DOI: https://doi.org/10.1109/RE.2013.6636708

DOI: https://doi.org/ 10.1109/RE.2013.6636708

A Mode-Based Pattern for Feature Requirements, and a Generic Feature Interface

David Dietrich and Joanne M. Atlee
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Canada
{d4dietri,jmatlee}@uwaterloo.ca

***Abstract*—In this paper, we propose a pattern for decomposing and structuring the model of a feature's behavioural requirements, based on modes of operation (e.g., *Active, Inactive, Failed*) that are common to features in multiple domains. Interestingly, the highest-level modes of the pattern can serve as a generic behavioural interface for all features that adhere to the pattern. We have applied the pattern in modelling the behavioural requirements of 19 automotive features that were specified in 5 production-grade requirements documents. We found that the pattern was applicable to all 19 features, and that our proposed generic feature interface was applicable to 50 out of 57 inter-feature references.**

***Index Terms*—requirements engineering; requirements patterns**

I. Introduction

Requirements patterns are reusable solutions that assist the engineer by providing advice on how to decompose a requirements problem into parts and how those parts interact with one another. The main benefits of using patterns are *efficiency* in eliciting or documenting the requirements problem, and *predictability*, in knowing that the resulting requirements specification should be nearly as good as previous specifications derived from the same patterns.

Our work investigates patterns in feature-oriented software requirements, where each feature of a software system or product line is specified as a separate module; otherwise, our work is not specific to any particular domain. We propose a pattern that decomposes and structures the behaviour model of a feature, expressed as a state machine, according to modes of operation that are common to features in all domains. High-level modes include *Active* (which captures a feature's essential requirements), *Inactive* (which captures a feature's enabling and disabling requirements), and *Failed* (which captures a feature's failure and recovery requirements). Figure 1 depicts the pattern's high-level modes and their transitions. Section III further decomposes the pattern into common sub-modes of operation. When features are modelled according to the pattern, *all* features have the same high-level behaviour model. As a result, features have a common structure, which eases the task of reading and reviewing the models of multiple features. Moreover, the modes decompose the task of writing the feature's detailed requirements (e.g., separating a feature's enabling/disabling requirements from its active requirements).

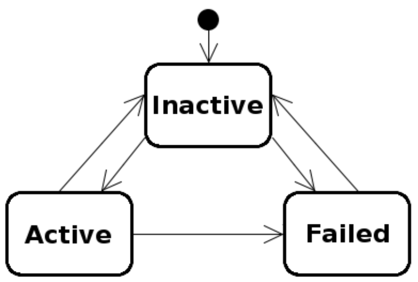

Fig. 1. A pattern for feature behavioural requirements, based on their modes of operation.

An important side effect of the pattern is that the common modes can serve as a public interface of a feature. Traditionally, models of features do not have interfaces: features are either modelled as if they have no knowledge of each other and interact solely through their shared context (e.g., through common input/output signals or shared environmental variables) [14], [19]; or features have complete knowledge about each others' details, such that they can monitor, extend, or override one another [2], [4], [5], [25]. If features had interfaces [16], the interface could export useful information about the feature without revealing detailed behaviour. Moreover, the pattern's modes could serve as a *generic* behavioural interface for all features that adhere to the pattern. Such an interface would ease the tasks of specifying and reviewing new features because references to other features' interface modes would be guaranteed to be consistent with those features' state-machine models.

To assess the pattern, we performed a case study in which we modelled the requirements of 19 automotive features, as described in 5 requirements documents provided by a major automotive manufacturer. The goal of the case study was to determine whether the pattern is naturally applicable in practice, and whether it is natural to use the pattern's modes as a generic feature interface. We were able to create behaviour models of all 19 features according to the pattern, with one minor deviation. Moreover, we were able to model the vast majority of inter-feature references as references to information provided in features' interfaces.

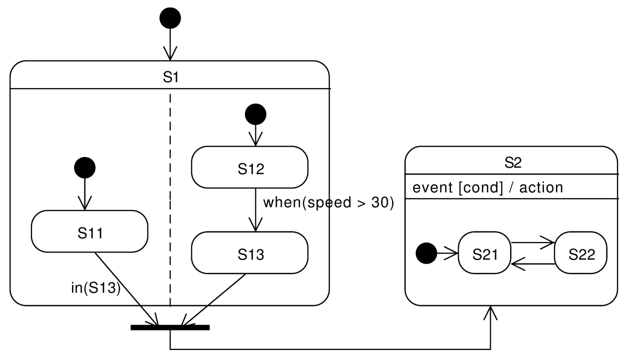

Fig. 2. State-machine notation.

The rest of this paper is organized as follows. Section II provides a brief overview of the notation that we use to model features. In Section III, we introduce the behaviour pattern and sub-patterns that we have devised, as well as some variants and a small example. In Section IV, we discuss how the pattern's modes can serve as a generic feature interface. We present the results of our case study in Section V and discuss perceived benefits in Section VI. We summarize related work in Section VII and conclude in Section VIII.

II. Background

Feature-oriented requirements decompose the requirements of a system or a product line into distinct feature modules. Each feature is a "coherent and identifiable bundle of system functionality" [27] that is specified in isolation, can be developed as an independent increment to the system or product line, and may be optional in the final product. The context of a feature includes other features in the system, so a feature model may reference other features. In this paper, we are interested in behaviour models of individual features that are expressed using hierarchical state machines (e.g., statecharts [11], UML State Machines [22]).

Figure 2 shows a state-machine model that exhibits notation worth reviewing. A state may contain sub-states; in this case, the former is called a *superstate* (e.g., S2). A superstate may be decomposed into one or more *concurrent regions* that are separated by dashed lines (e.g., S1); regions model orthogonal behaviour that can occur in parallel. A transition from a black circle to a state designates the initial state of a machine or sub-machine (e.g., S1, S11). If a transition's destination is a superstate (e.g., S2), then the next state is the initial state (e.g., S21) of the superstate's sub-machine (or the initial states of the superstate's regions).

Transitions are annotated with an event; a guard condition, normally on environmental conditions; and a set of actions on environmental conditions – all of which are optional.* Transition annotation *when(c)* [22] refers to the event of condition *c* becoming true. Transition annotation [*in(S)*] [11] is a condition that is satisfied when the system's execution is in state *S*; state *S* might refer to a state in another feature. The *join* pseudo-state (modelled as a black bar) is used to aggregate multiple transitions (e.g., the transitions from source states S11 and S13 to destination state S2). The join's outgoing transition executes only when all of its incoming transitions are enabled.

A state may be annotated with actions that are enabled by events and guard conditions (e.g., S2). Such actions are performed whenever their triggering event occurs while the system's execution is in the state and the guard condition is true.

III. The Pattern

Our research group is collaborating with a major automotive manufacturer on precise modelling and analysis of feature requirements. Through this collaboration, we have been granted access to production-grade requirements documents of several automotive features. Over the course of creating formal models of those features, we found that different features tend to have the same major modes of operation and similar enabling processes.

In the original requirements documents, a feature's required behaviour is primarily described in natural language accompanied by tables that provide information about environmental variables and input/output signals. We consider a feature to be **complex** if its enabling process progresses through multiple stages, or if it has multiple conditional behaviours once it is active. The description of a complex feature is often supplemented with a state-machine that models the feature's high-level behaviour. Such state machines rarely specify the active behaviour for the feature, and the transitions tend to specify only a few of the enabling conditions and no actions. We also found that there were slight differences in the notation used to model different features (e.g., states and transitions appear differently, and transitions were color-coded in some features but not in others).

Consider the state machines in Figure 3. These are anonymized versions of models found in the original requirements documents[†]. Feature **A**, depicted in Figure 3a, initializes in state Disabled and proceeds through its enabling process (states Standby Disabled and Standby Enabled) to the Engaged state, which represents all of the feature's active behaviour. Details of how the feature behaves when it is Engaged are not modelled as a part of the state machine. Lastly, the Disengaging state is entered when feature **A** is becoming disabled, but is not yet fully disabled. Feature **B**, depicted in Figure 3b, initializes in state 0 (On), indicating that the feature immediately starts to affect the system's behaviour. Again, the feature's behaviour while On is not included in the state machine. The feature transitions from 0 (On) to 1 (Partial On) if the user presses the disable button, at which point part of the feature's functionality is disabled. The feature transitions

*In this paper, most transition annotations are omitted to avoid revealing proprietary information about features.

[†]We have removed all feature-identifying names and all transition labels, and retained the original model structure and notation.

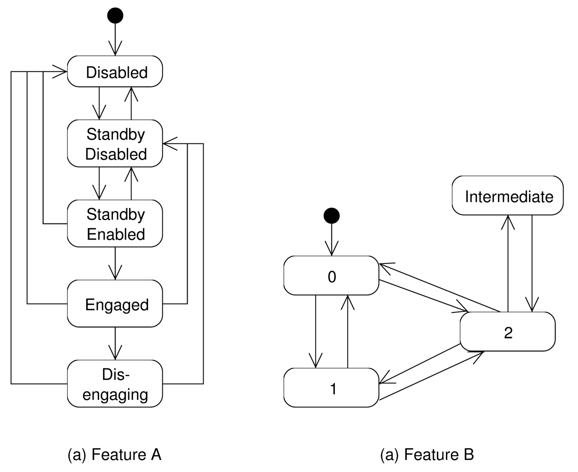

Fig. 3. Two anonymized state-machine models from production-grade requirements documents.

from 0 (On) to 2 (Off) if the user presses and holds the disable button, at which point all of the feature's behaviour is disabled. The feature transitions from 2 (Off) to 1 (Partial On) under certain environmental conditions, and transitions from 2 (Off) to 0 (On) when the user enables the feature. The Intermediate state provides the same behaviour as state 0 (On), and is used as a temporary override that activates the feature in the event of an emergency.

There are several interesting observations that can be made about features **A** and **B**.

- It is not uncommon for documents to use slightly different terminology, or for the same term to be used differently, across multiple documents (e.g., an activated feature is state On vs. Active vs. Engaged). This kind of inconsistent naming is not likely to be serious, but can lead to minor ambiguities and confusion. Domain knowledge gained from reading one feature's requirements does not ease the task of reading other features' requirements.
- State-machine models tend to be simple and non-hierarchical. Of the 19 feature requirements that we examined, 9 included state-machine models, and of these, 4 used hierarchy to describe some aspect of the active behaviour for the feature.
- The state machines often omit failures and recovery, although the textual requirements mention how (or at least that) the feature can fail. Of the 19 feature requirements, 11 requirements mention the possibility of the feature failing, 8 requirements specify a feature's failure and recovery conditions, and 4 of the 9 state-machine models include failure states. The feature-requirements documents are sometimes light on failure requirements because many features (and all safety-critical features) have a separate safety-requirements document that describes how the feature behaves in the presence of failures. We did not have access to any safety-requirements documents.
- Despite their many differences, the features have the same basic modes of operation: (1) active, in which the feature affects system behaviour; (2) becoming enabled; and (3) failed. These similarities in the features' behaviours prompted us to propose a pattern for structuring a feature's behaviour model in a way that explicates the similarites. Further study of the feature requirements suggested ways of decomposing and structuring the sub-behaviour of how features become enabled.

In the remainder of this section, we present the pattern, its variants, and some examples.

A. *High-Level Structure*

A feature whose behaviour model adheres to the proposed pattern is called an **adherent feature**. Figure 1 shows the high-level structure of the pattern, which consists of three states and their transitions. Each of the high-level states contains one or more sub-machines that model the detailed behaviour for the state.

The **Inactive** state captures all aspects of the feature as it becomes enabled. As will be seen, we propose sub-patterns for modelling complex enabling conditions. Normally, a feature initializes in Inactive. When the feature is completely enabled, it transitions to the Active state, in which the feature performs its essential behaviour.

The **Active** state records all aspects of the feature actively affecting the behaviour of the system. We hypothesize that features' Active behaviours are highly specific to the feature and are not amenable to being clustered into reusable sub-patterns. The transition from the Inactive state to the Active state is defined by the pattern and depends on the variant of the Inactive sub-pattern being used. The conditions that trigger the transition from Active to Inactive are feature specific, and thus are not part of the pattern. Normally, transitions to Inactive are from the boundary of the Active superstate, meaning that the enabling process starts from the initial state(s) of Inactive's sub-machine(s).

The **Failed** state captures all aspects of how the feature behaves when it has failed. The exact conditions under which a feature transitions to or recovers from the Failed state are feature specific and are not part of the pattern. On recovery, the feature transitions to the boundary of the Inactive superstate. The pattern does not decompose the internal structure of the Failed state because features typically do nothing more than monitor the environment while failed (if a feature provides degraded service under some failure conditions, we model that behaviour within the Active state).

The pattern's high-level states correspond to distinct major modes of operation. Consider the different ways in which a feature can interact with its environment:

- the feature monitors the environment
- the feature acts on the environment
- the environment monitors the feature
- the environment acts on the feature

Each state reflects different types of interaction (as depicted in Figure 4 using colour). In restricted (red) states, the only

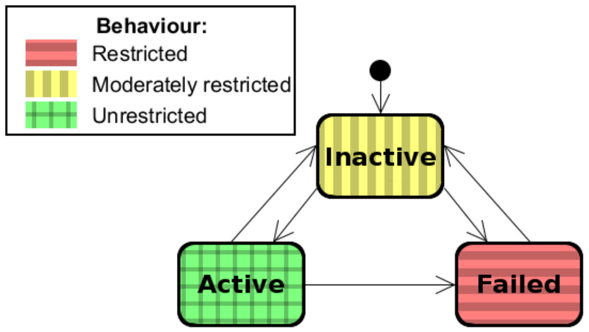

Fig. 4. Feature pattern, where the modes are coloured with the degree to which the feature's interactions with its environment are restricted.

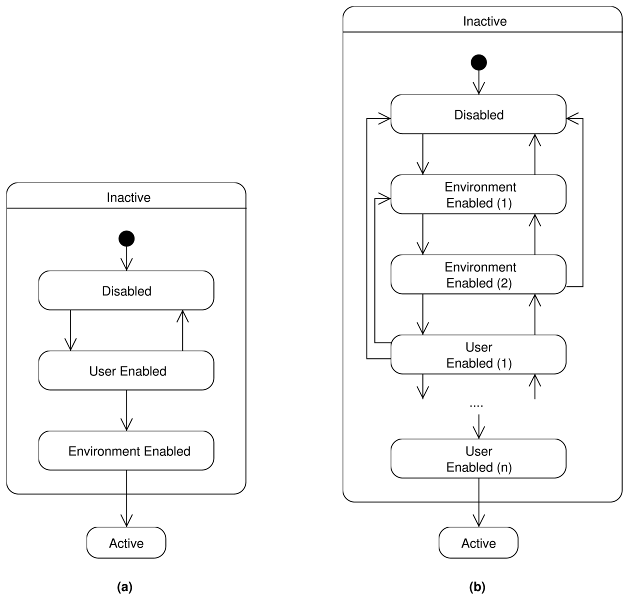

Fig. 5. The Ordered Enabling sub-pattern: (a) one of the default sub-patterns, (b) a multi-stage example of the sub-pattern.

allowable interaction is that the feature can monitor the environment – to determine if any of the state's outgoing transitions are enabled. For example, a feature that has Failed can monitor the environment for signs that recovery conditions have been met. In moderately restricted (yellow) states, the feature can monitor the environment and the environment can act on the feature. For example, it may be possible for a user to manipulate feature settings when the feature is still Inactive (e.g., a driver can set the cruising speed before the cruise-control feature becomes Active). In unrestricted (green) states, all four types of interactions are allowed. In this manner, the pattern's high-level states partition the features' behaviours into separate modes of operation.

B. *Inactive Sub-Patterns*

Sub-patterns for the Inactive mode provide advice on how to decompose and structure the enabling process of a feature, according to the type and order of enabling conditions. There are two types of enabling conditions: user actions and environmental conditions. A **user action** is an action performed directly by the user or human operator (e.g., the user turning on the feature). An **environmental condition** is a predicate over properties of the operating environment of the system or properties of the current state of the system (e.g., an automobile's speed or the state of another feature[‡]). We believe that a feature's behaviour model should prominently distinguish between user actions and environmental conditions because of the types of constraints each places on the feature design: each user action must have a corresponding user interface; whereas monitored properties must have corresponding sensors, and controlled properties must have corresponding actuators.

Because the conditions for transitioning among the high-level states tend to be feature specific, the pattern says nothing about the labels on those transitions. We have devised three sub-patterns that are based on the degree to which a feature's enabling conditions are ordered:

- There is a sequential ordering on enabling conditions.
- There are no ordering constraints on enabling conditions.
- Enabling conditions are partially ordered.

1) Ordered Enabling: This sub-pattern applies when a feature becomes enabled in stages. The Inactive sub-machine is a sequence of user actions and environmental conditions that must be satisfied in the specified order. Figure 5a shows the default sub-pattern where user actions precede environmental-condition checks. There is a second default sub-pattern (not shown) where environmental conditions must hold before user actions are recognized. In the sub-patterns, each transition can be triggered by a combination (i.e., conjunctions, disjunctions, negations) of user actions or a combination of environmental conditions. When the final state in the sequence is reached, the feature transitions to the Active state.

The Inactive state in Figure 5b uses the Ordered Enabling sub-pattern to specify a multi-stage enabling process. The name of each state is the type of the most recent combination of enabling conditions (user action or environmental conditions) and the sequence number for that type of condition. The enabling sequence may include back transitions from later states in the sequence to earlier states, if enabling conditions became unsatisfied and cause the feature to revert to a less-enabled state. As in the default sub-pattern, when the final state in the sequence is reached, the feature transitions to the Active state.

2) Unordered Enabling: It often does not matter in what order a feature's enabling conditions become true; as soon as they all hold, the feature becomes Active. The Unordered Enabling sub-pattern applies in these situations (shown in Figure 6). The concurrent regions separate user actions (on the left) from the environmental conditions (on the right). A transition can be triggered by a combination of user actions

[‡]Recall that a feature's environment includes other features in the system.

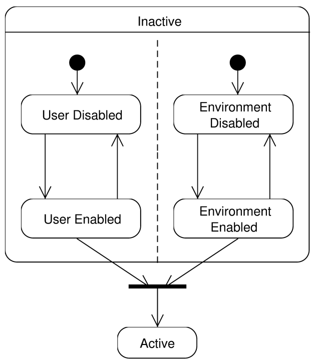

Fig. 6. The Unordered Enabling sub-pattern.

or a combination of environmental conditions. When all of the regions are simultaneously in their most-enabled state, the feature transitions to the Active state. We model this behaviour using a join pseudo-state whose source states are User Enabled and Environment Enabled and whose destination state is Active.

We recommend using the Unordered Enabling sub-pattern when a feature's enabling process includes only user actions or only environmental conditions, and not both. In such a case, the region that has no enabling conditions simply initiates in its enabled sub-state. This makes it explicit that the user actions or environmental conditions have been considered, but that none exist.

3) Hybrid Enabling: This sub-pattern applies when only part of the enabling process is ordered. The sub-pattern uses a mixture of concurrent regions and state sequencing to model the absence and presence, respectively, of ordering constraints. A simple example would be a variant of the Unordered Enabling sub-pattern, where the user-action region or the environmental-condition region, or both, comprise a sequence of enabling sub-states.

A more complicated example is given in Figure 7. In this example, the enabling process is primarily ordered, but at one point in the sequence, there are enabling conditions whose orderings are not important. The transition from Environment Disabled to Environment Enabled can not take place until after the first user action has occurred. As well, the transition from User Enabled (2) to User Enabled (3) can not take place until the environmental conditions are all valid.

C. Example: Cruise Control

Consider a simplified version of the Cruise Control (CC) feature that, once activated, will maintain the speed of the vehicle at a driver-specified value. We present in Figure 8 a behaviour model of CC that uses the pattern. In this figure, events are modelled using upper-case and all other conditions are lower-case. This example is not based on any production or academic feature description. We do not claim that the modelled behaviour is complete or correct; we simply use it as a pedagogical example.

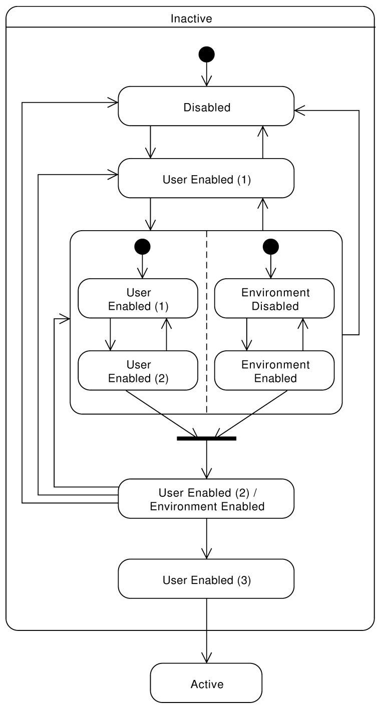

Fig. 7. The Hybrid Enabling sub-pattern.

We model CC using a variant of the Ordered Enabling sub-pattern. When the vehicle is powered on, the feature enters the Inactive state. The feature waits in sub-state Disabled until the user presses the CC On button. The feature then waits in sub-state User Enabled (1) until the *vehicleSpeed* is greater than or equal to 30 km/h. The feature then waits in sub-state Environment Enabled (1) until the user presses the CC Set button, setting the *cruiseSpeed* of the feature (i.e., the speed that CC will maintain) to the current *vehicleSpeed*. At this point, the feature transitions to User Enabled (2) and immediately activates.

We have modelled a simple Active behaviour for CC. The Active sub-machine initializes in the Maintaining Speed state. If the vehicle speed exceeds or becomes less than the *cruiseSpeed*, the feature will transition to Decelerating or Accelerating, respectively. If the user presses on the brake, turns off CC, or if the *vehicleSpeed* drops below 30 km/h, then the feature deactivates. Depending on the deactivating condition, the feature may transition back to a partially enabled state within the Inactive state.

We have modelled two reasons why CC may fail: (1) if the Electronic Brake Control (EBC) feature fails, and (2) if the sensors detecting the vehicle's speed fail. The machine transitions to Inactive when the reason for the failure no longer exists.

IV. A Generic Behavioural Interface for Features

The primary purpose of the proposed pattern is to ease the elicitation, documentation, and review of behavioural requirements of individual features. However, an important side effect

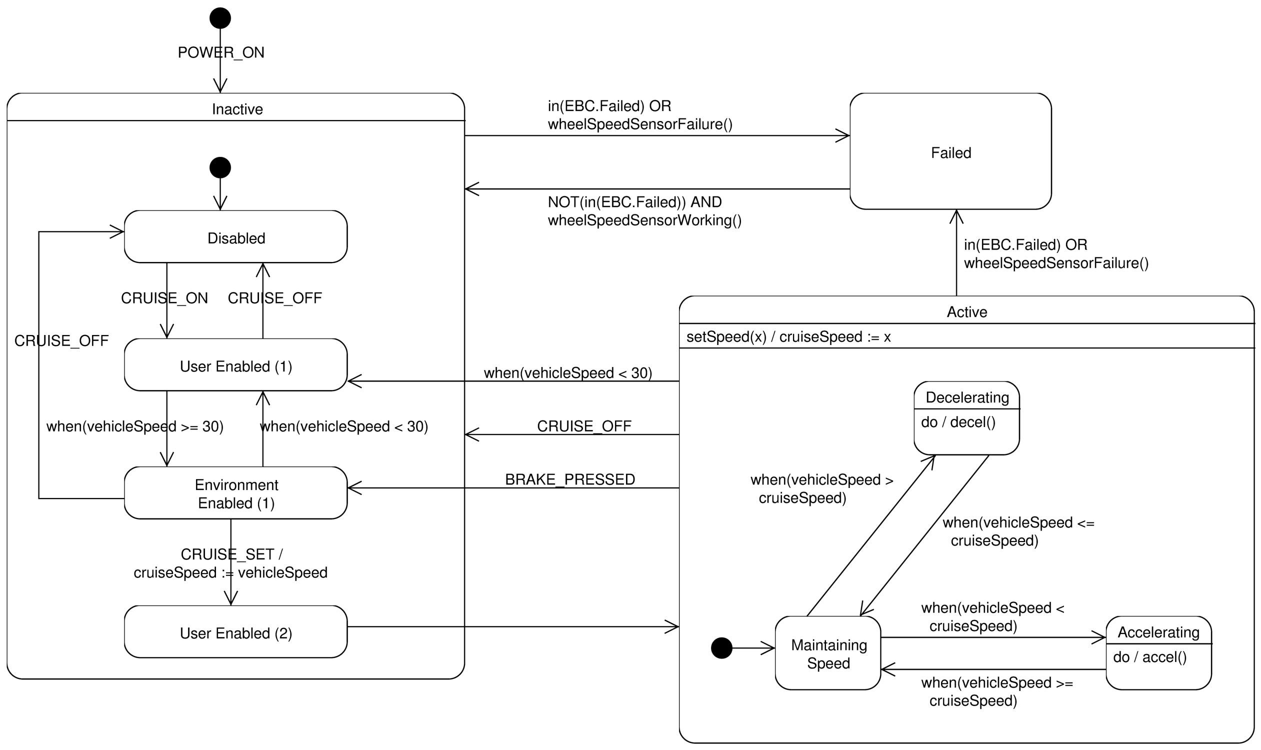
POWER_ON
Inactive
Disabled
CRUISE_ON
CRUISE_OFF
CRUISE_OFF
User Enabled (1)
when(vehicleSpeed >= 30)
when(vehicleSpeed < 30)
Environment Enabled (1)
CRUISE_SET / cruiseSpeed := vehicleSpeed
User Enabled (2)
in(EBC.Failed) OR wheelSpeedSensorFailure()
NOT(in(EBC.Failed)) AND wheelSpeedSensorWorking()
Failed
in(EBC.Failed) OR wheelSpeedSensorFailure()
Active
setSpeed(x) / cruiseSpeed := x
when(vehicleSpeed < 30)
CRUISE_OFF
BRAKE_PRESSED
Decelerating
do / decel()
when(vehicleSpeed > cruiseSpeed)
when(vehicleSpeed <= cruiseSpeed)
when(vehicleSpeed < cruiseSpeed)
Maintaining Speed
Accelerating
do / accel()
when(vehicleSpeed >= cruiseSpeed)

Fig. 8. An example application of the Ordered Enabling sub-pattern, used in a simplified model of Cruise Control.

is that the pattern can be used to define a *generic behavioural interface* to any adherent feature.

There has been little research on interfaces for features. In feature-oriented software development, work on feature modularity has focused on features as a criterion for system decomposition and assembly, such as in product-line development [15]; and on the cohesion of features [5], [16], including language or modelling support for coalescing all information related to a feature into a single module [21]. There is no information hiding among features, and one feature can directly refer to or override the details of other features. Alternatively, in the feature-interaction literature, feature modules are black boxes that have inputs and outputs, but otherwise share no information with each other [14], [19]. Such extreme information hiding facilitates parallel and third-party development of features, but makes it very difficult to specify intended interactions, such as when a new feature extends or overrides the behaviour of an existing feature, or when a feature ought to behave differently in the presence of other features.

We propose a compromise, in which features share a limited amount of information with each other by means of a feature interface. Our ideas are based on our initial analysis of the provided requirements documents: in most instances where one feature's requirements refer to another feature, the reference is an inquiry as to whether the other feature is active, has failed, or is even present in the system (since many features in a software product line are optional). Thus, we put forward our pattern's high-level modes as a **generic behavioural interface** for features, whereby a feature reveals whether its current execution state lies within Inactive, Active, or Failed. The interface exports information that can be viewed by observing features; it does not provide hooks that modifying features can use to directly affect the feature's behaviour. We hypothesize that such an interface reveals useful information about features, and would be sufficient in most cases where one feature needs to know the current state of another feature. Moreover, because the modes are common to all adherent features, the interface does not reveal any details of a feature's specification that is not already known to specifiers.

For example, consider again the behaviour model of our simple Cruise Control (CC) feature in Figure 8. CC fails if the Electronic Brake Control (EBC) feature fails, and we model this failure condition by labelling transitions to and from CC's Failed mode with *in(EBC.Failed)* guard conditions that monitor whether EBC is in its Failed mode.

Not all feature cross-references are references to interfaces. If two features are tightly coupled, they may refer to one another's internal details. For example, one feature's behaviour may depend on a related feature's current detailed state (e.g., there are limited autonomous driving features that depend on whether the driver is attentive). Alternatively, a new feature might override some detailed behaviour of an existing feature. We would expect such references to another feature's detailed behaviour to be relatively rare, and limited to tightly coupled features that are part of the same sub-system, developed by the same team, and specified in the same requirements document. We would deem our generic feature interface to be useful if references to features' interfaces were the norm.

V. Case Study

We performed a case study in which we analyzed the requirements of 19 automotive features, described in 5 production-grade requirements documents, and we created behaviour models using our mode-based pattern. Five of the features were modelled over the course of defining the pattern, and the other fourteen were modelled after the pattern had stabilized. The purpose of the case study was to assess:

- How well the pattern could be applied to features
- How well the pattern's modes could serve as a public interface for features

In all cases, we created behaviour models from scratch rather than refactor existing state-machines, either because the requirements documents included no state-machine models or because the machine was missing too many details.

A. Utility of the Pattern

The results of the case study are promising. Of the 14 features that were modelled after the pattern was determined, 11 conformed perfectly to the pattern. The other three features deviated from the pattern by starting in the Active state rather than in the Inactive state. We could have forced the features to start in the Inactive state and immediately transition to the Active state. However, we believe that the models in which these features start in the Active state are clearer, and that the pattern should allow some flexibility in a feature's initial state. Of the 19 features, 3 used the Ordered Enabling sub-pattern, 15 used the Unordered Enabling sub-pattern, and 1 used the Hybrid Enabling sub-pattern.

Examples: We highlight two of the case-study features. To avoid revealing proprietary information, we have abstracted away many details of each feature: the internal details of the Active state are omitted, and transition labels are replaced by the numbers of conditions on the transitions. In the models, events are represented in upper-case, the number of conditions are in bold, and all other conditions are lower-case.

The feature in the case study with the most complex enabling process is the Traction Control System (TCS). The TCS helps to limit tire slippage during acceleration on slippery surfaces by applying the brake and changing the amount of throttle applied to each wheel. The TCS is one of the features that activates immediately when the vehicle is powered on; it can be deactivated by manually disabling the feature or by environmental conditions.

Our model of the TCS employs the Hybrid Enabling sub-pattern (see Figure 9). Normally, re-enabling the TCS requires both a user action and an environmental condition to hold. However, the TCS can also activate in emergency situations without the user manually enabling the feature. This is modelled by having a transition directly from the unordered stage of the enabling process to state Environment Enabled (2) that is labelled with the emergency-override conditions. The failure conditions for the TCS are not specified in the requirements documents that we were provided, so we have not attempted to label the transitions to and from the Failed state.

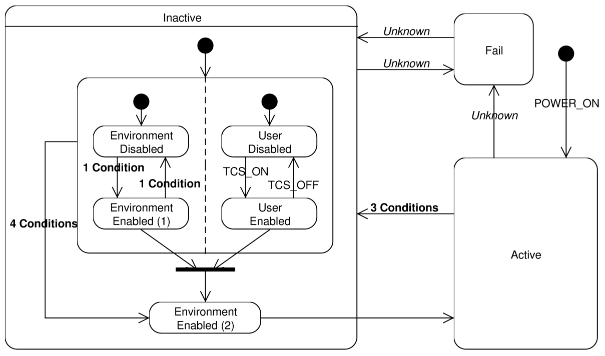

Fig. 9. The Traction Control System feature.

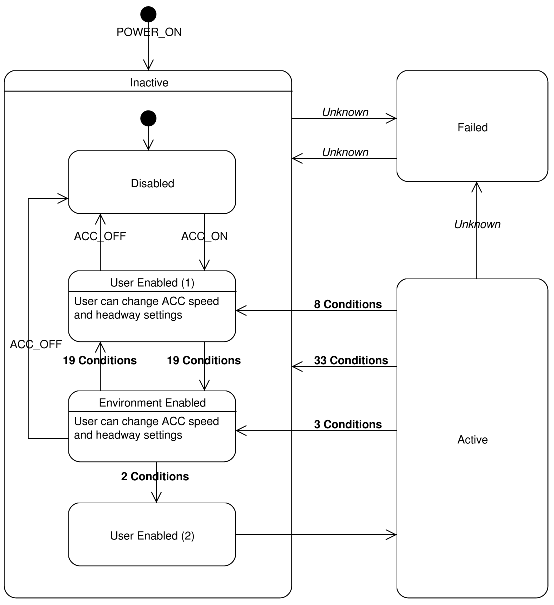

Fig. 10. Adaptive Cruise Control feature.

The feature in the case study with the largest number of enabling conditions is Adaptive Cruise Control (ACC). Once activated, the ACC feature will maintain the vehicle's speed at a driver-set value and will maintain a safe distance from the preceding vehicle. The ACC feature can be deactivated by either environmental conditions or user actions. Depending on the deactivating condition, the feature may transition back to a partially enabled state within the Inactive state.

Our model of ACC employs a variant of the Ordered Enabling sub-pattern (see Figure 10). Enabling the ACC feature is a multi-stage process: the user turns on the ACC feature, after which several environmental conditions are checked, and finally the user must perform one of two possible actions to complete the enabling process. There are three transitions from state Active back to state Inactive because, depending on the deactivation conditions, ACC will either deactivate completely and the enabling process will begin anew, or ACC will transition to a partially enabled state. The ACC failure conditions are not specified in the requirements documents that we were provided, so we have not attempted to label the transitions to and from the Failed state.

B. Examining Modularity

To check if the pattern would be viable as a public interface, we specifically examined: (1) how many features reference other features, (2) how often the references are to elements of the public interface, and (3) whether references are between tightly coupled features.

Each of the 5 features that were modelled while the pattern was being designed has between 1 to 14 references to other features. These features are primarily interested in whether other features are Inactive, Active or have Failed. However, three of the features have a small number of references to other features' detailed states (discussed in more detail below).

Of the 14 features that we modelled after the pattern stabilized, only 7 of the features have references to other features and all references are to high-level states in the other features' public interface. The seven features each have one to six references to other features. We hypothesize that the reason the latter features have fewer inter-feature references is that they are low-level features, whereas the features that we modelled as the pattern was being designed are complex, high-level features that build on the behaviour of lower-level features. We need more data to confirm this hypothesis.

We mentioned that there are three features that reference information not made available in other features' interfaces. In total, there are seven such references: four that read information and three that override some behaviour. Of the references that read information, three are to features in the same sub-system. The remaining reference is to a feature outside of its sub-system, to obtain information that is required to perform an algorithmic calculation. The references that override behaviour are all to features in the same sub-system and occur in cases where the feature is also reading information from the other feature.

In total, there are 57 inter-feature references. Of these, 50 are references to interface data, and 6 are references to internal details of tightly coupled features. Only one reference violates our convention of how features ought to behave and accesses private information from a feature outside of the sub-system.

VI. Discussion

A. Expected Benefits

Based on the known benefits of using patterns, interfaces, and information hiding, we expect that the use of our feature pattern and generic feature interface would improve the productivity of specifiers and reviewers of requirements models. We discuss these expected benefits below, though they must be confirmed empirically through user studies.

Improved consistency of models: In the original requirements documents, there is significant variability among the state-machines included with features (e.g., notation, structure and vocabulary all varied). In contrast, features that are modelled with the pattern tend to vary only in three ways: the sub-pattern used, the conditions on transitions, and the internal details of Active. This benefit is particularly useful for readers of requirements documents. A common structure and style for modelling features would prepare readers to review multiple feature requirements with increasing efficiency (this is particularly important when working on feature-rich systems). For requirements specifiers, the pattern provides a consistent state-machine framework that should simplify the requirements elicitation and formalization process.

Separation of concerns: The pattern clusters the three behavioural modes (Inactive, Active, and Failed) in a way that is consistent for all features. The clustering can be applied to the document description as well as the state-machine model. We hypothesize that readers and reviewers would benefit from such a separation in that they could more easily locate within a requirements document mode-specific details about a feature's behaviour (e.g., how a feature becomes enabled versus its active monitoring and controlling of the environment). Likewise, specifiers can separate the tasks of defining the different modes of a feature's behaviour.

Standardized vocabulary for feature behaviour: Standardized vocabulary is one of the widely regarded benefits of design patterns [10]. Ambiguities can arise when vocabulary is used inconsistently among multiple documents. The pattern addresses mode-based ambiguities by standardizing the names of features' modes and by basing a feature's public interface on these modes, so that a feature's references to another feature's modes also use standard vocabulary. We expect that readers would benefit from the improved consistency of vocabulary. Model specifiers would benefit from the generic feature interface because they know the names of other features' modes (and can reference those modes in their models) without having to look up the names.

Feature modularity: Interfaces to features have a number of expected benefits [23]: (1) They provide an immediate benefit to requirements engineers by reducing their cognitive load: an engineer need study only the interfaces of existing features to understand how a new feature fits into the system. (2) Once their interfaces are defined, new features can be modelled in parallel, by engineers working independently. (3) If designed with sufficient care and foresight, a feature's interface exports only information that is unlikely to change as the feature evolves. Such interfaces provide a *deferred* benefit of easing future feature evolution, in that changes to a feature's internal details are less likely to affect the specifications of related features. The second and third benefits listed above seem particularly realizable because our feature interface is predefined and is constant.

Compositional verification: It is also possible that our notion of feature interfaces could be used to facilitate compositional verification of features. In compositional verification, each feature is verified separately within its respective context (which includes the other features in the system). Compositional verification is effective if the effort to verify all of the features individually is smaller than the effort to verify a model of the whole system. We are currently investigating whether features' interfaces can be used to construct abstract feature contexts that significantly reduce the problem of verifying individual features.

B. Generality

Although the case study involved only automotive features, we expect that this work will generalize to other feature-oriented requirements in which the behaviour of each feature is modelled as a separate state machine. The pattern should be especially applicable to safety-critical systems, in which features have complex enabling/disabling conditions and failure/recovery conditions, in addition to their core functionality.

C. Threats to Validity

The requirements documents to which we had access were for older versions of features. According to the requirements engineers, current requirements of the same features include additional conditions for enabling/disabling/failing. Because of this, it is possible that our pattern may not apply to the current requirements, but we do not have any reason to think this is the case. The updated requirements documents use state names like Inactive, Active and Failed more consistently. However, to our knowledge, the requirements models still do not employ patterns, make use of hierarchy, or have a consistent terminology for the enabling conditions of a feature.

We have modelled a relatively small number of features, which may not be representative of all features. In particular, many of the documents we have examined omit details regarding failure and recovery conditions, thus the evaluation of that part of the pattern is weaker than that of other parts of the pattern.

There is potential for experimenter bias because the primary author performed both the pattern design and the case-study analysis. Experimenter bias is unlikely to affect the results of the modelling exercises, but our claims of improved readability and writeability are vulnerable and need to be validated with user studies.

Lastly, although we have presented our work to the engineers who provided the original requirements documents, they have not verified that our models are correct. That said, the engineers have seen enough value in our work to look at incorporating our pattern into their requirements-management system.

VII. Related Work

Patterns for Easing Requirements Elicitation: Early work on requirements patterns includes domain abstractions or clichés [24] and domain models [3], [26], which record general domain knowledge. The *Requirements Apprentice* [24] employs a library of clichés that can be reused in the specification of multiple systems, where a *cliché* is a set of roles – such as a repository, its contents, and its users – and constraints between roles. Clichés are normally documented using semi-structured text rather than graphical models. Sutcliffe and Maiden [26] extended these ideas to a catalogue of generic reusable *domain models* that encode structural and behavioural requirements of domain entities. More generally, Jackson introduced *problem frames* [13] as a way of classifying problems and sub-problems according to desired changes to or constraints on environment phenomena (e.g., a *transformation* problem, or a *workpieces* problem). A problem frame depicts a context diagram that relates the proposed machine, distinct domains of the environment (á la domain models [26]), and desired requirements among domains (e.g., a transformation requirement relating an input domain and an output domain). Clichés, domain models, and problem frames help the engineer to elicit an accurate and complete set of requirements; and the use of domain terms improves the consistency of vocabulary in requirements documentation. Our work on feature patterns is complementary in that it aids in the structuring and documentation of behavioural requirements after they have been elicited and decomposed into feature modules.

The Software Cost Reduction (SCR) requirements model [12] decomposes a system's behaviour into mode classes and modes, and our use of the term *mode* comes from their work. A mode class in an SCR model is comparable to a feature or a sub-system of tightly coupled features in our work. However, SCR does not propose any reusable pattern for decomposing behaviour into modes, does not employ hierarchy for organizing a mode class's modes, and does not have any concept like an interface for mode classes.

Domain-Specific Patterns: Domain experts have collected and codified patterns for modelling specific types of requirements, such as business problems (e.g., accounts, transactions, plans, contracts) [8], embedded-system requirements (e.g., controllers, fault handling, watchdogs) [7], [18], information systems (e.g., information, presentation, access control) [28], security requirements [6], and nonfunctional requirements [9]. Most of these patterns focus on how to structure inter-related components, though Douglass [7] and Konrad et al. [18] include behaviour models of interactions among components. Our concept of a feature could be analogous to the behaviour of a single component in these other approaches, or could be analogous to some system-level functionality that involves multiple components. In either case, our pattern for modelling a feature's behaviour would be complementary to these domain-specific requirements patterns.

Feature Interfaces: Much has been written about modularity, interfaces, and information hiding [23]. We focus our discussion on feature modularity. Within the feature-oriented software-development community, the emphasis of feature modularity is on cohesion and locality of feature information [5], [16], [21]. There is no concept of feature interfaces or support for information hiding, so features refer to and directly override the details of other features. At the other extreme, the feature-interaction community often models features as distinct modules that have no knowledge of each other [14], [19]. There is no need for feature interfaces because each feature's information is completely hidden. The downside of total information hiding is that it is hard to specify and manage intended feature interactions (e.g., enhanced or overriding behaviour). The aspect-oriented community has proposed aspect-aware interfaces that advertise a module's pointcuts as well as its public data attributes and methods [1], [17], thereby providing limited means by which other modules can use, extend, or override the module's services. However, such interfaces are not stable, as a module's set of public pointcuts tends to change as new aspects are introduced or evolve [17]. In contrast, we propose a feature interface that provides limited information about a feature's current execution state (information that is useful in the modelling of other features), such that the interface is stable and generic for all features. There is also work on deriving feature interfaces to support compositional verification [20], however such interfaces do not aid in the specification or evolution of feature models.

VIII. Conclusion

We have presented a pattern for structuring the behaviour model of a feature, and have introduced a generic behavioural interface for features based on the common modes of our pattern. The results of our case study show that automotive features conform well to the pattern, and that most inter-feature references are to information that would be available in a feature's interface. Although the pattern was extracted from and evaluated using automotive requirements, there is nothing inherently automotive-specific about the pattern and it should be generally applicable to feature-oriented requirements.

We continue to evaluate our pattern on more complex automotive features, and we plan to use it to model feature-oriented requirements in other domains (e.g., the Pacemaker challenge[§]). We also plan to conduct a user study to determine whether use of the pattern and proposed feature interface improves the productivity of requirements reviewers and specifiers, as expected. Finally, we plan to provide tool support for creating and editing feature models that adhere to the pattern.

References

[1] J. Aldrich, "Open modules: modular reasoning about advice," in *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, 2005, pp. 144–168.

[2] S. Apel, F. Janda, S. Trujillo, and C. Kästner, "Model superimposition in software product lines," in *Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations (ICMT'09)*, 2009, pp. 4–19.

[3] G. Arango and P. Freeman, "Modeling knowledge for software development," in *Proceedings of the 3rd International Workshop on Software Specification and Design (WSSD'85)*, 1985, pp. 63–66.

[4] S. Arora, P. Sampath, and S. Ramesh, "Resolving uncertainty in automotive feature interactions," in *Proceedings of IEEE International Requirements Engineering Conference (RE)*, 2012, pp. 21–30.

[5] D. S. Batory, J. N. Sarvela, and A. Rauschmayer, "Scaling step-wise refinement," *IEEE Transactions on Software Engineering*, vol. 30, no. 6, pp. 355–371, 2004.

[6] B. H. C. Cheng, S. Konrad, L. A. Campbell, and R. Wassermann, "Using security patterns to model and analyze security," in *Proceedings of the Workshop on Requirements for High Assurance Systems, at the IEEE International Requirements Engineering Conference*, 2003, pp. 13–22.

[7] B. P. Douglass, *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns*. Addison-Wesley, 1999.

[8] M. Fowler, *Analysis Patterns: Reusable Object Models*. Addison-Wesley, 1996.

[9] X. Franch, C. Palomares, C. Quer, S. Renault, and F. D. Lazzer, "A metamodel for software requirement patterns," in *Proceedings of Requirements Engineering: Foundation for Software Quality (REFSQ'10)*. Springer, LNCS 6182, 2010, pp. 85–90.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] D. Harel, "Statecharts: A visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231–274, June 1987.

[12] C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw, "Automated consistency checking of requirements specifications," *ACM Transactions on Software Engineering and Methodology*, vol. 5, no. 3, pp. 231–261, July 1996.

[13] M. Jackson, *Problem Frames: Analyzing and Structuring Software Development Problems*. Addison-Wesley, 2000.

[14] M. Jackson and P. Zave, "Distributed feature composition: A virtual architecture for telecommunications services," *IEEE Transactions on Software Engineering*, vol. 24, no. 10, pp. 831–847, Oct. 1998.

[15] 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.

[16] C. Kästner, S. Apel, and K. Ostermann, "The road to feature modularity?" in *Proceedings of the 15th International Software Product Line Conference (SPLC), Volume 2*, 2011, pp. 5:1–5:8.

[17] G. Kiczales and M. Mezini, "Aspect-oriented programming and modular reasoning," in *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE'05)*, 2005, pp. 49–58.

[18] S. Konrad and B. H. C. Cheng, "Requirements patterns for embedded systems," in *Proceedings of the IEEE Joint International Conference on Requirements Engineering (RE'02)*, 2002, pp. 127–136.

[19] R. C. Laney, T. T. Tun, M. Jackson, and B. Nuseibeh, "Composing features by managing inconsistent requirements," in *International Conference on Feature Interactions (ICFI)*, 2007, pp. 129–144.

[20] H. C. Li, S. Krishnamurthi, and K. Fisler, "Interfaces for modular feature verification," in *Proceedings of the IEEE International Conference on Automated Software Engineering (ASE'02)*, 2002, pp. 195–204.

[21] R. E. Lopez-Herrejon, D. Batory, and W. Cook, "Evaluating support for features in advanced modularization technologies," in *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, 2005, pp. 169–194.

[22] Object Management Group, "UML Specification: Superstructure, version 2.2," 2009. http://www.omg.org/spec/UML/2.2/Superstructure/PDF

[23] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 15, no. 12, pp. 1053–1058, December 1972.

[24] H. Reubenstein and R. Waters, "The requirements apprentice: automated assistance for requirements acquisition," *IEEE Transactions on Software Engineering*, vol. 17, no. 3, pp. 226 –240, March 1991.

[25] P. Shaker, J. M. Atlee, and S. Wang, "A feature-oriented requirements modelling language," in *Proceedings of the International Requirements Engineering Conference (RE'12)*, 2012, pp. 151–160.

[26] A. Sutcliffe and N. Maiden, "The domain theory for requirements engineering," *IEEE Transactions on Software Engineering*, vol. 24, no. 3, pp. 174 –196, March 1998.

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

[28] S. Withall, *Software Requirement Patterns*. Microsoft Press, 2007.

[§]http://wiki.cas.mcmaster.ca/index.php/Pacemaker