DOI: https://doi.org/10.1109/RePa.2013.6602664

A Pattern for Structuring the Behavioural Requirements of Features of an Embedded System

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

***Abstract*—Feature-oriented software requirements specify features in a product line as separate modules. In this paper, we present the *Mode-Based Behaviour* pattern that provides advice on how to structure the behavioural requirements of an individual feature using state machines. The pattern not only defines the control flow of a feature, but also places constraints on the kinds of behaviour that a feature can perform while in certain operating modes. The pattern has been created by examining several production-grade automotive features and identifying commonalities in their high-level behaviours; however the pattern is not automotive specific.**

***Keywords*-requirements engineering; requirements pattern**

I. Introduction

Several industries are embracing the benefits provided by feature-oriented software development (FOSD) [1]. We are collaborating with a major automotive manufacturer that uses FOSD, and we have been given access to the requirements for several production-grade features. The pattern that we present in this paper has been created by examining those requirements and identifying common structures that can be used to model a feature's behaviour. Although we have used automotive requirements as guidance while creating the pattern, there is nothing automotive specific about it; the pattern seems generally applicable to domains in which features have reactive behaviour.

A feature is a "coherent and identifiable bundle of system functionality" [16] 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. A feature with reactive behaviour is a feature that monitors its environment and reacts to various inputs, has reactions that depend on execution history, and has long execution time. We are specifically interested in reactive features that are specified using a hierarchical state machine (e.g., statecharts [8], or UML State Machines [13]).

We propose a pattern that decomposes and structures the behaviour model of a feature (expressed as a state machine) by operating mode. The 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). The pattern also provides advice on how to model the enabling process and the active requirements of a feature.

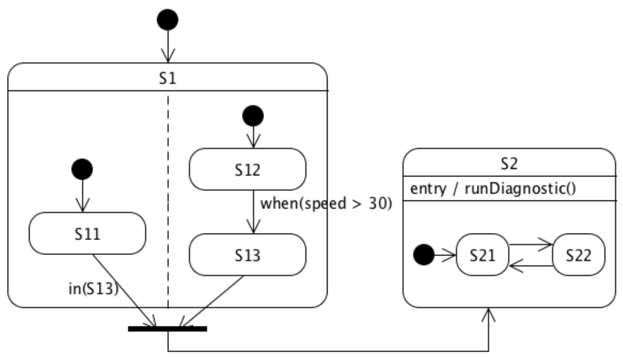

Fig. 1. State-machine notation.

An interesting aspect of this pattern is that it seems to be general enough to be applicable to most features in a product line while still defining a noticable structure for the state-machine model of the feature. It is likely that there will be a large number of features in a product line (in our experience, automotive product lines can have upwards of 1000 features). The pattern makes it progressively easier to review and understand the requirements of multiple features if they all follow the same pattern.

The pattern presented in this paper is an extension of one that we have previously presented [4]. Our previous paper focused on describing the main pattern, the feature's enabling process, and our notion of a generic interface for observing features. In this paper, we include an extension of the main pattern for modelling the Active behaviour of a feature. This is also our first attempt to structure the pattern description using a pattern template.

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 pattern that we have devised, Section IV discusses several related concepts, and we conclude in Section V.

II. Background

In this section, we present a brief overview of several aspects of our state-machine notation worth reviewing.

Consider the state machine exhibited in Figure 1. A state may contain sub-states; in this case, the former is called a *superstate* (e.g., S1, 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 (e.g., S1). If a transition's destination is a superstate, then the next state is the initial state of the superstate's sub-machine (e.g., S21) or the initial states of the superstate's regions (e.g., S11 and S12).

Transitions can be annotated with:

- An event, which is a user action or a change in the values of environmental variables.
- A guard condition, which is a predicate over environmental variables.
- A set of actions on environmental variables.

Each of those three annotations is optional. Transition annotation *when(c)* refers to the event of condition *c* becoming true (e.g., *when(Vehicle.speed* $>$ *30)*). Transition annotation [*in(S)*] is a condition that is satisfied when the system's execution is in state *S* (e.g., *in(S13)*); 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 simultaneously.

A state may be annotated with actions that are performed while the modelled feature's execution is in that particular state (e.g., S2). Actions can be undefined functions whose behaviour is not specified.

III. The Pattern

In this section, we present the pattern that we have defined. The pattern is presented using the pattern template provided by the Third International Workshop on Requirements Patterns [3]. We do not include the *Known Uses* and *Cataloging* sections because we are proposing this pattern and have not seen it used in any existing works. Thus, we are actually presenting a *proto-pattern*. However, we use the term *pattern* rather than *proto-pattern* throughout this paper to simplify the presentation. The *Solution* section of the pattern uses state machines to illustrate the pattern concepts. The example that we provide is based on a specific production-grade requirement from our automotive collaborator (details are removed or changed to avoid revealing proprietary information). We have added a *Resulting Context* section to describe the forces that result from applying the pattern because we feel that they are easier to describe after the solution is presented.

1) Name:

Mode-Based Behaviour Pattern

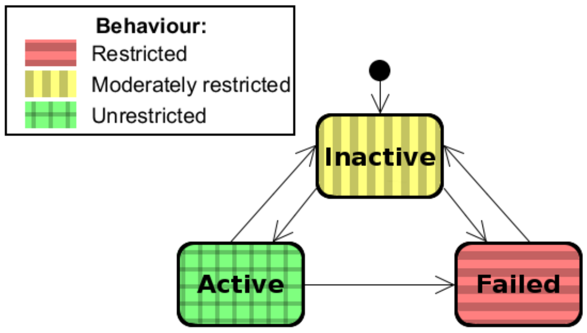

Fig. 2. The Mode-Based Behaviour pattern, where the modes are coloured with the degree to which the feature's interactions with its environment are restricted.

2) Context:

- RE Activity: Requirements Elicitation and Specification
- Pattern Type: Product
- Stakeholders: Requirements Engineers, Software Developers, Project Managers

---

3) Problem:

Practitioners have difficulty writing state-machine models and decomposing behaviour into states. Different stakeholders are likely to make different decisions, which can complicate the task of reading a model if the structure of the model doesn't match the reader's mental model of feature behaviour.

---

4) Forces:

- Creating complex state machines is difficult. There is a trade-off between the productivity of the requirements specifier and the percentage of a feature's requirements that are modelled. At our automotive collaborator, state-machine models do not go into detail because it is easier and faster to describe complex functionality in text.
- Writing requirements in text can lead to ambiguity, subjective interpretation, and inconsistencies. In contrast, state-machine models have a well-defined syntax and semantics.
- The pattern introduces some constructs that, if unused, introduce additional overhead.

---

5) Solution:

The Mode-Based Behaviour pattern consists of three high-level states that correspond to the three distinct operating modes: Active, Inactive, and Failed. Features may have very complex control flow and it is important to ensure that the operating modes are well represented. Figure 2 shows the high-level structure of the pattern.

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

- the feature monitors the environment (e.g., an Adaptive Headlights (AH) feature monitoring the ambient light level)
- the feature acts on the environment (e.g., AH automatically turning on the vehicle's headlights)
- the environment monitors the feature (e.g., other features in the system monitoring AH's current execution state)
- the environment acts on the feature (e.g., the vehicle operator changing the sensitivity of AH)

Each state permits different types of interaction.

The **Inactive** state comprises a sub-state machine that models the behaviour of the feature as it becomes enabled. For example, most features have a collection of conditions that must all be true prior to activation. 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. In the Inactive state, the feature can monitor the environment and the environment can act on the feature but the feature does not act on the environment.

The **Active** state comprises a sub-state machine that models all of the ways in which the feature actively affects the behaviour of the system. The exact conditions that cause the feature to deactivate (i.e., that trigger the transition from Active to Inactive) are feature specific, and thus are not part of the pattern. Transitions to Inactive can be triggered by any sub-state within Active. The destination of a transition from Active is normally the boundary of the Inactive superstate, meaning that the enabling process starts from the initial state(s) of Inactive's sub-machine(s). In the Active state, all four types of interactions between the feature and its environment are allowed.

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 Inactive state and the enabling process begins again. The pattern does not decompose the internal structure of the Failed state because the features we have examined typically do nothing more than monitor the environment while failed. The industrial feature-requirements documents that we have examined are light on failure requirements because many features (and all safety-critical features) have separate safety-requirements documents that describe how the features behave in the presence of failures. We did not have access to any safety-requirements documents. In the Failed state, the feature can only monitor the environment – to determine if any of the state's outgoing transitions are enabled.

5.1) Extension to Inactive State

The enabling process for a feature can range from simple (e.g., only requiring the user to press a button) to very complex (e.g., requiring the user to perform multiple actions and requiring several environmental properties to hold). We define two extensions of the main pattern that apply to the Inactive state's sub-machine; they are organized by the degree to which a feature's enabling process is ordered:

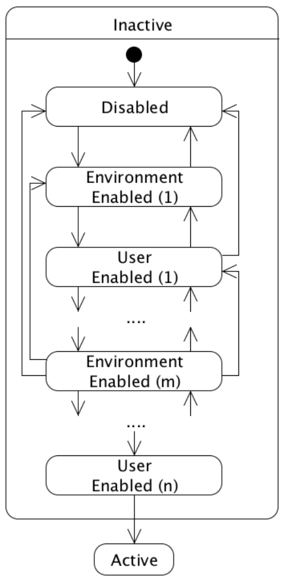

Fig. 3. The Ordered Enabling extension of the Mode-Based Behaviour pattern.

1) There is a sequential ordering on enabling conditions.
2) There are no ordering constraints on enabling conditions.

There are two types of enabling conditions to consider: 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). The description of a feature's enabling process 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.

Ordered Enabling: The Ordered Enabling extension applies when a feature becomes enabled in stages. The Inactive submachine is a sequence of user actions and environmental conditions that must be satisfied in the specified order. 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 state-machine fragment in Figure 3 shows how the Ordered Enabling extension models 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.

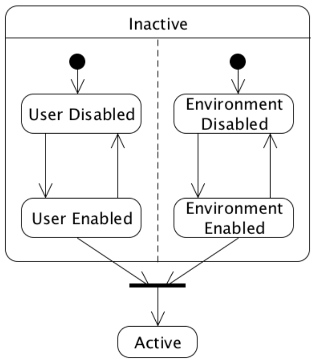

Fig. 4. The Unordered Enabling extension of the Mode-Based Behaviour pattern.

Unordered Enabling: It sometimes 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 extension (shown in Figure 4) applies in these situations. The concurrent regions separate user actions (on the left) from the environmental conditions (on the right). A transition within these regions can be triggered by a combination of user actions or a combination of environmental conditions. When both 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.

The Unordered Enabling extension can be generalized to multiple concurrent regions when the enabling process comprises multiple sequences of user actions or environmental conditions, where the sequences are unordered with respect to one another.

The Unordered Enabling extension also applies when a feature's enabling process comprises 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.

5.2) Extension to Active State

There are several different kinds of behaviour that a feature performs while active. Sometimes a feature is active but only monitoring the environment; other times, the feature is affecting the system's behaviour.

The pattern extension for the Active state provides advice on how to structure the active behaviour of a feature. Figure 5 shows the Active extension and how it relates to the Inactive and Failed states.

The Active extension is composed of four states:

1) **Monitoring:** When in the monitoring state, the feature is only monitoring the behaviour of the vehicle; it is not actively controlling the vehicle's behaviour.

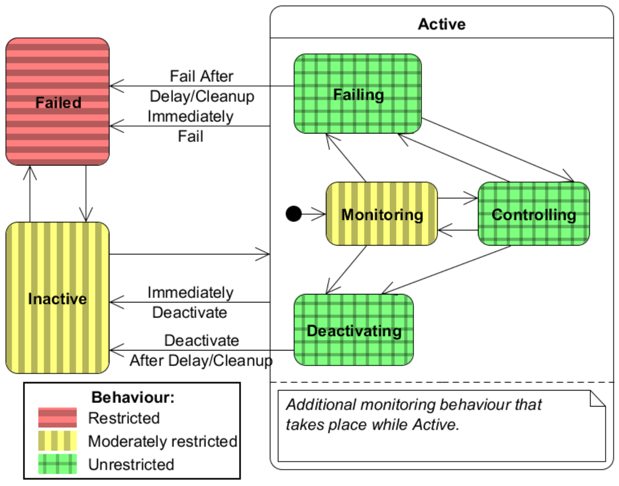

Fig. 5. The framework of the Active extension showing the kinds of behaviour that can occur in each state.

2) **Controlling:** This is the state in which a feature actively affects the vehicle's behaviour. A feature's Controlling state typically contains one or more sub-machines that model the controlling behaviour of the feature.
3) **Deactivating:** In this state, a feature is in the process of deactivating, but needs to perform some actions before it becomes Inactive (e.g., the feature automates some user's task and notifies the user when it is deactivating – remaining operational for some time to allow the user to resume responsibility for that task).
4) **Failing:** The purpose of this state is similar to the purpose of the Deactivating state – when a feature is failing, it warns the user and attempts to remain temporarily operational to allow the user to resume responsibility over the feature's task.

An activating feature normally initializes in the Monitoring sub-state and transitions between the Monitoring and Controlling sub-states, depending on whether the feature is currently performing some actions that control some aspect of the vehicle. An Active feature will transition to Deactivating as an intermediate sub-state towards transitioning to Inactive, or to Failing as an intermediate sub-state towards transitioning to Failed. The events, conditions, or actions with which these transitions are labelled are feature specific and thus are not part of the pattern. If a feature is able to provide any degraded service in the event of a failure, that behaviour is modelled as part of the Failing sub-state because the feature is still partially active. This allows the Failed state to focus on the feature's behaviour when it is completely failed.

Some features will transition from the Monitoring or Controlling sub-states directly to Inactive or Failed, either because no intermediate state is necessary, or because no intermediate state is possible. Such transitions should be from the border of the Active superstate to the Inactive or Failed states.

The colours of the Active sub-states in Figure 5 depict the ways in which the feature can interact with its environment: In moderately restricted (yellow) states, the feature can monitor the environment and the environment can act on the feature (e.g., the user can modify feature settings). In unrestricted (green) states, all types of interactions are allowed.

In addition to the four states within the pattern, there may be additional concurrent regions within Active. Those additional regions are useful when the feature continuously monitors multiple environmental phenomena when in the Active state.

---

6) Variants:

There are several possible variants of the pattern:

1) The initial state of the feature may be Active.
2) Some features may deactivate but still be partially enabled depending on the user actions or environmental conditions that prompt the deactivation. To handle that case, transitions from Active to Inactive can cross the Inactive superstate's border and initialize the Inactive sub-machine at some point in the enabling process.
3) The initial state of the Active extension may be Monitoring or Controlling.
4) Features in the Failing state can transition back to Monitoring or Controlling if the error is recovered before the transition to Failed takes place.
5) Deactivating and Failing sub-states can be omitted from the Active sub-machine if the feature has no such sub-behaviour.
6) A feature can have two or more instances of the Active extension, each in a separate concurrent region with its own Monitoring and Controlling states. However, if a feature has multiple Active extensions that are truly orthogonal, it may be a sign that the feature should be split into separate features, such that each Active state contains only one instance of the Active extension.

---

7) Resulting Context:

- Providing a pattern for creating a feature's state-machine models simplifies the requirements engineers task when modelling a feature's behaviour.
- The pattern helps resolve ambiguity because it standardizes the structure and vocabulary of any textual requirements described alongside the state machine. The state-machine model also serves as a central figure that the textual requirements can all reference.
- Additional states that are not used can be omitted, but this could lead to confusion when others read the requirements document as they do not know if the states have been omitted intentionally or unintentionally.

---

8) Example:

Consider a simplified version of a Lane Centring Control (LCC) feature which, once activated, will maintain the vehicle's position in the centre of its current lane without any driver input. A state machine for LCC is given in Figure 6. To avoid revealing proprietary information, we abstracted away several details of the state machine and replaced several transition labels with the number of conditions that are checked by the transition.

The LCC feature applies the Mode-Based Behaviour pattern extended with the Unordered Enabling and Active extensions. The enabling process inside Inactive checks that the user presses the On button for LCC (denoted by the LCC_ACTIVATE event) and confirms that two environmental conditions hold. Once activated, LCC initializes in the Monitoring sub-state of Active and monitors eight conditions to determine if the feature should transition to Controlling in order to influence the vehicle's steering. While in the Controlling state, LCC maintains the vehicle's position in the centre of the lane. There is an additional concurrent region in the LCC feature that keeps track of whether the vehicle is currently centred in the lane. The sub-states of this region (*Centred* and *Not Centred*) are used by the Controlling sub-machine to determine whether the feature should correct the vehicle's position.

When LCC is deactivating or failing, it goes through the Deactivating and Failing states, respectively. While in the Deactivating or Failing state, LCC slowly yields control of the vehicle to the driver (handled by the *YieldControl()* function) and only transitions to Inactive or Failed when the driver has control of the vehicle.

IV. Related Work

Patterns for Easing Requirements Elicitation: Early work on requirements patterns includes domain abstractions or clichés [14] and domain models [2], [15], which record general domain knowledge. The *Requirements Apprentice* [14] utilizes 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. Sutcliffe and Maiden [15] extended these ideas to a collection of generic reusable *domain models* that encode structural and behavioural requirements of domain entities. More generally, Jackson introduced *problem frames* [10] 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 [15]), 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 [9] decomposes a system's behaviour into mode classes and

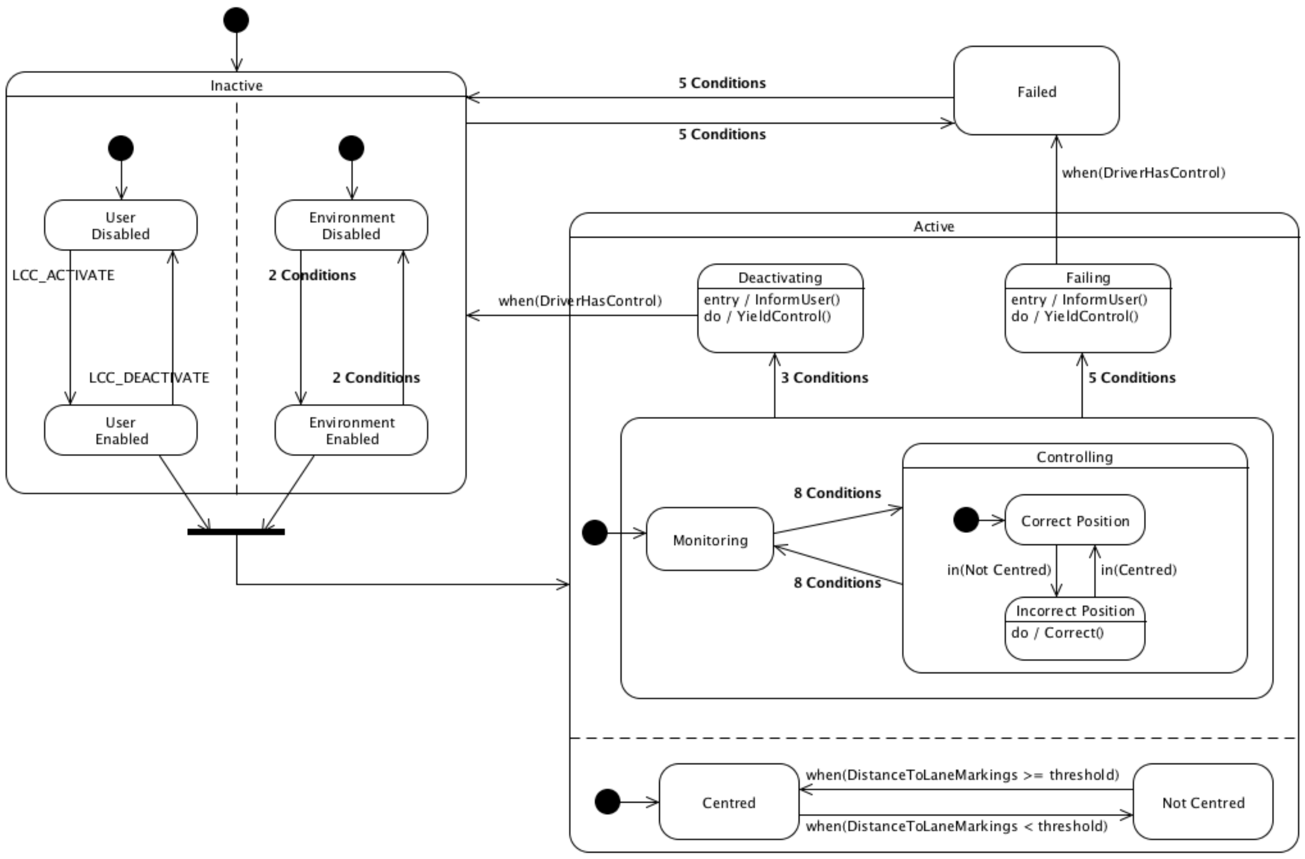

Fig. 6. The Lane Centring Control feature.

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) [6], embedded-system requirements (e.g., controllers, fault handling, watchdogs) [5], [11], information systems (e.g., information, presentation, access control) [17], seismology software requirements [12], and nonfunctional requirements [7]. Most of these patterns focus on how to structure inter-related components, though Douglass [5] and Konrad et al. [11] 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. Our pattern for modelling a feature's behaviour is complementary to these domain-specific requirements patterns.

V. Conclusion

The Mode-Based Behaviour pattern provides a generic template that can be applied to feature requirements. Although we have created the pattern using automotive features as guidelines, there is nothing that is automotive specific about the pattern. We expect that the pattern generalizes to any feature that has reactive behaviour, making this pattern ideal for feature-oriented embedded systems.

This paper presents a proto-pattern that we have not found used in practice but that we propose as an improvement over practice. We have evaluated the proto-pattern by applying it to 21 production-grade features provided by our automotive collaborator. We found that the pattern can be used to model every one of the features. We have performed a small user study with 12 participants and found that models constructed using the pattern tend to have improved readability, writability, and correctness over models constructed without any pattern. We expand on the benefits of the pattern in our previous paper [4].

We intend to perform a larger user study to gain more statistically significant results about the benefits and consequences of using the pattern.

References

[1] S. Apel and C. Kastner, "An overview of feature-oriented software development," *Journal of Object Technology*, vol. 8, no. 5, pp. 49–84, 2009.

[2] 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.

[3] L. Chung, B. Paech, L. Zhao, L. Liu, and S. Supakkul, "Repa requirements pattern template," 2012. www.utdallas.edu/ supakkul/repa13/RePa Requirements Pattern Template v1.0.1.pdf

[4] D. Dietrich and J. M. Atlee, "A mode-based pattern for feature requirements, and a generic feature interface," *To appear in Requirements Engineering 2013*, 2013.

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

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

[7] 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.

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

[9] 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.

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

[11] 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.

[12] Y. Li, C. Pelties, M. Kaser, and N. Nararan, "Requirements patterns for seismology software applications," in *Proceedings of the IEEE International Workshop on Requirements Patterns (RePa)*, 2012, pp. 12–16.

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

[14] 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.

[15] 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.

[16] 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.

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