Weaving Behavior into Feature Models for Embedded System Families

T. J. Brown, R.Gawley, R. Bashroush, I. Spence, P. Kilpatrick, C.Gillan
School of Electronics, Electrical Engineering and Computer Science,
The Queen's University of Belfast.
{tj.brown, r.gawley, r.bashroush, i.spence, p.kilpatrick}@qub.ac.uk
C.Gillan@ecit.qub.ac.uk

Abstract

Product Line software Engineering depends on capturing the commonality and variability within a family of products, typically using feature modeling, and using this information to evolve a generic reference architecture for the family. For embedded systems, possible variability in hardware and operating system platforms is an added complication. The design process can be facilitated by first exploring the behavior associated with features. In this paper we outline a bi-directional feature modeling scheme that supports the capture of commonality and variability in the platform environment as well as within the required software. Additionally, 'behavior' associated with features can be included in the overall model. This is achieved by integrating the UCM path notation in a way that exploits UCM's static and dynamic stubs to capture behavioral variability and link it to the feature model structure. The resulting model is a richer source of information to support the architecture development process.

1. Introduction

Over recent years Software Product-line Engineering methods [1] have emerged as a major strategy for maximizing reuse when a family of related software systems is to be fielded. The key idea is to exploit the commonality within the family of products, by designing the family as a whole, rather than developing products on a one-at-a-time basis. A difficulty of course is that significant variability from product to product is generally also present, and must be accommodated. An initial phase involving commonality-variability analysis is generally required, and in current practice feature modeling [2] has emerged as a widely used technique at this stage of the process. Additionally, the design of a generic reference architecture for the family as a whole is widely recognized as a key activity within the process.

Opportunities for applying product-line methods are often encountered in the arena of embedded systems, where the family of systems comprises both hardware and software. Within such system families, commonality and variability may arise within both the software and hardware aspects of products. Moreover, within such a family it is possible to find some aspects of a product's functionality being implemented in software within some family members, but in hardware within others. In the construction of feature models for such families, it can be useful to model both hardware and software components, and capture the interrelationships between the features contributed by both. We have evolved a scheme of feature modeling targeted at such system families which provides this capability. It is significantly inspired by earlier approaches to feature modeling, including FODA [2] and particularly the FORM [3, 4] feature modeling process, but it allows bi-directional modeling to capture the features and feature variability within the operating platform as well as within the software.

Feature Modeling is primarily a means of capturing requirements and exposing variability within the product line. It has been argued [5] that a basic feature model does not completely characterize the variability within a product line. Pohl et al. [6] have introduced the notion of an orthogonal variability diagram as a way of resolving the ambiguity they identify in conventional feature models.

While unambiguous documentation of variability in requirements is important, it is highly desirable that the information assembled within the model should support the subsequent phases of the domain engineering process. A key downstream activity in domain engineering is the development of a generic reference architecture for the product line. However, there is a substantial transition involved in going from a feature model to an architecture. Moreover, while feature models provide a significant input to the process, our

10th International Software Product Line Conference (SPLC'06)
0-7695-2599-7/06 $20.00 © 2006 IEEE

experience is that a feature model of itself contains insufficient information to support the derivation of a generic architecture for a product-line. The difficulty is that some degree of knowledge of the behavioral aspects of the family is also necessary. This is not provided by feature modeling in its conventional form. The value of behavioral information has also been recognized by others. For example, Mei et al. [7] in their FODM modeling framework include the concept of behavioral characteristics for functional features. In the PLUSS modeling approach [8], broadly similar motives have prompted Eriksson et al. to combine features with Use Case models and Use Case realizations.

In an earlier paper [9] we introduced the concept of bi-directional feature models and described an outline methodology for evolving software architectures from feature models. This approach made significant use of additional information derived from scenarios designed to exercise features in a systematic way. Using scenarios in this way serves to expose aspects of the behavior associated with individual features. (Some features may, of course, be intrinsically non-functional.) This in turn makes it easier to recognize, for example, features that could be implemented within a single component, as opposed to features that are inherently cross-cutting and require an implementation approach involving several components.

Experimentation in this area led us to recognize the potential benefit of having a means of modeling not just features, but feature behavior: in other words a mechanism for integrating behavior and behavioral variability into feature models. For this to work successfully, it is essential that the behavioral information be captured in a highly abstract manner, making no assumptions about any pre-existing software structure. Feature models normally have optional and alternative features that serve to capture the feature variability within the model. Clearly any matching behavioral notation has to support the synchronized capture of optional or alternative behavior. In essence, the feature model structure and the related variability in behavior need to be woven together within an integrated modeling framework.

In the remainder of this paper we describe our feature and behavioral modeling schema. In Section 2 the basic scheme of bi-directional feature modeling is described. Section 3 discusses the selection of a suitable notation for capturing feature behavior and explains the choice of the UCM path notation. The basic features of this notation are briefly described, although fuller details are available in other publications [10, 11, 12, 13], and online via [14]. In section 4 we then describe how we compose these notations. Section 5 outlines a general methodology for identifying behavior and its relationship to features, and section 6 presents a short case study highlighting some potential benefits of the combined notation. An important longer term objective is to evolve an architecture development methodology for product-line architectures using this kind of model. This is the subject of active research, but lies outside the scope of this paper.

2. Bi-Directional Feature Modeling

The feature modeling scheme proposed herein, and referred to as *Rationalised Feature Modeling*, is intended to provide a framework which can be used for modeling ***system*** families, where individual features and functionality may be provided wholly in hardware, wholly in software, or partially in hardware and partially in software, and where different products within the family may have different policies of distributing functionality between hardware and software. In such a situation it is important that any feature model should capture the relationships and interdependencies between the hardware based features and features or functionality to be provided in software.

Rationalised Feature Modeling is partially inspired by the FORM feature modeling process, and incorporates useful ideas from other feature modeling schemas. It retains FORM's idea of layered feature modeling but separates the operating environment layer from the other three found within FORM. The operating environment layer is replaced by an optional platform layer intended to contain O/S and hardware related features. The framework allows modeling optionally from two directions: a top-down feature tree models the product family's software, while an inverted feature tree models the platform layer. If the platform layer is present then there can be relationships across the hardware/software boundary, including mutual dependencies and hardware/software feature alternatives. These latter relationships indicate product functionality which may be provided in software on some products within the family, and as hardware on others.

2.1. Supported Feature Types

In common with the basic FODA framework, and most subsequent notations, Rationalised Feature Modeling allows features to be mandatory, optional or alternative. A mandatory feature will be supported by every product instance that supports it parent. Optional features are features that may be present or absent from any product within the family. Alternative feature sets are sets of features from which only one is selected for inclusion in any given product. They are thus mutually exclusive: if one is supported the others cannot be. In addition, rationalised feature modeling allows the use of

OR features [15]. An OR feature set is a set of features from which one or more may be selected into any product within the family. At least one must be selected but there is no exclusivity relationship, and in fact a product may contain all features within any OR feature set.

2.2. Hierarchical Feature Relationships

Two forms of hierarchical relationship between features are supported. These are the relationships *consists_of* and *provided_by*, and are commonly supported by other feature modeling schemas. The *consists_of* relationship may be used to indicate that a feature at a certain level consists of one or more lower-level sub-features. The *provided_by* relationship indicates that a feature at one level is provided by other lower level features. These two relationships serve to capture the hierarchical structure of the feature tree.

2.3. Feature Dependencies and Constraints

There are two forms of feature dependency supported. These are the *requires* and *excludes* dependencies which are also found in the FODA approach. *Requires* dependencies arise when the inclusion of one feature within a product is only appropriate so long as another required feature is included as well. Although both features may be optional or alternative, they must be included or excluded together. The opposite situation arises when the inclusion of an optional or alternative feature makes it necessary to exclude some other feature. This constitutes an *excludes* dependency. In the graphical notation, both forms of dependency are illustrated by dashed arcs. In the case of a *requires* dependency the arc carries a single terminating arrow pointing to the required feature. In the case of an *excludes* relationship the arc carries an arrow at both ends.

2.4. Bi-Directional Feature Modeling

Perhaps the most radical aspect of the core feature modeling scheme is its support for bi-directional models. In this approach a conventional top-down feature tree models features of the family that are software based, or have a software component, and an inverted feature tree models the hardware and operating system platform. The top-down feature tree follows the FORM practice of layering the feature tree. A three layer model is used with a capability feature layer, which models high level product features, a domain technology layer and then an implementation feature layer below.

The inverted feature tree can hold features arising from the operating system and / or the hardware platforms on which the software will operate. There can be relationships across the boundary between software and the operating system platform. The first form of *across-boundary* relationship is that of mutual dependency between an optional or alternative software feature in the upper feature tree and an operating platform feature. The implication is that the software feature requires or depends on the availability of the platform feature. If an optional platform feature is excluded then the software feature depending on it cannot be provided. Although this may be a low level feature, the implications can extend upwards to the capability feature layer.

The second across-boundary relationship is that of a *hardware-software feature alternative*. In this case we are dealing with the same feature which may be provided in software within one member of the family but in hardware within others. This kind of situation may arise in practice when the first products within an intended family are released with a certain feature provided in software, whereas in later models the feature migrates to a hardware device such as an ASIC, FPGA or DSP (we have encountered this phenomenon with some families of network products). It is worth noting that any kind of feature may participate in this relationship. Thus we could have a mandatory feature which in some products is provided via software and in others via hardware. Likewise we could have an optional feature which, within some products, may not be provided at all, but if it is provided then it may be provided as either hardware or as software.

2.5. Feature Properties and Property Relationships

The concepts of feature properties and property relationships, which are included in our approach, arise from original work at Nokia [15]. There are three kinds of feature property and two different kinds of property relationship. Possible kinds of property are:

a) Properties which are fixed for a family of products e.g. screen resolution on a family of mobile handsets.
b) Family variable properties which are fixed for any one product within the family, but may vary from product to product.
c) Variable properties which can change within one product

Properties are illustrated in our feature diagrams as rectangles attached via a broken line to the feature. The first form of relationship is called an *existence_modify*

relationship, and occurs between a feature and a feature property. It arises when the inclusion of an optional or alternative feature modifies the value of a property attached to another feature. The second form of relationship is called a *value-modify* relationship and occurs between a property of one feature and a property of another feature. The essence of this relationship is that changing the value of a property of one feature causes a change to the value of a property of another feature.

2.6. Graphical Notation for Rationalised Feature Modeling

Some aspects of the graphical notation for rationalised feature modeling are illustrated below in Fig. 1. In our prototype graphical editor for the notation, features are represented initially as dots with colour coding to indicate the main feature types.

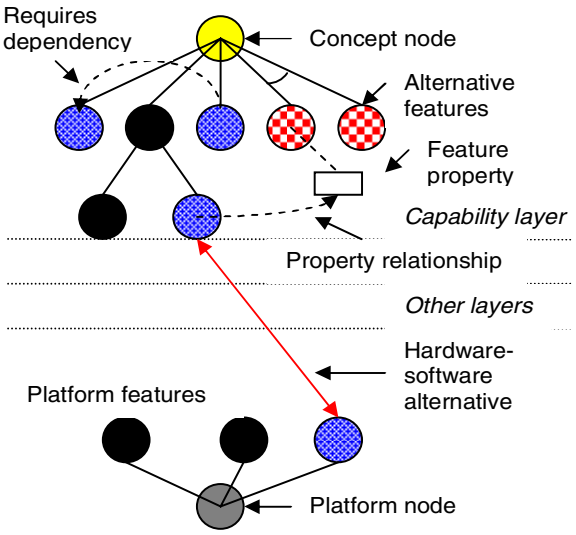

Fig. 1: Core Features of the graphical notation for Rationalised Feature Modeling. (For clarity, patterning has been used in place of the colour coding employed by the graphical editor. Checkerboard replaces red and diagonal hatching replaces blue)

Blue is used for optional features, red for alternative and black for mandatory. OR features are also shown in black with a black arc. Within the graphical editor, feature names are shown beside the dot symbols, although these are omitted here for clarity. Hierarchical relationships between features are indicated using an arrow for the *provided_by* relationship, and a solid line for the *consists_of* relationship. *Excludes* dependencies between features are illustrated with a dashed arc with double arrow, while *requires* dependencies use a dashed arc with single arrow.

Properties are illustrated as open rectangles, with dashed attachment lines linking them to features. *Existence_modify* relationships between features and properties use a dashed single arrow, while *value_modify* relationships between properties use a dashed double arrow.

3. Linking Behavior to Features

To capture feature behavior, it is essential to have a suitably abstract notation. There are several well known notations that are often used for modeling behavior. Within the UML, sequence and collaboration diagrams, Use Case diagrams, and activity diagrams can all be used for behavior capture. Use Case diagrams are often used to relate system interaction with users (Actors) and tend to provide high-level contextual information. Sequence diagrams, like message sequence charts, define behavior in terms of interactions between components or classes. The difficulty in using such notations in the current context is that behavior attached to features must be independent of any kind of component architecture. This is because the behavior modeling process will typically be performed as part of the process of evolving an architecture and before any definitive architecture has been identified. A notation that captures behavior, independent of components is required.

Activity diagrams capture behavior in terms of a sequence of actions. Although assignment of actions to components is possible (using swim lanes) it is not essential. Branching and concurrent paths are supported but support for timing constraints is weak. This is a limitation in the context of real time embedded systems.

In the PLUSS process Use Cases and Use Case Realisations are used as part of the requirements capture process. However, in the context of our work we are interested in more than requirements capture. Our aim is to use feature behavioral information as an aid to the architecture design process. For this purpose, none of these notations can be regarded as fully satisfactory for capturing feature behavior. However, a notation that is highly appropriate in this context is the Use Case Maps (UCM) path notation. Use Case Maps, like feature modeling itself, is a requirements capture notation. Its focus is on the capture of behavior at a reasonable level of detail. The founding concepts of the notation were introduced by Buhr [10] and have subsequently been extensively developed by Amyot and others [12]. Whereas feature modeling is inherently a notation targeted at product-line requirements, UCM was developed as a general purpose requirements modeling notation, aimed at providing an abstract, path-centric view of system functionality. It has now been

standardized and integrated into the User Requirements Notation (URN) [13].

In the UCM notation, behavior is captured in terms of a causal path. The path begins at a starting point, which may have triggering events and/or pre-conditions associated with it, and it continues to one or more end points, which may have associated resulting events and/or post-conditions. Along the way it may contain responsibility points, representing actions or responsibilities that must be discharged in the sequential order in which they appear. Paths may have loops, OR-forks, which indicate alternative paths, and AND-forks that give rise to concurrent path segments that may be executed in parallel. Alternative paths may be labeled with the conditions that give rise to their selection. Concurrent and alternatives paths may rejoin at an AND-join, or OR-join, respectively. Data items may be created or destroyed and may be placed on, or removed from a path. Data placed on a path is considered to move along the path. The notation supports the concept of a pool, which is a form of generic data store, and data items may be moved into or out of pools. Paths may contain waiting points representing situations where processing is delayed awaiting the arrival of some external event, or the satisfaction of some condition. Synchronization and rendezvous points may also be included. A timer feature allows the introduction of timed path segments, in which execution must complete within a defined time, otherwise the normal execution path is aborted in favour of an alternative error path. In the basic notation a path may cross one or more components. Components need not be shown if no component architecture is available, or they may be included as rectangles.

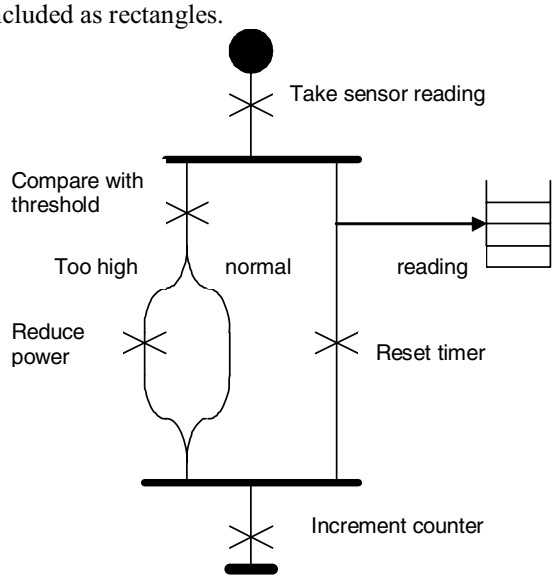

Fig. 2 An example illustrating the UCM path notation

Where a responsibility point is located on the path in such a way that it is coincident with a component, this denotes the fact that the responsibility is being assigned to that component.

An example of a UCM path is illustrated in Fig. 2. The path start symbol is the circular disc at the top and the path travels downwards. The first item after the start is the responsibility point labeled 'take sensor reading', and shown as crossed lines on the path. Responsibility points indicate an action to be taken at that point on the path. The horizontal bar indicates an AND fork, with the path dividing into two, possibly concurrent, sub-paths. The rightmost sub-path progresses with a data movement operation, illustrated by the arrowed line, which indicates that the reading is deposited in a 'pool', which is the feature shown on the right of the diagram. This is followed by a further responsibility point, labeled 'Reset timer', that indicates a further action to be taken. The leftmost concurrent path segment has a responsibility point labeled 'compare with threshold' followed by an OR-fork. An OR-fork indicates that the path divides, in this case into two alternative sub-paths. The rightmost alternative sub-path, which is labeled 'normal', contains no responsibility points, indicating that no action is to be taken. The leftmost alternative sub-path, labeled 'too high', has one responsibility point, before the two alternative sub-paths converge at an OR-join. The left and right concurrent paths then join and a final responsibility point occurs before the path termination symbol, which indicates the end of this fragment of behavior.

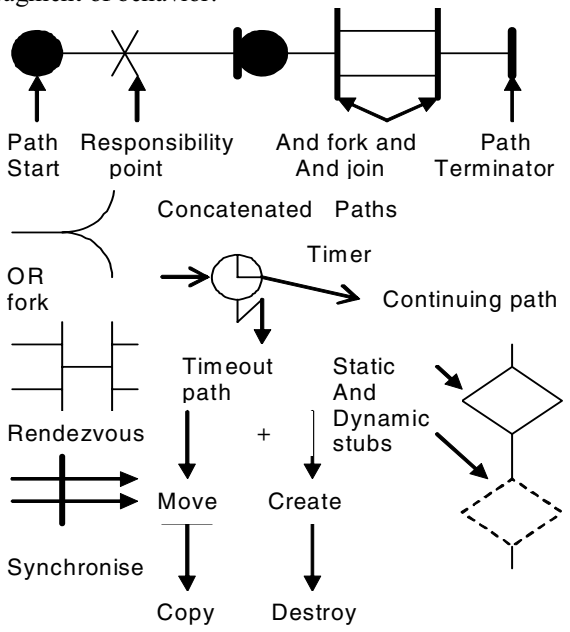

Fig. 3. Some visual elements of the UCM path notation

While this is obviously a trivial example, it illustrates some of the core features of the UCM path notation, namely path start and path termination symbols, responsibility points, forks and joins, as well as the pool symbol indicating a generic data storage facility of some form. It is clear that the notation is both highly abstract and at the same time quite intuitive.

The path start symbol may have associated pre-conditions and also specific triggering events. Path termination symbols may have associated post-conditions and also resulting events. Paths can have only one start point but sub-paths arising from either OR-forks or AND-forks do not need to converge, so paths can have multiple termination points. However paths may be concatenated and secondary paths may start from points on a parent path.

3.1 Static and Dynamic stubs

A very important concept in the UCM notation is the idea of stubs. When a stub is embedded within a path it acts as a placeholder into which further behavior can be plugged. Graphically a stub is represented as a diamond on the UCM path, and the plug-in behavior will be represented as another UCM path. Stubs can be of two types. The simplest are called static stubs and only one subsidiary path can be plugged in to them. In this case the plug-in serves as a definition of the behavioral detail at that point within the containing path. The second kind of stub, called a dynamic stub, is characterized by the fact that several alternative plug-in maps may be inserted in them. The UCM concept is that the actual plug-in may be selected at run time, depending on the satisfaction of associated pre-conditions. Dynamic stubs therefore represent points at which behavior may vary. However, the plug-ins that may be inserted into either static or dynamic stubs may themselves contain stubs that may in turn be either static or dynamic. So, paths may have stubs for which the plug-ins may contain stubs, essentially to any level of nesting. Clearly this mechanism provides scope for the capture of behavioral variability to any level of detail. This is a very important capability and one that is exploited fully in the integration of Use Case Maps with feature modeling.

Some further widely used elements of the UCM path notation are shown in Fig. 3. For a comprehensive account of all UCM path symbols and their meaning the reader is referred to the Draft specification [13].

4. Expressing Feature Behavior

To add behavior to a feature, in the simplest case, we attach a UCM path to the feature. The Rationalised Feature Modeling tool, currently under development, provides a facility to allow a UCM path to be defined for any feature, within a separate window. Once behavior has been assigned to a feature it will appear in the main window with a superimposed star. Clearly however, we need to observe some rules. To begin with some features may be inherently non-functional and therefore cannot have attached behavior. Moreover, feature models are inherently tree structured, with high level features possibly aggregating a number of behaviors rather than just one. We follow the principle that a UCM path will only be attached to a feature if that feature's associated behavior can be captured by one unique path. This clearly can result in a situation where, in some cases, a higher-level feature will have no path attached, but its children will possess paths. One possible interpretation is that the high level feature *aggregates* the multiple behaviors of its children (Fig. 4). At the level of software structure, one possible outcome of such a situation might simply be a class whose methods provide the behavior identified within the child features, although this is not the only possible consequence. Of course the situation is more complex if some child features are optional or alternative. In this case we have a variation point within the product family and a simple design based on a single class is unlikely to be appropriate.

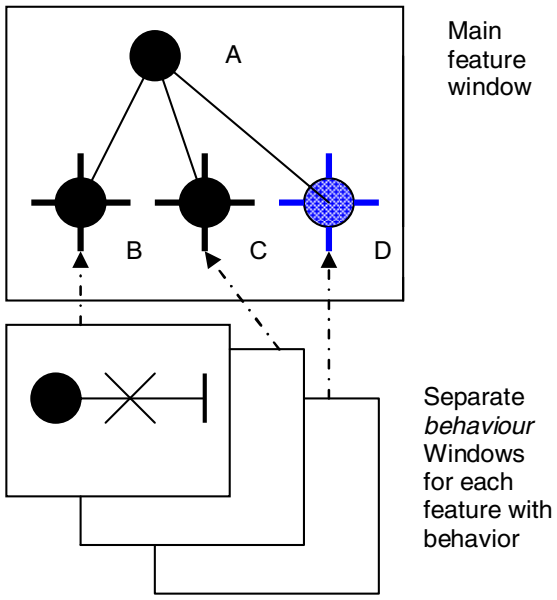

Fig 4. Features B, C, and D have associated behavior. Feature A *aggregates* the behavior of its children B and C and also D if it is present

A major mechanism for the capture of behavioral variability is provided by the availability, within the UCM notation, of stubs and, in particular, dynamic stubs. Consider a situation in which a feature has an

associated UCM path that contains stubs. It is perfectly appropriate for its child features also to have associated behavior. In fact the child feature behavior can provide the plug-in paths needed for the stubs within the parent feature's behavior. Where the child feature is a mandatory feature, its fixed behavior can be an appropriate plug-in for a static stub within the parent feature's path. A group of alternative child features, on the other hand, can provide alternative plug-ins for a dynamic stub within the parent path. Here we are dealing with a clear example of a variation point characterized by alternative nested behaviors (Fig. 5).

Another case is that of a dynamic stub within the parent, whose plug-in behavior is provided by an optional child feature. In this case, when the optional feature is supported, its behavior path is nested within the parent path. When it is absent, the plug-in is simply an empty path with no responsibility points or other behavioral elements.

This process of defining stubs in the path related to a parent feature, and plug-in paths for these stubs as behavior attached to child features, can obviously be repeated to any level of nesting within a feature tree. Thus a child feature may have an associated path that also contains stubs, with their plug-in behavior being provided by grand-child features of the original parent. The mechanism therefore allows for both the progressive refinement of behavior defined for a high level feature at lower levels within the feature tree, and for the progressive exposure of nested behavioral variability.

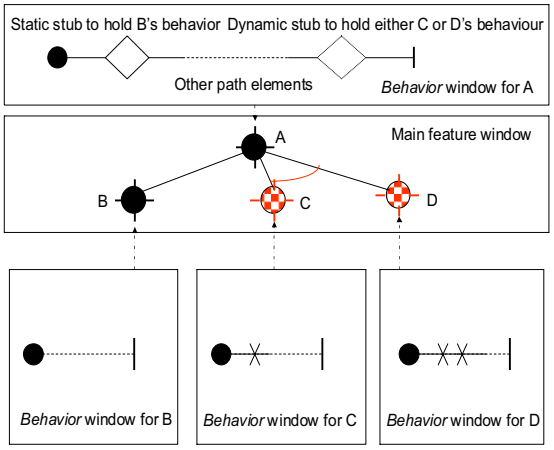

Fig.5 A's path contains a static stub that embeds B's behavior and a dynamic stub that embeds either C or D's behavior

It is this ability to relate feature structure to behavior and to behavioral variability that is the main benefit that derives from integrating the UCM path notation into feature modeling. Marrying the two notations requires very few changes to the core concepts of either. One point, however, where the normal semantics of UCM path elements needs some refinement concerns the role of dynamic stubs. In the UCM notation the choice of plug-in for a dynamic stub is deemed to occur at run-time and to depend on the satisfaction of pre-conditions related to each alternative plug-in. In the context of product-line requirements, it is more appropriate to relax this interpretation and to allow the binding of plug-ins to dynamic stubs to be decided at any appropriate point, from individual product design-time onwards. This is because alternative plug-ins, in this context, are related to alternative product features and the decision on which features to include within a product can clearly be taken at many points within the lifecycle.

To derive maximum benefit from the integration of these notations we clearly need to understand how to use the resulting framework to evolve, ultimately, a generic architecture. Thus we need to be able to identify those features to which behavior can and should be attached. We also need a general methodology for identifying the pattern of behavior appropriate to each feature.

5. Elaborating Feature Behavior

In the first instance a feature model, without any behavioral detail, will be developed essentially using product requirements information. Such information will need to be assembled from product specifications, discussions with stakeholders and other sources. In the case of the operating platform features, information from suppliers and potential suppliers of future and current hardware devices, alternative operating systems and quite possibly third party software will need to be collated and considered. Once this has been completed and the feature model itself has been created, the task of adding behavioral detail can begin. The first step is to identify those features to which behavioral detail can properly be added. There will certainly be some features that are essentially non-functional, to which the addition of behavior is inappropriate. There will be others, particularly towards the top of the feature tree which clearly do encompass functionality, but most properly align with major sub-systems rather than with specific functional activity. In general, if the functionality associated with a specific feature is not representable by means of a single UCM path, then it should not be modeled. Instead the child feature set should be so defined that that functionality can be divided over the children. Modeling of behavior should begin at the highest level at which this requirement can still be met. If it becomes clear that some features at a certain level meet this criterion while children of the same parent

feature do not, then this may indicate some anomaly within the feature model structure. Feature modeling in general is not an exact science and it is often possible to perform some re-factoring of the model structure.

5.1. Identifying Behavioral Detail with Scenarios

Identifying the actual behavior to be assigned to each feature requires experience within the domain and a clear understanding of the relationships between the feature and the underlying requirements. However, that will often not be sufficient. A general mechanism that can help with the identification of a feature's behavior is to develop a scenario that causes it to be exercised. Key questions to consider at this point are the pre-conditions or triggers that are expected to cause execution. Then the main responsibilities to be undertaken when the path executes can be identified. Responsibilities in the UCM notation are defined in general terms using text descriptions. However, individual complex responsibilities can be deferred to child features and incorporated via stubs. Path termination may create post-conditions or generate resulting events that trigger behavior associated with other features. An appropriate overall approach is to begin with the mandatory features at a given level within the tree and to prioritise those features. Thereafter, the behavior for higher priority mandatory features can be developed first. As work progresses, generated and consumed events, pre- and post-conditions and data stores (pools) that are read from or written to within path definitions are all registered within the tool. Lower priority features may thus become enabled, for example, by generated events or the satisfaction of pre-conditions and their required behavior then becomes easier to discern. The fact that the tool automatically registers events, conditions and pools makes it possible, for example, to identify events that are generated, but not consumed, or pools that are read from but not written to. Anomalies of this kind indicate, as a minimum, lack of completeness within the model, if not some more fundamental error.

Behavior for optional and alternative features can be identified once that for mandatory features at the same level is in place. In the case of optional and alternative features, hidden dependencies can sometimes emerge. For example, if an optional feature has a path that is triggered by an event that results only from execution of the path associated with another optional feature, this implies a relationship between the two optional features. Unless both are present together, the behavior associated with one will not be triggered at all, or the resulting event generated by the other will not be handled at all. Hence this situation implies the existence of a 'requires' relationship between the two features. Once behavioral details have been identified for all such features, any anomalies of the kind described above should become identifiable.

6. An Example: Modeling a Safety Function for Optical Networks

To illustrate the modeling notation and the tool we look at a topic from the domain of optical transmission networks. Optical network products are frequently highly complex and a complete feature model for such a device would contain very many features. A pragmatic approach is to divide such a system into a number of subsystems which are modeled and designed separately. For these products an important sub-system is concerned with safety. In optical networks, communication is based on laser light traveling between devices (nodes) within fibre. A fibre break can result in high energy laser light escaping with potential for injury to persons, or possible fire hazards. For this reason, all such equipment includes safety mechanisms which detect such occurrences and take rapid preventative action.

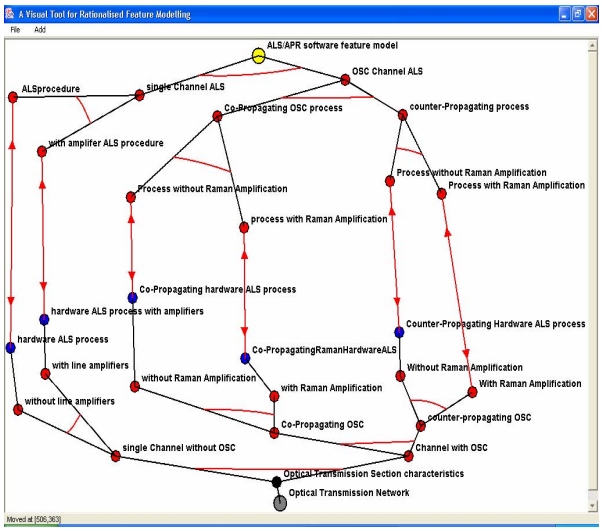

Fig. 6 Screen output showing feature model of ALS/APR safety procedures for optical network products.

A complicating factor arises from the fact that not all fibre communication channels have the same characteristics. An element may be deployed within a network based on any one of a number of possible link characteristics. Although there are similarities in the required responses to a fibre break, the detailed actions required can vary depending on the link characteristics. Furthermore, the need for backward compatibility with earlier generations of products (which may still be

deployed) means that historical safety procedures may still need to be supported.

Initially the safety procedure employed was designed for single channel fibre links and involved shutting down the transmitting laser completely when a fibre break was detected (a process referred to as automatic laser shutdown ALS). More modern fibre links often have an optical supervisory channel carrying only status and control information, while the traffic payload is carried in a separate higher power channel. In this case a fibre break can be managed by ceasing transmission of the payload data and reducing power on the supervisory channel (automatic power reduction, APR). Although incidents of this nature are generally reported to network management, who may assume manual control, it is normal to provide procedures for automatically restoring transmission power when the break is repaired (a restart process). The behavioral details of such processes are again dependent on the link characteristics. The key procedures involved are outlined within ITU-T standards recommendations [17], which may be used as the basis of a bi-directional feature model.

Fig. 6 shows such a feature model. Interestingly, this is an example of a situation where the important platform features that need to be considered, are not associated with the computing platform (processor / operating system). Instead, the platform layer captures some of the important alternative fibre link characteristics.

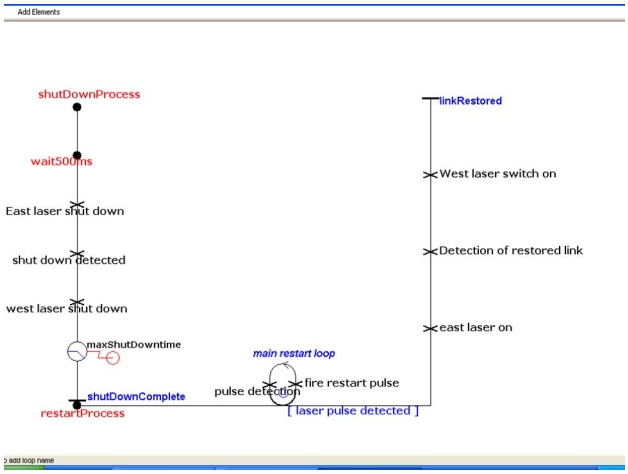

Fig. 7. Outline behavior of the ALS shutdown and restart processes for single channel links without optical supervisory channels

Shutdown and restart procedures are often supported in software, but the current trend is towards hardware-based implementations. Consequently the model shows hardware and software alternatives for each required safety procedure. For simplicity only the minimum required detail has been shown.

Figures 7 and 8 show UCM paths illustrating shutdown and restart procedures for two of the alternative situations (paths can also be defined for the remainder).

Although the feature model (Fig. 6) illustrates the various software feature alternatives and their relationships with the link characteristics, the most interesting information can be discerned from the related behavioral models, of which figs 7 and 8 illustrate only two.

Our interest here is primarily in demonstrating our feature and behavioral modeling notations. However, building models of this kind serves to clarify the behavioral differences induced by differing link characteristics and would therefore be an important early-stage activity in any attempt to devise a single protection system that could readily be configured to function with any form of link.

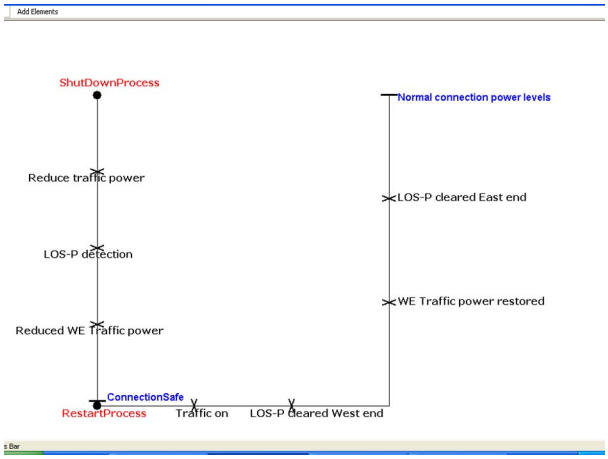

Fig. 8 Behavior of the APR shutdown and restart processes for a link having an optical supervisory channel, but without line amplification.

7. Conclusions and Future Work

Commonality and variability modeling techniques, particularly techniques centered on feature modeling, have been the topic of research for more than a decade. The developments reported here are oriented towards the needs of families of embedded software systems. In these systems variability in the platform features can induce variability in the matching software. The notion of bi-directional feature modeling with separate but interrelated models for software and platform features was first introduced in [9], and is further explored herein. The capture of this information is intended to facilitate the design of platform independent software. In this paper we have also described our strategy for

capturing, within the same model, the behavior associated with functional features. For this we make use of the Use Case Maps (UCM) path notation. This provides a suitably abstract means to attach behavior to features. Our approach to the integration of these notations exploits the availability of static and dynamic stubs within UCM. Stubs within a parent feature path can accept child feature paths as plug-ins where this corresponds to the issues being modeled. Dynamic stubs within a parent path allow alternative or optional child feature behavior to be readily embedded within the parent's behavior.

Although our experience in using this notation is limited, it does suggest that feature modeling and the UCM path notation are closely complementary notations that together allow the capture of commonality and variability in terms of both feature behavior and feature model structure. This provides a stronger starting point for architecture development. It can help to identify the principal data stores (pools) needed within a system and, for each such data store, the features whose behavior includes reading or writing to the data store. It can also help with the identification of the event messages needed within a system as well as the features whose related behavior either generates those events, or is triggered by them. These are valuable inputs for the architect. Much research is still needed to strengthen the link between feature modeling and architecture design, and to evolve a methodology for evolving generic architectures from feature models. Further development of the prototype tool is also required to improve its ability to cope with large models.

8. References

[1] L. M. Northrop, "A Framework for Software Product-Line Practice – version 3", *Software Engineering Institute*, 2001.

[2] Kyo C. Kang, G. C. Shalom, J. A. Hess, W. E. Novak and A. S. Petersen, "Feature-Oriented Domain Analysis (FODA) Feasibility Study", *Technical Report CMU/SEI 90-TR-21*, 1990.

[3] K. Lee, Kyo C. Kang, W. Chae and B.B. Choi, "Feature-based approach to object-oriented engineering of applications for reuse", *Software Practice and Experience*, Vol. 30, 2000, pp. 1025 – 1046.

[4] Kyo C. Kang, S. Kim, J. Lee and K. Lee, "Feature-Oriented Engineering of PBX Software for Adaptability and Reusability", *Software Practice and Experience*, vol. 29, 1999, pp. 875 – 896.

[5] S. Buhne, K. Lauenroth, K. Pohl, "Why it is not Sufficient to Model Requirements Variability with Feature Models", *Proceedings of the Workshop: Automotive Requirements Engineering (AURE'04),* Nagoya, Japan, 2004.

[6] K. Pohl, G. Bockle, F van der Linden, *"Software Product Line Engineering, - Chapter 5"*, Springer, 2005.

[7] H. Mei, W. Zhang, F, Gu, "A Feature Oriented Approach to Modelling and Reusing Requirements of Software Product Lines*", Proceedings of the 27th International Computer Software and Applications Conference (COMPSAC'03).* IEEE Computer Society Press, 2003.

[8] M. Eriksson, J. Borstler, K. Borg, "The PLUSS Approach – Domain Modeling with Features, Use Cases and Use Case Realisations*", Proceedings of the 9th International Conference on Software Product Lines (SPLC 2005),* Springer LNCS 3714, 2005.

[9] T.J.Brown, R. Bashroush, I. Spence, P.Kilpatrick, "Feature Guided Architecture Development for Embedded System Families", *Proceedings of the IEEE Working International Conference on Software Architecture, (WICSA),* 2005.

[10] R.J.A. Buhr, R.S. Castleman, "*Use Case Maps for Object Oriented Systems*", Prentice Hall, 1996.

[11] R.J.A. Buhr, "Use Case Maps as Architectural Entities for Complex Systems", *IEEE Transactions on Software Engineering*, Dec. 1998, pp 1131 - 1155.

[12] D. Amyot, "Use Case Maps as a Feature Description Language", Proceedings of FireWORKS '00, S. Gilmore and M. Ryan (Eds), *Language Constructs for Designing Features*. Springer-Verlag, 2000, pp. 27 - 44.

[13] ITU-T URN Focus Group (2002) Draft Rec. Z152 – UCM: "*Use Case Map Notation (UCM)*", ITU_T, Geneva, 2002.

[14] UCM web site at : http://www.usecasemaps.org.

[15] K. Czarnecki and U. W. Eisenecker, "*Generative Programming: Methods Tools and Applications, - Chapter 4*", Addison-Wesley, 2000.

[16] D. Fey, R. Fajta and A. Boros, "Feature Modeling: A Meta-model to Enhance Usability and Usefulness", *Proceedings of the 2nd International Conference on Software Product Lines (SPLC2),* Springer, LNCS 2379, 2002, pp. 198 – 216.

[17] ITU-T Recommendation G.664, "*Optical safety procedures and requirements for optical transmission systems*", International Telecommunication Union, 2003.