IEEE Copyright Notice

Published in: ***Proceedings of the International Requirements Engineering Conference (RE'12)***, September 2012

"A Feature-Oriented Requirements Modelling Language"

Cite as:

P. Shaker, J. M. Atlee and S. Wang, "A feature-oriented requirements modelling language," *2012 20th IEEE International Requirements Engineering Conference (RE)*, Chicago, IL, 2012, pp. 151-160.

BibTex:

```
@INPROCEEDINGS{6345799,
author={P. {Shaker} and J. M. {Atlee} and S. {Wang}},
booktitle={2012 20th IEEE International Requirements Engineering Conference (RE)},
title={A feature-oriented requirements modelling language},
year={2012},
pages={151-160},
month={Sep.},}
```

DOI: https://doi.org/10.1109/RE.2012.6345799

A Feature-Oriented Requirements Modelling Language

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

Shige Wang
General Motors Global R & D
30500 Mound Road, Warren, MI 48090
shige.wang@gm.com

***Abstract*—In this paper, we present a feature-oriented requirements modelling language (FORML) for modelling the behavioural requirements of a software product line. FORML aims to support feature modularity and precise requirements modelling, and to ease the task of adding new features to a set of existing requirements. In particular, FORML decomposes a product line's requirements into feature modules, and provides language support for specifying tightly-coupled features as model fragments that extend and override existing feature modules. We discuss how decisions in the design of FORML affect the evolvability of requirements models, and explicate the specification of intended interactions among related features. We applied FORML to the specification of two feature sets, automotive and telephony, and we discuss how well the case studies exercised the language and how the requirements models evolved over the course of the case studies.**

***Keywords*-Requirements modelling, software product lines**

I. Introduction

A software system is often thought of in terms of its constituent features, where each *feature* is a "coherent and identifiable bundle of system functionality" [1]. Moreover, *feature modularity* eases system development and evolution because features can be developed in isolation, in parallel, and by third-party vendors. Feature orientation is particularly relevant in the context of software product lines (SPLs), where families of similar products are understood, constructed, managed, and evolved in terms of their features. In fact, there is a *feature-oriented software development (FOSD)* [2] paradigm that advocates that features be treated as first-class entities throughout the lifecycle of a software system.

The downside of feature orientation is that, when deriving a product from a selection of features, engineers must consider how the features interact. Two features interact with each other when "one feature affects the operation of [the other] feature" [3]. Some features interact by design: for example, advanced cruise-control features are *designed* to supersede basic cruise control. Other features interact by accident as a consequence of operating in a shared context [4][5]. For example, a number of automotive features regulate brake-pressure fluid and, if uncoordinated, may perform conflicting actions. To be safe, the engineer needs to be able to understand and reason about the behaviours of features in combination.

We are investigating requirements modelling and analysis in the feature-oriented development of a SPL. In this paper, we focus on the modelling task. In future work, we will consider analyses of features to detect unintended feature interactions and explore the space of requirements of a SPL's products.

We propose a feature-oriented requirements modelling language (FORML), the goals of which are as follows:

Use existing standards and best practices: First, in accordance with Jackson and Zave's widely-accepted reference model for RE [6], the model of the requirements should be separate from the model of the problem world. Second, FORML should make use of standard software-engineering notations (e.g., the UML, feature models), to ease adoption by practitioners.

Feature modularity: Each feature should be modelled as a separate feature module. This property eases the task of tracing a feature to model(s) of its behavioural requirements, and enables independent development of feature modules.

Ease of evolution: Ideally, the task of adding a new feature to the requirements of a SPL should be additive, prompting little to no changes to the existing feature modules.

Support for modelling differences: It should be possible to express new features in terms of their *differences* from existing features – for example, as model fragments that extend existing features' models. Modelling a feature as a model fragment focuses the modeller's or reviewer's attention on the requirements being introduced by the fragment.

Explicit modelling of intended feature interactions: A new feature's requirements may include intended feature interactions: that is, changes (e.g., removals or replacements) to the behaviours of existing features. We advocate modelling intended feature interactions explicitly, so that they are more apparent to the modeller and the reviewer. This information can also be used to direct feature analyses to focus on detecting unintended interactions.

Commutative and associative composition: Some approaches (e.g., DFC [7] and AHEAD [8]) use the order of

composition to realize intended interactions, in that the location of a feature in a composition determines what features it overrides. In such techniques, intended interactions must be implicitly inferred from the composition order, rather than being explicitly specified. Worse, it means that additional unintended interactions can emerge from the overriding nature of the composition. As such, it may be necessary to consider multiple feature orderings to find one that yields desirable results – and the desired orderings may need to be recomputed when new features are added. If instead the composition operator is commutative and associative, the order of composition does not matter and the order in which the SPL evolves to include new features does not matter.

Precision: The language should be precise so that the models are amenable to analyses.

Existing approaches for specifying the behavioural requirements of features [9], [10], [11], [12], [13], [14] and approaches that support feature modularity in behavioural models [4], [7], [8], [15], [16], [17] fall short with respect to one or more of the above goals: they all either lack support for the modelling of intended interactions explicitly or are sensitive to the order in which feature modules are composed.

FORML overview and contributions: The contribution of FORML is to combine and adapt the best practices for modelling behavioural requirements with the feature modularity of FOSD. A FORML model is decomposed into two requirements views (see Figure 1):

- A **world model** is an ontology of concepts that describes the problem world of a SPL.
- A **behaviour model** is an extended finite state-machine model that describes the requirements for a SPL. The model's inputs are events and conditions about world phenomena (i.e., concepts and relationships in the world model), and its outputs are actions over world phenomena. A behaviour model is decomposed into feature modules, which describe the requirements for features of the SPL.

FORML is distinguished from existing RE and FOSD approaches in the following ways:

- A FORML world model includes *SPL* and *feature concepts*, which relate products and features to relevant problem-world phenomena. A **feature model** specifies the valid feature combinations of a SPL.
- FORML provides a systematic treatment for how new features evolve a state-machine model of requirements, with respect to added, removed, or replaced behaviours.
- FORML introduces language constructs for explicitly modelling intended interactions in state-machine models of feature requirements.
- The composition of feature modules in FORML is commutative and associative, while preserving intended interactions.

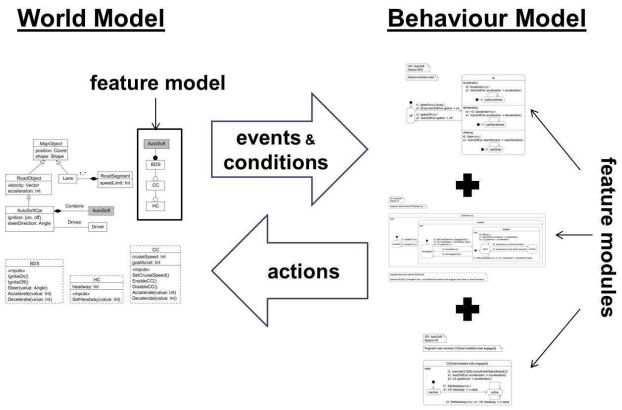

Figure 1. Overview of a FORML model

The rest of this paper is organized as follows. Sections II and III describe the components of a FORML model. Section IV describes how features are composed together. In Section V, we describe our experiences with applying FORML to families of automotive and telephony features. Section VI discusses related work in more detail, and we conclude in Section VII.

II. FORML World Model

A FORML world model is an ontology of the concepts in the problem world (*world* for short) of a SPL. In FORML, a problem is decomposed into features, and one could think about each feature as having its own problem world. However, to ensure a consistent ontology relevant to multiple features, a FORML world model is an integrated model of the features' problem worlds.

As a running example, we use a SPL of automotive software controllers, called *AutoSoft*, comprising the following features:

- *Basic driving service (BDS)*, which responds to commands to change the car's ignition state, acceleration, and steering direction
- *Cruise control (CC)*, which maintains the car's speed at a driver-specified cruising speed
- *Headway control (HC)*, which keeps the car at a driver-specified distance from road-objects ahead

Figure 2 presents an integrated world model of the three *AutoSoft* features.

As in existing approaches to modelling a problem world (e.g., KAOS [18] and Larman [19]), a FORML world model is a UML-based concept model: A *concept* in the world model represents a type of world phenomenon. A concept instance, called an *object*, is characterized by a set of properties, called *attributes*. A concept can be abstract and can have subtypes. There are special types of concepts; for example, an *association* represents a type of relationship that can exist among the instances of other concepts (e.g., automobiles can have drivers), and *compositions* and *aggregations* are

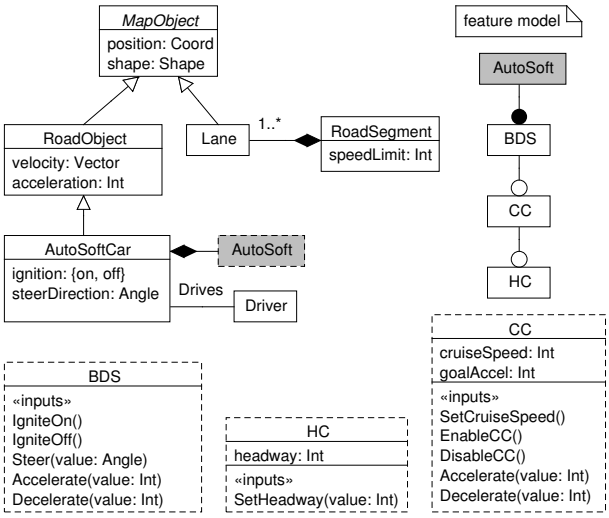

Figure 2. The *AutoSoft* world model

special binary associations that represent strong and weak whole-part relationships, respectively. The above constructs are expressed as a UML class diagram. Looking again at Figure 2, the *AutoSoft* world model includes *RoadSegment*s, which consist of one or more *Lane*s, and *RoadObject*s. A special type of road object is an *AutoSoftCar*, which is a vehicle (with a *Driver*) that incorporates *AutoSoft* products.

What distinguishes a FORML world model from existing approaches is that it includes the following:

SPL and feature concepts: A *SPL concept*, distinguished by a dashed border and a grey background (*AutoSoft* in Figure 2), represents a SPL's set of possible products. A SPL instance is a product with a particular *feature configuration* (i.e., set of features) represented by instances of the feature concepts. A SPL concept has one or more associations, which scope the world phenomena relevant to the SPL's products. For example, the scope of an *AutoSoft* product is the containing *AutoSoftCar* and its surroundings. A *feature concept*, distinguished by a dashed border (*BDS*, *CC*, and *HC* in Figure 2), specifies *feature phenomena* that is introduced by the SPL but is visible and possibly controllable by the world. Feature phenomena are either communications to or from a SPL product, or data shared between the product and the world. Communications are modelled as *messages* that are listed inside of relevant feature concepts (e.g., *IgniteOn()* is relevant to *BDS*). Feature data are modelled as attributes or associations of the feature concept that first introduces the data (e.g., *cruiseSpeed* is introduced by *CC*). Feature phenomena are organized by feature to improve traceability between features and their respective feature phenomena. This organization eases the task of updating the world model when a feature is removed from the SPL.

Feature model: A FORML world model includes a feature model that constrains the *valid* feature configurations of a SPL. FORML uses the original feature-model notation [20]. A feature model is depicted as a tree whose root is a SPL concept and every other tree node is a feature concept. A feature topped with a filled circle denotes a mandatory feature, and a feature topped with an empty circle denotes an optional feature. An optional feature can be present in a product only if its parent feature is present in the product. Hence, the *AutoSoft* feature model shown in the top-right of Figure 2 specifies that an *AutoSoft* product must include BDS, can optionally include CC, and if it has CC, it can optionally include HC.

A. World States and World-State Constraints

A world model identifies a space of *world states*, each representing a possible state of a SPL's problem world. A world state consists of a set of objects, their attribute values, their associations, etc. Additional domain knowledge and assumptions about the problem world may be specified as constraints on the world model. A *world-state constraint* restricts the space of possible world states defined by the world model. A *world-state transition constraint* specifies restrictions on consecutive world states; that is, it restricts how the world state can change. Such constraints are written as predicates in FORML's language for expressions over world states, described in Section II-B.

B. World-State Expressions

General world-state expressions: FORML has a language for writing expressions over the objects, attribute values, association links, etc. in a world state. This expression language is based on Alloy [21] and OCL [22], which provide widely-used operations over object models (e.g., navigation, filtering, queries). The grammar of Figure 3 shows the different types of FORML expressions. An expression is a parenthesized expression, set, predicate, integer, or *unspecified function or predicate* (rule 1); the latter is introduced to represent data logic or a constant value that is left unspecified in a FORML model (rule 17). A set (rule 2) can be empty ('none') or it can be a collection of objects (or values) in a world state (**ID**). A navigation expression starts from a set of objects and derives a set of related objects, links, or attribute values (rule 4). A set-selection expression defines a subset in terms of members that satisfy some predicate (rule 5). An *if-then-else* expression returns a set based on the value of a predicate (rule 6). There are operations on sets (e.g., union, cardinality, membership), integers (addition, subtraction), booleans (e.g., negation, conjunction, implication), and comparison operators (e.g., $=$, $<$). One can express predicates about the cardinality of a set (i.e., that a set contains zero, zero or one, one, or some elements) (rule 10). There are quantifiers for asserting that some number of a set's members satisfy a predicate (i.e., zero, zero or one, one, some, or all members) (rule 16). If a FORML expression is over two consecutive *before* and

```
01 expr := '(' expr ')' | expr '@pre' | set | pred | int | unspec
02 set := 'none' | ID | nav | select | cond | set set-op set
03 set-op := '+' | '&' | '-'
04 nav := set '.' ID
05 select := set '[' ID 'I' pred ']'
06 cond := 'if' pred 'then' set 'else' set
07 int := '#' set | int int-op int
08 int-op := '+' | '-'
09 pred := set-pred | logic | card-op set | quant | int int-comp int
10 card-op := 'no' | 'lone' | 'one' | 'some'
11 int-comp := '=' | '<>' | '>' | '<' | '<=' | '>='
12 set-pred := set '=' set | set 'in' set
13 logic := 'not' pred | pred logic-op pred
14 logic-op := 'and' | 'or' | 'implies' | 'iff'
15 quant := quant-op ID ':' ID 'I' pred
16 quant-op := 'no' | 'lone' | 'one' | 'some' | 'all'
17 unspec := ID '()' | ID '(' expr-list ')'
18 expr-list := expr | expr ',' expr-list
```

Figure 3. Syntax of FORML's language for general world-state expressions: *Non-terminal* and **terminal** symbols are denoted with different fonts (**ID** represents the name of a world-model element, variable, or unspecified function/predicate). Literals are enclosed in single quotes. "I" denotes choice.

after world states, then subexpressions with suffix *@pre* are evaluated with respect to the *before* world state.

In addition, FORML introduces the following constructs for expressing *events* and *actions* over world states.

World-change events (WCEs): A *world-change event* (WCE) is a notification of a primitive change to the world state, such as the addition or removal of an object, or a change in the value of an object's attribute. A WCE is expressed in one of the following basic forms, where **C** is a concept in the world model:

- **C+(o)** = object **o** of type **C** has just been added to the world state.
- **C-(o)** = object **o** has just been removed from the world state.
- **C.a~(o)** = **o**'s attribute **a** has just changed.

For example, **Accelerate+(o)** denotes the event of *AutoSoft* receiving an accelerate command.

World-change actions (WCAs): A *world-change action* (WCA) is an action performed by a product that effects a primitive change to the world state. A WCA can be expressed in one of the following basic forms, where **C**, **A**, and **M** are a basic concept, an association, and a message in the world model, respectively; and $\mathbf{exp/exp_i}$ and $\mathbf{o/o_j}$ are FORML expressions, where the latter expressions evaluate to object sets.

- $\mathbf{+C(a_1 = exp_1, ..., a_n = exp_n)}$ creates a **C** object whose attributes $\mathbf{a_i}$ have values $\mathbf{exp_i}$.
- $\mathbf{+A(a_1 = exp_1, ..., a_n = exp_n, r_1 = o_1, ..., r_m = o_m)}$ creates an **A** link that relates objects $\mathbf{o_j}$ in roles $\mathbf{r_j}$, and whose link attributes $\mathbf{a_i}$ have values $\mathbf{exp_i}$.
- $\mathbf{!M(p_1 = exp_1, ..., p_n = exp_n)}$ creates an **M** message object whose parameters $\mathbf{p_i}$ have values $\mathbf{exp_i}$.
- **-o** removes the objects **o** and their dependent links[1].
- **o.a := exp** changes the value of **o**'s attribute **a** to value **exp**.

For example, **c.ignition := on** denotes the action of turning on the ignition of an *AutoSoftCar* **c**.

[1]Note that object expression **o** may refer to a collection of objects (e.g., remove all the calls associated with a caller who has hung up).

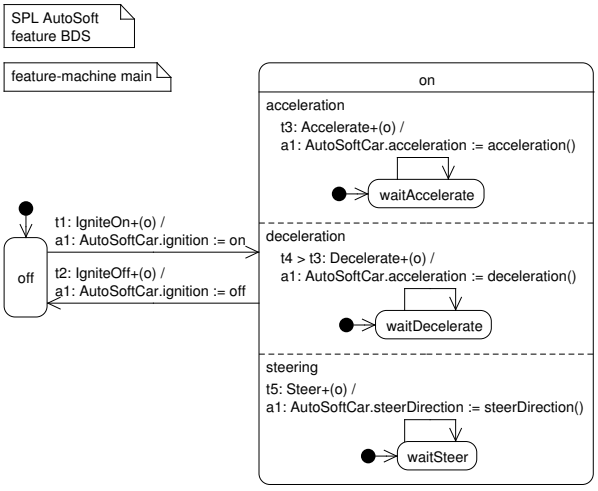

Figure 4. BDS feature module

III. FORML Behaviour Model

A FORML behaviour model is an operational specification of the requirements of a SPL's products. This specification is decomposed into *feature modules*, where each feature module specifies the requirements for a single feature of the products.

Following the AHEAD model of FOSD [8], FORML distinguishes between *base features* that constitute the initial requirements of a SPL and new features that evolve the SPL's requirements over time. The requirements of a base feature are expressed as one or more parallel state machines, and those of a new feature are expressed as state machines, or as state-machine fragments to be superimposed onto existing feature modules, or both. The requirements of a new feature can add behaviours, remove behaviours, or replace behaviours in the requirements of existing features.

A. Base-Feature Modules

The requirements of a base feature are expressed as one or more parallel UML-like state machines (called *feature machines*). Figure 4 shows the feature module of the *basic-driving service (BDS)* feature, which specifies basic driving behaviour, such as turning the car's ignition on and off, acceleration and deceleration, and steering. A feature module starts with a UML note that declares the name of the feature and the SPL to which it belongs. In addition, each feature machine (or machine fragment) in the module is prefaced with a UML note that declares the machine.

In general, a feature machine consists of

- A set of states, one of which is designated the initial state (by an arrow originating from a small black circle). Each state is either a *superstate*, which contains other states, or a *basic state*, which contains no other states. A superstate contains one or more *orthogonal regions*, where each region models a concurrent sub

state machine. Each sub machine may have a local initial state.

- A set of transitions between states. A transition has a *label* of the form

  id: e [c] / id$_1$: [c$_1$] a$_1$, ⋯, id$_n$: [c$_n$] a$_n$

  where **id** is the name of the transition; **e** is an optional triggering event; **c** is a guard condition; and **a$_1$**..**a$_n$** are concurrent actions, each with its own name **id$_i$** and guard condition **c$_i$**. If a guard condition is not specified, it is **true** by default.

In accordance with the Jackson and Zave RE reference model [6], a feature machine monitors and controls world phenomena: transition triggering events are normally world-change events, guard conditions are predicate expressions over the world state, and transition actions are world-change actions (see Section II-B). Thus, a transition t is enabled when the feature machine is in t's source state, and the current world state satisfies t's triggering event and guard condition. When t executes, the feature machine transitions to t's destination state and executes the actions of t whose guard conditions are satisfied in the current world state. For example, when the feature machine for BDS is in state *on* and (message) event *Accelerate* occurs, the car's intended acceleration is recomputed using the unspecified function *acceleration()* (transition *t3* in Figure 4)[2].

A triggering event may alternatively be a *time event*, with syntax **after(t)**. A time event occurs when the specified duration *t* has passed since the transition's source state was entered.

B. Evolving a SPL

The requirements of a new feature can *add* new behaviours, or can specify intended feature interactions that *remove* or *replace* behaviours in existing features. In all three cases, such requirements can be modelled as state-machine *fragments* that extend existing feature modules at specified locations.

1) Adding Behaviours: New behaviours can be specified by the following types of fragments:

- **A new region** that extends an existing state
- **A new transition** that extends an existing feature machine. Adding a transition may involving adding new states as the source or destination states.
- **A new action** that extends an existing transition
- **A weakening clause** that weakens the guard condition of an existing transition or action by appending another predicate through *disjunction*. Weakening a condition results in the guarded transition or action being executed more frequently, and therefore leads to added behaviours.

[2]The names of transitions, actions, and unspecified functions are unique throughout the containing feature machine, transition label, and FORML model, respectively.

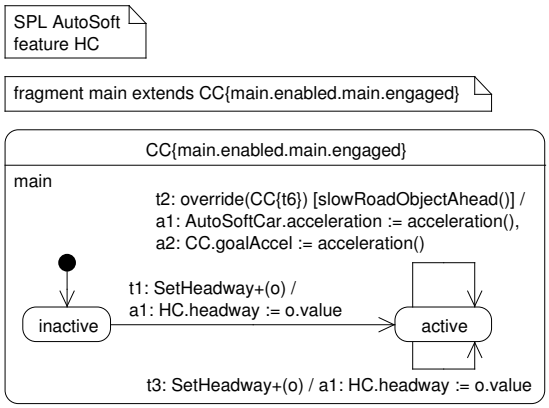

Figure 6. HC feature module

Consider Figure 5, which shows the feature module for *cruise control (CC)*. The enabling of CC and its subsequent behaviours — its activation, setting the cruise speed, suspending CC when the driver accelerates too much or applies the brakes — is only possible when the car is on. Thus, CC is modelled as a sub machine running in a new orthogonal region of *BDS*'s state *on*. The UML note that declares the fragment also specifies its context: state *BDS{main.on}* (i.e., state *on* of feature-machine *main* in feature-module *BDS*).

It is possible to have enhancements of enhancements: feature *headway control (HC)*, shown in Figure 6, is enabled only when CC is engaged and thus is modelled as a new orthogonal region in *CC*'s state *engaged*.

2) Removing Behaviours: A new feature may want to intentionally restrict the behaviours of existing features. This is specified by strengthening the guard condition of an existing transition or action with a new conjunct (called a **strengthening clause**). Strengthening a condition results in the guarded transition or action being executed less frequently, and therefore leads to removed behaviours. For example, whenever CC is active, BDS should not respond to *Accelerate* messages and should defer to CC's handling of these messages. This behaviour is specified as a fragment in *CC* (in the third UML note): the fragment strengthens the guard condition on *BDS*'s transition *t3*, which ensures that the transition executes only if *CC* is not in state *active* and the driver is not accelerating enough to kick *CC* out of state *active*[3].

3) Replacing Behaviours: A second form of intended interaction is where a new feature replaces existing behaviours – by specifying new transitions that have the same or similar enabling conditions as existing transitions, but have different actions. FORML introduces language constructs to specify such interactions explicitly:

- A **transition priority** specifies that a new transition,

[3]Strengthening and weakening clauses are named (e.g., the strengthening clause in *CC* is named "c") so that they can be referred to and extended by other strengthening and weakening clauses.

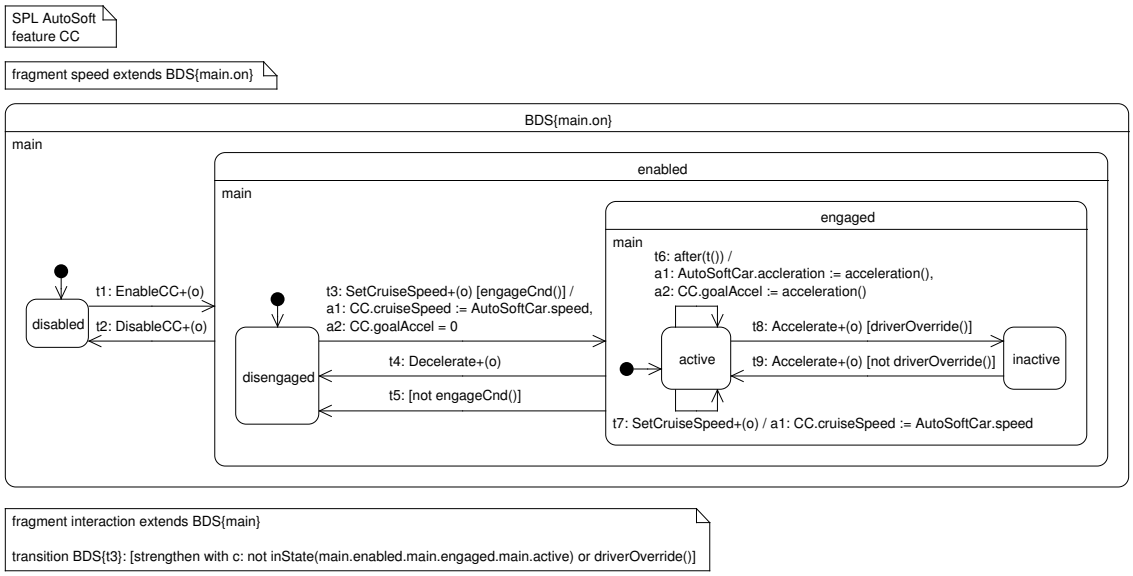

Figure 5. CC feature module

t2, has priority over an existing transition, *F{t1}* in feature-module *F*, whenever both are simultaneously enabled. The construct is used when specifying the new transition (the ellipsis refers to all of t2's details):

t2 > F{ t1} : ...

- A **transition override** specifies that a new transition, *t2*, that overrides an existing transition, *F{t1}* in feature-module *F*. An override differs from a transition priority in that the enabling condition of *t2* is implicitly the same as that of *t1*, but could be strengthened with an additional guard *c* (the ellipsis refers to t2's actions):

  t2 : override(F{t1}) [c] / ...

  Literally, this states that whenever **F**'s transition **t1** is enabled, **t2** executes instead — provided that the behaviour model is in **t2**'s source state and guard condition **c** is true.
- An **action override** specifies a new action, *a2*, that overrides an existing action, *F{a1}* in a transition *t* in feature-module *F* (the ellipsis refers to a2's action).

  F{t} : / a2: override(a1) [c] ...

  Literally, this states that whenever **F**'s transition **t** executes, action **a2** is performed in place of **a1** — provided that the guard condition **c** is true.

For example, *HC* in Figure 6 overrides *CC* whenever the car gets too close to the car ahead of it: *HC* (transition *t2*) overrides *CC* (transition *t6*) to perform its own computations of the intended acceleration, in order to maintain the desired headway distance.

The requirements of a new feature can always be specified as a separate feature machine, using FORML's behaviour-replacement constructs to model intended feature interactions. However, we generally model new features as fragments that extend other feature modules. Modelling extensions as fragments focuses the reader's attention on the new requirements introduced by the fragments. It also improves modifiability, because the requirements of the existing features being extended are not replicated in the new features. A new feature modelled as a fragment never removes model elements from existing feature modules (e.g., as in [23]). Rather, a fragment is a structural addition to the existing model, representing a variant behaviour due to the new feature. This design choice eases the evolution of FORML models in that the impact of new features is *additive*, so modified features do not suffer from dangling references to removed elements.

IV. Composing Feature Modules

The requirements for a SPL are derived by composing the feature modules in the SPL's behaviour model. The composition yields an *integrated model* comprising a set of parallel feature machines that have been extended with fragments. Figure 7 shows part of the integrated model for the *AutoSoft* SPL: *BDS*'s feature-machine *main* is extended with *CC*'s region *main*, which itself is extended with *HC*'s region *main*; and the guard on *BDS*'s transition *t3* is strengthened by feature *CC* with clause *c*. The region extensions and the strengthening clause are shown in grey. Note that the integrated model uses global names (e.g., *CC*'s transition *t1* is named *CC{t1}*).

The integrated model captures the requirements of a SPL and all of its valid configurations. Features that are optional in the SPL are expressed as requirements that are conditional on the feature's presence in a product. For example in Figure 7, the transitions and strengthening clause introduced by *CC* are guarded by presence condition *CC*: the guarded

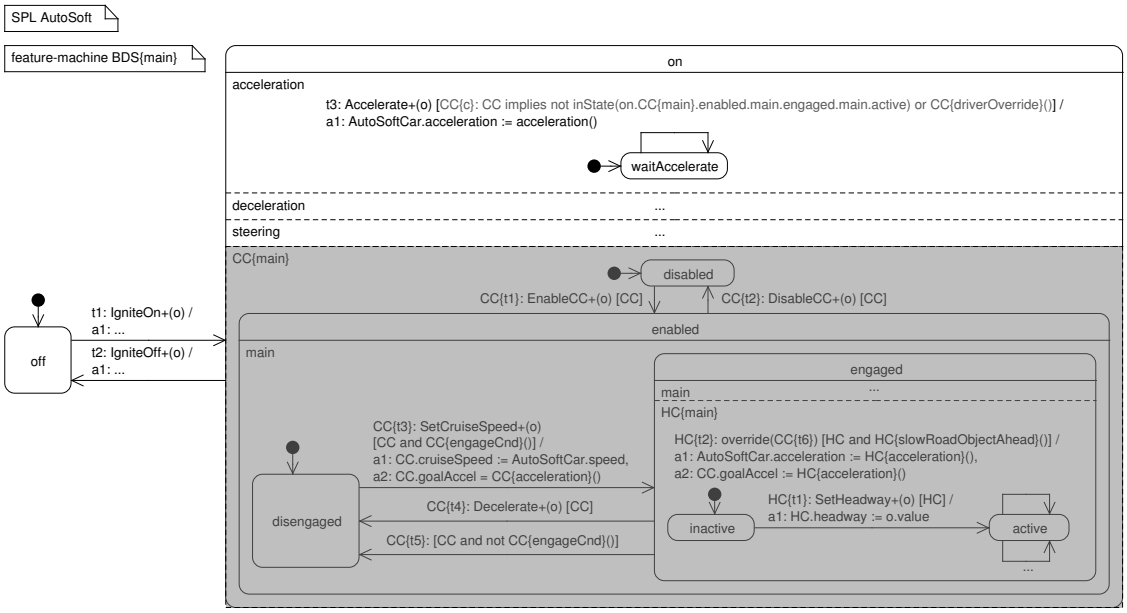

Figure 7. Integrated *AutoSoft* behaviour model ("..." in transition labels and regions elides portions of the model)

behaviours are among a product's behaviours iff *CC* is a feature in the product.

If multiple weakening and strengthening clauses extend the same guard condition, the way they are combined can affect the valuations of the resulting condition. We define a canonical composition of clauses:

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

where c is a guard condition that is extended by some number of weakening clauses $\{w_1, \cdots, w_n\}$ and strengthening clauses $\{s_1, \cdots, s_m\}$. Our canonical composition ensures that the set of possible valuations is insensitive to the order in which the features are composed. Moreover, it matches the operation precedence, of conjunction over disjunction, used in logic and programming languages.

We have designed the composition operator, which composes feature modules into an integrated model, to be commutative and associative. The proofs are based on the commutativity and associativity of superimposition when applied to *unordered* abstract syntax trees (of feature machines and of model fragments and their respective contexts). For example,

- 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[4]
- the *union* of concurrent actions in a single transition
- the *disjunction* of weakening clauses in a guard; where one disjunct is the *conjunction* of strengthening clauses and the original guard condition

More complete proof sketches are provided in [24].

[4]Transitions are ordered only when explicitly prioritized or overridden, and these orderings are insensitive to the order of feature composition.

V. Case Studies

FORML achieves most of its goals (listed in Section I) by design: it is based on the Jackson and Zave framework for RE and uses standard notations such as UML-like syntax and feature models; it supports modularity through feature decomposition, the modelling of differences through fragments, and the explicit modelling of intended interactions; and it provides a composition operator for feature modules that is commutative and associative. That FORML is precise enough to enable automated analysis is the topic of ongoing work.

With respect to the remaining goal of ease of evolution, FORML supports the additive modelling of new features in the behaviour model. However, the fact that a world model is shared by all feature modules can complicate its evolution. In particular, a new feature may change the world model in such a way that syntactically invalidates existing feature modules. For example, removing a world-model element can result in dangling references; and adding an attribute, association role, or message parameter can invalidate WCAs that create objects or links (because such actions list the values of the new object's or link's attributes, roles, or parameters).

We have performed two case studies, one from the automotive domain and one from the telephony domain, with the goals of (1) exploring the expressiveness of FORML, and (2) evaluating the impact of evolving a FORML world model with new features.

The automotive case study is an extension of *AutoSoft* and is adapted from a GM Feature Technical Specification for a family of automotive software features. In addition to the three features described in the paper so far, the case

Table I
SUMMARY OF CASE STUDIES

feature	add	remove	replace	world model	uf
automotive					
BDS (sm)				1c, 1a, 5m, 1e	3
CC	1r (BDS)	1sc (BDS)		3at, 3m	4
HC	1r (CC)		1to (CC)	1c, 1a, 1m, 1at, ref	2
LCA	1r (BDS)			3c, 1a, 3m, ref	1
FCA	1r (CC)			1at, 3m, 1e	1
HP	2a, 1t (HC)			1c, 2a	
SLC	1r (CC)		1to (CC)	1at, 2m	2
LCC	1a (BDS), 1r (CC)		1to (CC)	1at, 2m	4
LXC	1cs, 9t (LCC), 1wc (FCA), 2t (LCA)	1sc (FCA)	1to (LCC)	1at, 3m	5
RCA	1r (LCC), 1r (LXC)			1m	2
DMS	1r (LCC)			1at, 1m	
telephony					
BCS (sm)				1c, 3a, 4m	
CW	1bs, 2t (BCS)		3to (BCS)	1a, 1m	
CD	1a (BCS)			1m	
CDB		1sc (CD)			
CFB	2a (BCS)	1sc (BCS)		1a	
CT	1bs, 1cs, 5t (BCS)	1sc (BCS)	4to (BCS)	1a, 1m	
TWC	2bs, 1cs, 6t (BCS)	1sc (BCS)	3to (BCS)	2a, 2m	
GR	6a, 2t (BCS)			4a	
RBF	1a, 1t (BCS)			1a	
TL	1bs, 2t (BCS)		1to (BCS)	1a, 1at, 2m	2
TCS			1to (BCS)	1a	
VM	1a, 2t (BCS)	1sc (BCS)		3c, 3a, 1m, 1at, ref	2
billing (sm)				1c, 1a, 2at	2
RC			1ao (billing)		
SB	2a (billing)		1ao (billing)	1at	2

feature column:
LCA: lane-change alert, **SLC:** speed-limit control, **FCA:** forward-collision alert, **HP:** headway personalization, **LCC:** lane-centering control, **LXC:** lane-change control, **RCA:** road-change alert, **DMS:** driver-monitoring system, **BCS:** basic call service, **CW:** call waiting, **CD:** caller-number delivery, **CDB:** caller-number delivery blocking, **CFB:** call-forwarding on busy, **CT:** call transfer, **TWC:** three-way calling, **GR:** group ringing, **RBF:** ringback when free, **TL:** teenline, **TCS:** terminating call screening, **VM:** voice mail, **RC:** reverse charging, **SB:** split billing

add, remove, and replace columns:
Table entries list numbers of extensions to feature (F):
r: region, **bs:** basic state, **cs:** composite state, **t:** transition, **a:** action, **ao:** action override, **to:** transition override, **wc:** weakening clause, **sc:** strengthening clause

world model column:
Table entries list numbers of changes to world model:
c: basic concept, **a:** association, **m:** message, **e:** enumeration, **at:** attribute, **ref:** refactoring operation

study includes eight additional features. The adaptations of this case study were to remove design-level information from the original descriptions of the features, and to omit behaviours that were of the same nature as already-modelled behaviours and thus would not further exercise the expressiveness of FORML. The telephony case study is adapted from the Second Feature Interaction Contest [25], and comprises a *telephone-service* SPL with 15 features. The adaptions of this case study were minor adjustments to feature descriptions with the goal of better exercising FORML. The features are listed in the leftmost column of Table I.

The case studies were performed in *exploratory* and *confirmatory* phases. In the exploratory phase, the language was refined as needed to specify completely a small subset of the case studies' features (specifically, the BCS, CW, and CFB features in the telephony case study and the BDS, CC, and HC features in the automotive case study). In the confirmatory phase, the refined language was applied to the rest of the features to assess expressiveness.

Expressiveness: The language features developed during the exploratory phase were sufficiently expressive for the confirmatory phase. However, the confirmatory phase did result in language refinements to improve usability.

Table I gives summary information about the case-study models. In each case study, the SPL's requirements were modelled incrementally by adding features one at a time in the order in which they appear in Table I. The order was constrained by functional dependencies between feature types (e.g., LCC depends on CC), but was otherwise random. The requirements for BDS, BCS, and billing are each modelled as a single state machine, as denoted in the table by the postfix *(sm)*. The other features' requirements are modelled as fragments; the *add*, *remove*, and *replace* columns state the number and type of fragments used to specify each feature's added, removed, and replaced behaviours, respectively. The *world-model* column indicates how the world model changed with the addition of each feature, to reflect new concepts, attributes, associations, etc. (excluding changes to the feature model). Finally, the *uf* column gives the number of unspecified functions that were introduced to abstractly represent data logic and constants.

Ease of evolution: None of the world-model changes in the case studies syntactically invalidated existing feature modules. The addition of features HC and VM led to world-model refactorings that generalized some concepts into a supertype and some of their attributes/associations moved to the supertype; however, references to the moved attributes/associations remained valid, because these elements are inherited by the subtypes. In the case of LCA, the refactoring modelled some phenomena in an alternative way, which involved removing some concepts; in this case, we were lucky because unspecified functions abstracted away any dependence on the refactored portion of the world model. In general, we believe that the impact due to evolution is small: the removal of world-model elements is not likely to occur frequently, and the invalidation of existing WCAs can be avoided by assigning default values to new attributes, roles, and parameters.

Due to space constraints, the models for the case studies are not included in this paper. The telephony models can be found in [24], and the automotive models are included in a GM report that is still undergoing the GM internal approval process.

VI. RELATED WORK

Requirements modelling: FORML adapts and extends behavioural modelling methods that follow the Jackson and Zave RE reference model [6] (e.g., KAOS [18], Fusion [26], Larman [19], and SCR [27]). Our main extension is to make

features and their intended interactions explicit. Furthermore, we chose not to adopt the formal models of behaviour in SCR and KAOS: SCR and KAOS models are decomposed into fine-grained units (e.g., variables in SCR, and operations in KAOS), such that further decomposition into features would lead to a high scattering of feature behaviours across these units. Instead, we added precision to the semiformal use of UML state-machine diagrams.

Feature-oriented descriptions of SPL requirements: The idea of modelling the common and variable requirements of a SPL's products as features was introduced in Kang et al.'s feature-oriented domain analysis (FODA) method [20]. FODA is best known for introducing feature models but it also proposes describing a SPL's requirements using a combination of an ER model, and an integrated behavioural model parameterized by features. FODA does not prescribe a particular behavioural modelling language, but [20] gives an example that uses statecharts. However, the statechart and ER models in the example are not interrelated. In subsequent work, feature models have been integrated with several requirements-description techniques, including use-cases (in [9], [10]), use-case maps (in [11]), and goals and scenarios (in [12]). However, such approaches do not follow the Jackson and Zave RE reference model, do not result in precise models, and do not support explicit modelling of intended feature interactions.

Decomposing state-machines into feature modules: FORML supports both parallel composition of feature machines and structural composition of machine fragments. FORML uses superimposition for structural composition. Although superimposition has been primarily applied to code (e.g., [8], [28]), its application to state-machine models has also been explored in AHEAD-based approaches [8], [15] and in approaches for detecting unintended feature interactions [17], [16]. However, the latter approaches have limited [15], [16] or no support [17], [8] for explicitly modelling intended feature interactions in state-machine models, and in some approaches [15], [16] the order of composing feature modules matters. Intended feature interactions are realized by transitions that override existing transitions that have the same name [15], or whose actions affect the same variables [16]. However, there is no indication of whether a new transition *is supposed to be* an overriding transition, and if so, *which* transition(s) are overridden. Also, in Apel et al.'s approach [15], it is not possible to explicitly specify conditional overrides. Jayaraman et al. [23] propose using graph transformations applied to model optional features as transformations to an integrated model of mandatory features. However, in this approach, mandatory features are not separately described, the order in which feature modules are composed matters, and behaviour replacement cannot be explicitly modelled.

Two prominent approaches for composing feature machines are layered composition (e.g., DFC [7]), in which feature machines react to events in some order and can control the events seen by subsequent machines; and a variant of parallel composition (e.g., [17], [4]), in which an explicit (e.g., [4]) or implicit (e.g., [17]) precedence relation between features resolves conflicts between their machines during execution. However, such approaches do not support the explicit modelling of intended feature interactions. Also, in layered-composition approaches, the mechanism for specifying intended interactions (i.e., controlling events sent to previously composed machines) depends on the composition order, and can result in unintended overrides among seemingly unrelated features.

Liu et al. [29] propose composing feature machines by specifying a set of transitions between the states of the feature machines. However, in this approach, explicit models of intended interactions are not supported and the order of composition matters.

Features and the RE reference model: The notion of features has been previously explored in the context of the RE reference model. Classen et al. [13], Nhlabatsi et al. [5], and Tun et al. [14] consider a feature to be a triple comprising a set of requirements, a set of problem-world properties, and a specification. Nhlabatsi et al. do not prescribe a notation for expressing the behaviours of features; Classen et al. and Tun et al. use event calculus. In contrast, our aim was to use standard notations, such as the UML, to ease adoption by practitioners. Furthermore, these approaches do not support explicit modelling of intended feature interactions.

Other approaches for specifying SPL behaviours: Larsen et al. [30] propose a theory where the interfaces of product-line assets are modeled as modal I/O automata. However, the approach is not feature oriented. Lauenroth et al. [31], Czarnecki et al. [32], and Classen et al. [33] propose methods involving integrated models of SPL behaviours with mappings from features/variants to the model elements that describe them. However, such approaches do not support feature modularity and have limited [33] to no [31], [32] support for explicit models of intended interactions. Classen et al. model such interactions using transition priorities; however, conditional priorities are not supported.

VII. Conclusions and Future Work

We have presented FORML, a language for modelling the requirements for features in a SPL. FORML integrates and adapts best practices from RE and from FOSD, focusing on behavioural models of feature modules. The distinguishing aspects of FORML are (1) the inclusion of feature phenomena and feature configurations in a SPL's problem world; (2) a systematic treatment of how new features evolve a requirements model, with respect to added, removed, or replaced behaviours; (3) language constructs for explicitly

modelling intended interactions among state-machine models of features; and (4) an operator for composing feature modules that preserves intended feature interactions, yet is commutative and associative. Other advantageous properties of FORML include a UML-like syntax, to ease adoption; and the ability to model new features as model fragments, in order to focus on the feature's essential requirements.

We are currently investigating analyses of FORML models. We are interested both in detecting unintended interactions among features and in exploring the configuration space of a SPL. This work entails determining how unintended feature interactions manifest themselves in FORML models, so that we know what properties and patterns the analyses should look for. It also means modifying analysis tools to so that they adhere to our semantics of feature composition.

REFERENCES

[1] C. R. Turner, A. Fuggetta, L. Lavazza, and A. L. Wolf, "A conceptual basis for feature engineering," *JSS*, vol. 49, no. 1, pp. 3–15, 1999.

[2] S. Apel and C. Kästner, "An overview of feature-oriented software development," *JOT*, vol. 8, pp. 49–84, 2009.

[3] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. E. Herman, and L. Y-J, "The feature interaction problem in telecommunication systems," in *SETSS*, 1989, pp. 59–62.

[4] J. D. Hay and J. M. Atlee, "Composing features and resolving interactions," in *SIGSOFT FSE*, 2000, pp. 110–119.

[5] A. Nhlabatsi, R. Laney, and B. Nuseibeh, "Feature interaction as a context sharing problem," in *ICFI*, 2009, pp. 133–148.

[6] M. Jackson and P. Zave, "Deriving specifications from requirements: an example," in *ICSE*, 1995, pp. 15–24.

[7] P. Zave and M. Jackson, "A component-based approach to telecommunication software," *IEEE Software*, pp. 70–78, 1998.

[8] D. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling stepwise refinement," *IEEE TSE*, vol. 30, pp. 355–371, 2004.

[9] G. Chastek, P. Donohoe, K. C. Kang, and S. Thiel, "Product Line Analysis: A Practical Introduction," Software Engineering Institute, Carnegie Mellon University, Tech. Rep. CMU/SEI-2001-TR-001, 2001.

[10] M. Griss, J. Favaro, and M. d' Alessandro, "Integrating feature modeling with the RSEB," in *ICSR*, 1998, pp. 76–85.

[11] T. J. Brown, R. Gawley, I. T. A. Spence, P. Kilpatrick, C. Gillan, and R. Bashroush, "Requirements modelling and design notations for software product lines," in *VaMoS*, 2007, pp. 27–35.

[12] S. Park, M. Kim, and V. Sugumaran, "A scenario, goal and feature-oriented domain analysis approach for developing software product lines," *IMDS*, vol. 104, pp. 296–308, 2004.

[13] A. Classen, P. Heymans, and P. Schobbens, "What's in a feature: A requirements engineering perspective," in *FASE*, 2008, pp. 16–30.

[14] T. T. Tun, T. Trew, M. Jackson, R. Laney, and B. Nuseibeh, "Specifying features of an evolving software system," *SPE*, vol. 39, pp. 973–1002, 2009.

[15] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model superimposition in software product lines," in *ICMT*, 2009, pp. 4–19.

[16] M. Plath and M. Ryan, "Feature integration using a feature construct," *Sci. Comput. Program.*, pp. 53–84, 2001.

[17] R. J. Hall, "Feature combination and interaction detection via foreground/background models," in *FIW*, 1998, pp. 449–469.

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

[19] C. Larman, *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process (3rd Edition)*. Prentice Hall, 2005.

[20] K. C. Kang, S. G. Cohen, J. A. Hess, W. E. Novak, and A. S. Peterson, "Feature-oriented domain analysis (FODA) feasibility study," Carnegie-Mellon University Software Engineering Institute, Tech. Rep. CMU/SEI-90-TR-21, 1990.

[21] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[22] OMG, *Object Constraint Language, OMG Available Specification, Version 2.0*, 2006. [Online]. Available: http://www.omg.org/docs/formal/06-05-01.pdf

[23] P. K. Jayaraman, J. Whittle, A. M. Elkhodary, and H. Gomaa, "Model composition in product lines and feature interaction detection using critical pair analysis," in *MoDELS*, 2007, pp. 151–165.

[24] P. Shaker and J. M. Atlee, "A feature-oriented requirements modelling language," David R. Cheriton School of Computer Science, University of Waterloo, Tech. Rep. CS-2012-05, 2012.

[25] M. Kolberg, E. H. Magill, D. Marples, and S. Reiff, "Second feature interaction contest," *FIW*, pp. 293–310, 2000.

[26] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*. Prentice Hall, 1994.

[27] C. L. Heitmeyer and R. D. Jeffords, "The SCR tabular notation: A formal foundation," Naval Research Lab, Tech. Rep. NLR/MR/5546-03-8678, 2003, nLR/MR/5546-03-8678.

[28] S. Apel, C. Kästner, and C. Lengauer, "Featurehouse: Language-independent, automated software composition," in *ICSE*, 2009, pp. 221–231.

[29] J. Liu, S. Basu, and R. R. Lutz, "Compositional model checking of software product lines using variation point obligations," *ASE*, pp. 39–76, 2011.

[30] K. G. Larsen, U. Nyman, and A. Wąsowski, "Modal i/o automata for interface and product line theories," in *ESOP*, 2007, pp. 64–79.

[31] K. Lauenroth and K. Pohl, "Dynamic consistency checking of domain requirements in product line engineering," in *RE*, 2008, pp. 193–202.

[32] K. Czarnecki and K. Pietroszek, "Verifying feature-based model templates against well-formedness OCL constraints," in *GPCE*, 2006, pp. 211–220.

[33] A. Classen, P. Heymans, P. Schobbens, A. Legay, and J. Raskin, "Model checking lots of systems: efficient verification of temporal properties in software product lines," in *ICSE*, 2010, pp. 335–344.