *union* of concurrent regions in a machine
- the *union* of states and transitions in a region
- the *union* of concurrent actions in a single transition
- the conjunction of strengthening clauses in a guard; where one conjunct is the disjunction of weakening clauses and the original guard condition of the transition.

If multiple weakening ($w_1 \ \cdots \ w_n$) and strengthening clauses ($s_1 \ \cdots \ s_n$) extend the same guard condition c, we define a canonical composition of the guard and clauses as:

[2]A complete proof is provided in [5].

Fig. 6. SPL model with features BDS, CC and HC. Light-gray and gray areas represent feature fragments of CC and HC introduced into feature machine of BDS, respectively.

$$(c \vee w_1 \vee \cdots \vee w_n) \wedge s_1 \wedge \cdots \wedge s_m$$

This canonical composition ensures that the set of possible valuations is insensitive to the order in which the features are composed.

D. Generating the Product Line Model

The default output of FeatureHouse after superimposing a collection of FSTs is a 150% model - a composed model for a *product* that has all the features in the SPL. We convert the 150% into an SPL model by guarding each feature-specific behaviour with a *presence condition* [8] that makes the presence of that behaviour conditional on whether its feature is present in the product. Specifically, a presence condition is a boolean variable that is named after a feature. Presence conditions are added to the guard conditions of the transitions and actions that are introduced by an optional feature. Let F be an optional feature in an SPL, for which we introduce a corresponding presence condition F.

- For a transition or action whose guard condition is gc, the presence condition is added as a conjunct resulting in a new guard: $F \wedge (gc)$.

How to augment a clause with its presence conditions is less obvious. Let $\mathcal{W}$ be an optional feature that introduces a weakening clause and $\mathcal{S}$ be an optional feature that introduces a strengthening clause to the above guard gc.

- A weakening clause w (a disjunct) affects its guard only if its feature $\mathcal{W}$ is present and the clause holds: $F \wedge (gc \vee (\mathcal{W} \wedge w))$.
- A strengthening clause s (a conjunct) contributes only if its feature $\mathcal{S}$ is present: $F \rightarrow (gc \wedge (\mathcal{S} \rightarrow s))$.

The final step is to construct the canonical composition of each guard condition and its (possibly multiple) clauses. Weakening and strengthening clauses have lower precedence than the presence condition of the element they extend[3]. If a presence condition $\mathcal{P}$ is applied to a condition p (note that p can be a

[3]Recall the footnote on feature dependencies in Section II-B2: if a feature P is not present, then the features that introduce extensions to P cannot be present.

guard condition, or a weakening or strengthening clause) that is extended by a set of weakening clauses $w_1 ... w_n$ (introduced by $\mathcal{W}_1 ... \mathcal{W}_n$, respectively) and a set of strengthening clauses $s_1 ... s_m$ (introduced by features $\mathcal{S}_1 ... \mathcal{S}_m$, respectively), the resulting condition is as follows:

- If p is a guard condition or a weakening clause:

$$\mathcal{P} \wedge \left[\begin{array}{c} (p \vee (\mathcal{W}_1 \wedge w_1) \vee \cdots \vee (\mathcal{W}_n \wedge w_n)) \wedge \\ (\mathcal{S}_1 \rightarrow s_1) \wedge \cdots \ \wedge (\mathcal{S}_m \rightarrow s_m) \end{array} \right]$$

- If p is a strengthening clause:

$$\mathcal{P} \rightarrow \left[\begin{array}{c} (p \vee (\mathcal{W}_1 \wedge w_1) \vee \cdots \vee (\mathcal{W}_n \wedge w_n)) \wedge \\ (\mathcal{S}_1 \rightarrow s_1) \wedge \cdots \ \wedge (\mathcal{S}_m \rightarrow s_m) \end{array} \right]$$

For example, in Figure 6 (an SPL model composed of BDS, CC and HC), if BDS were an optional feature, the canonical composition of $t3$'s guard ($True$) and the strengthening clause introduced by feature CC ($not\ instate \cdots$) would have resulted in the condition

$$BDS \rightarrow (True \wedge (CC \rightarrow (not\ inState...)))$$

Since BDS is not an optional feature in this example, there is no addition of presence condition BDS, resulting in the expression shown in the figure.

E. Implementation in FeatureHouse

When integrating a new language into FeatureHouse one has to decide if the merging of terminal nodes is allowed or not. If the merging of terminal nodes is allowed, *composition rules* for each terminal node type should be specified. For example, in a Java-like language, two method fields (which are terminal nodes) can be composed by replacing value of one method field with the value of the other. The issue with such composition rules is that it becomes difficult to achieve commutativity, which enables incremental composition. We therefore disallow the merging of terminal nodes in our composition. During the superimposition step only nonterminal nodes are merged to produce a *composed FST*. The 150% and SPL models are generated from the composed FST.

Both outputs (150% and SPL models) were implemented using variants of the *Printer* class which is responsible for generating the output of FeatureHouse from the composed FST. This modified versions of the Printer class rewrite the conditional expressions of transitions using the rules described in Section III-D. Running our version of the FeatureHouse tool therefore generates two outputs: (1) The 150% model with the canonical composition of clauses but without presence conditions and (2) an SPL model with both presence conditions and canonical composition of clauses. The first model allows for incremental composition of future feature machines and fragments while the second allows for model checking the whole product line.

Figure 7 depicts FeatureHouse architecture with FORML integration. The *Pre-processor* shown in Figure 7 expands fragments and qualifies model elements to produce the *Feature Modules*. The addition of presence conditions and canonical composition of clauses are implemented in the *Printer* processes. The arrow from the *150% model* back to the input of

Fig. 7. FeatureHouse architecture showing FORML integration (shaded boxes).

FSTComposer indicates that new features can be composed into an existing composed (150%) model.

We have exercised our tool by using it to construct an SPL model from feature modules modelling automotive features. The resulting SPL model was used in a case study to evaluate some SAT-based model checkers for software product-line requirements [9].

IV. Discussion

One of the main goals of this work is to support feature-oriented software development of a software product-line. With feature modularity, features are modelled separately. However, the modeller will eventually want to visualize and (manually or automatically) analyze feature combinations corresponding to products of the SPL. One simple way to achieve this is through superimposition, which is essentially the union of all the features such that each new feature and its behaviours are added as variations to the original system. The issue is how to deal with features' elements that cannot be unioned (because they represent explicit overrides). For example, transition conditions cannot simply be unioned since they may be strengthening or weakening (i.e. overriding) the same existing condition. One way to address this is by specifying composition rules for dealing with such elements.

We wanted our composition to be commutative and associative to enable incremental development. Adding conjuncts and disjuncts arbitrarily to an original guard condition is not commutative or associative. To achieve these properties in our feature composition, we define a canonical composition of clauses in which all conjunctions have priority over disjunctions, or vice versa; either priority order results in a composition that is both commutative and associative. Previously [1], we used an ordering that matches the operator precedence used in logic and programming languages. Based on our experiments of using these modelling constructs, we now promote giving strengthening clauses priority over weakening clauses – because strengthening clauses typically model resolutions to known feature interactions, whereas weakening clauses typically liberalize the enabling conditions of behaviours in other features. We view resolutions to undesired interactions as being more critical than liberalizing the enabling conditions of behaviour. But this hypothesis needs to be tested.

When it came to implementing a tool, we wanted to create a robust tool which could be implemented within a short amount of time. But often, the process of implementing a new tool is time consuming and error prone. For instance, to implement a tool from scratch that automates the composition of FORML state-machines, we would need to develop a parser, a composer and a pretty printer. Instead of manually developing our tools from scratch, we followed our own recommendation [10] to *extend and adapt existing well-developed standard tools*, by integrating FORML in the FeatureHouse framework. Our experience with FeatureHouse has been a positive one. One main challenge we faced was that composition rules could not be used to implement our canonical representation because the rules are only for merging terminals and clauses are nonterminals (so that they can be extended). Another challenge we faced initially was the lack of proper documentation on how to integrate a new language. Also, while FeatureHouse generates a lot of the tools automatically, there are currently still some manual tasks (e.g. adding a builder and printer classes and registering the new language) that from our experience can be automated.

V. Related Work

Although superimposition has been primarily applied to code [3], [11], its application to state-machine models has also been explored in AHEAD-based approaches [12]. However, all these approaches have limited or no support for explicitly modelling intended interactions (like priorities and overrides) in state-machine models which FORML supports. Other approaches such as DFC [13] and fSMV [14] use composition order to specify intended feature interactions, and this can result in unintended behaviour overrides.

Commutative composition has been proposed in other SPL development paradigms such as in delta-modelling [15], aspect-oriented programming [16] and action-based approaches[17]. In the feature-oriented software development paradigm, many existing approaches typically sacrifice commutativity for the sake of modelling intended interactions. Our approach is commutative and still supports explicit modelling of intended interactions.

Most of the languages currently integrated into FeatureHouse are programming languages (e.g. Java, C, Alloy, Python and Haskell) [3]. In [18], the Java Modelling Language (JML) was integrated into FeatureHouse for generating a metaproduct. Apel et al. [19] also explored the feasibility and expressiveness of superimposition as a model composition technique using FeatureHouse to compose UML class, state, and sequence diagrams. Our composition operation simplifies the generic merge operation defined in the FOSD algebra [7] by excluding the merging of common terminal nodes. Also the terminal nodes in our approach are more fine-grained than the terminal nodes in previous approaches that merge state machines thereby giving us more compositional expressiveness. Finally our composition in FeatureHouse is commutative which facilitates incremental composition of future features.

VI. Conclusion and Future Work

In this paper, we presented a technique using FeatureHouse for incremental and commutative composition of state-machine models of features, preserving the intended interactions in the composition. We discussed some of the main challenges in achieving a commutative composition of state-machines and our approaches to addressing these challenges. We are currently investigating family-based analysis of the SPL model that can be generated from this work. This involves translating the SPL models into the input language of an existing model checker and using model checking to detect unintended feature interactions.

References

[1] P. Shaker, J. M. Atlee, and S. Wang, “A feature-oriented requirements modelling language,” in *Requirements Engineering Conference (RE), 2012 20th IEEE International*, pp. 151–160, IEEE, 2012.

[2] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin, “Model checking lots of systems: efficient verification of temporal properties in software product lines,” in *Proc. Int. Conf. on Software Engineering (ICSE)*, pp. 335–344, 2010.

[3] S. Apel, C. Kästner, and C. Lengauer, “FeatureHouse: Language-independent, automated software composition,” in *Proc. Int. Conf. on Software Engineering (ICSE)*, pp. 221–231, 2009.

[4] A. van Lamsweerde, *Requirements Engineering: From System Goals to UML Models to Software Specifications*. Wiley, 2009.

[5] P. Shaker, *A Feature-Oriented Modelling Language and a Feature-Interaction Taxonomy for Product-Line Requirements*. PhD thesis, University of Waterloo, 2013.

[6] J. Rumbaugh, I. Jacobson, and G. Booch, *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004.

[7] S. Apel, C. Lengauer, D. Batory, B. Möller, and C. Kästner, “An algebra for feature-oriented software development,” Tech. Rep. MIP-0706, University of Passau, 2007.

[8] K. Czarnecki and M. Antkiewicz, “Mapping features to models: A template approach based on superimposed variants,” in *Generative Programming and Component Engineering*, pp. 422–437, Springer, 2005.

[9] S. Ben-David, B. Sterin, J. M. Atlee, and S. Beidu, “Symbolic model checking of product-line requirements using SAT-based methods,” in *Proceedings of the 37th International Conference on Software Engineering*, 2015. to appear.

[10] J. M. Atlee, S. Beidu, N. A. Day, F. Faghih, and P. Shaker, “Recommendations for improving the usability of formal methods for product lines,” in *Proc. of ICSE Workshop on Formal Methods in Software Engineering (FormaliSE)*, pp. 43–49, 2013.

[11] D. Batory, J. N. Sarvela, and A. Rauschmayer, “Scaling step-wise refinement,” *IEEE Trans. on Software Engineering*, vol. 30, pp. 355–371, 2004.

[12] S. Apel and C. Kästner, “An overview of feature-oriented software development,” *Journal of Object Technology*, vol. 8, pp. 49–84, 2009.

[13] P. Zave and M. Jackson, “A component-based approach to telecommunication software,” *IEEE Software*, vol. 15, no. 5, pp. 70–78, 1998.

[14] A. Classen, P. Heymans, P.-Y. Schobbens, and A. Legay, “Symbolic model checking of software product lines,” in *Proc. Int. Conf. on Software Engineering (ICSE)*, pp. 321–330, 2011.

[15] D. Clarke, M. Helvensteijn, and I. Schaefer, “Abstract delta modeling,” in *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE ’10, (New York, NY, USA), pp. 13–22, ACM, 2010.

[16] S. Gélineau, *Commutative Composition*. PhD thesis, McGill University, 2009.

[17] S. Mosser, M. Blay-Fornarino, and L. Duchien, “A commutative model composition operator to support software adaptation,” in *Modelling Foundations and Applications*, pp. 4–19, Springer, 2012.

[18] J. Meinicke, “JML-based verification for feature-oriented programming,” *Bachelorarbeit, University of Magdeburg, Germany*, pp. 2–21, 2013.

[19] S. Apel, F. Janda, S. Trujillo, and C. Kästner, “Model superimposition in software product lines,” in *Theory and Practice of Model Transformations*, pp. 4–19, Springer, 2009.