

# Capability & Potential for Formal Feature-oriented Reuse in Event-B

Ali Gondal, Michael Poppleton, Michael Butler

*School of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, UK*

---

## Abstract

**Context:** Event-B is a leading state-based language for formal modelling and verification of systems supported by an extensible Rodin toolkit. Its existing composition techniques provide a starting point for the investigation of capability for reuse via feature-based modelling. We contribute early methodology for formal development of software product lines (SPLs). An SPL is a set of related products built from a shared set of resources with a common base and having variabilities. Feature modelling has been widely used as a technique for building SPLs.

**Objective:** Our objective is to explore existing capability and future potential for Event-B, in the formal modelling, verification and reuse of domain assets, ultimately targeting verifiable SPL development. We will also suggest further requirements for tools and techniques in Event-B/Rodin for formal product line modelling.

**Method:** By modelling two-case studies in Event-B using different modelling styles, we explore current capability for feature modelling in Event-B. We show that Event-B decomposition techniques can be exploited for problem-space feature decomposition and solution-space architectural composition. We have also developed a feature-modelling tool for Event-B to experiment with our example case-studies.

**Results:** The case-study experiments show that the existing Event-B techniques can be used for feature-based modelling and revealed further requirements for feature modelling tool support. A guideline for feature-based modelling with Event-B/Rodin has been proposed based on the case-study experiments that could be used to achieve the benefits of reusing formal models.

**Conclusions:** By providing a prototype tool and guidelines for feature

modelling in Event-B, we can formally specify product line features that could be reused to build different variants of a product family. This would enable us to benefit from formal modelling and software reuse. This work has shown the potential of Event-B's existing tool and techniques for feature modelling and we have also generated tooling requirements and research questions for the future.

*Keywords:* Feature Modelling, Event-B, Formal Methods, Software Product Lines, Software Reuse

---

## 1. Introduction

### 1.1. Research Background & Motivation

Formal methods are mathematically based languages, tools and techniques for specification and verification of software and hardware systems [1, 2]. The correctness of models with respect to functional, safety and certain other requirements can be verified with theorem prover and model-checking tools. Proof activity and the investigation of failed proof obligations (POs) can serve as model and requirements debugging earlier in the software development life cycle (SDLC). It is also well known that cost of finding and fixing the bugs later in the SDLC is much more expensive than in the earlier requirements analysis and design phase [3]. So, formal modelling, which is carried out in the earlier phase, helps in finding such bugs and reducing the overall cost for fixing these [4]. The reduction in testing cost to offset increased cost in requirements/design phase can be seen in various industrial examples. Although more time is spent in the earlier phase of SDLC while using formal modelling, significant reduction in the overall development time has been reported. [5]

Successful application of formal methods in safety-critical systems can be seen in aerospace, transportation, defence and medical sectors [6, 7, 3, 8, 5]. There are several formal modelling languages around, some of the state-based formal methods include: Z [9], B [1], VDM [10], ActionSystems [11], Event-B [12] etc. and process-based formal methods include: CSP [13], ACP [14], CCS [15] etc.

Event-B [12] is one of the recent formal specification languages. It is a successor of B specification language [1], based on set-theory and first-order logic, proposed by Abrial and others. Event-B is supported by an Eclipse [16] based open-source toolkit called Rodin [17], which includes editors, provers,

animator, model checker and various other plugins. Event-B's mathematical language forms the core of the Rodin platform which can be easily extended by developing additional plug-ins on top of the Rodin core. Event-B models are specified at high abstractions levels and then refined gradually to model further details of the system. Each refinement step is checked for correctness by discharging proof obligations auto-generated by the tool with the help of theorem provers. An Event-B development consists of a model with various refinements. Once an Event-B development is refined down to concrete implementation, it can then be translated to executable code using code generators.

A software product line (SPL) is a set of related products which share a common base having significant variabilities to meet user requirements [18]. Each member of an SPL differentiates it from other members in terms of functionality or behaviour but share the common base. SPL development promises benefits of reusability, i.e., improved quality and reduced effort and time to market when building similar products by reusing parts of the systems that have already been developed. Feature modelling [19] has been established as a technique for modelling commonality and variability in SPLs. A feature has been defined to be a functionality or behaviour that is of some value to a stakeholder [20]. It can be a module in a programming language or a group of requirements in the requirements specification document. A feature model is a hierarchically arranged set of product line features [21]. It consists of a tree structured feature diagram with various features of a product line (PL) as tree nodes.

We have developed a framework that combines the strengths of both formal methods and SPL engineering, inspired by the work of Snook et al. [22]. This means that if we build a database of Event-B models for a PL that have already been proved, we can specify various products of the PL by configuring and composing these models in different ways. We propose guidelines and tool support through which the process of configuration and composition of Event-B models (with various refinements) would save lot of user effort, time and reduce overall development cost.

### *1.2. Our Contribution*

Our contribution is in providing methodology for feature-based reuse in Event-B, exploiting its existing capabilities, building further tools and techniques and suggesting modelling guidelines for SPL development using Event-B. We have provided notations for feature-modelling in Event-B to specify a

feature model of a product line with variabilities. We considered an Event-B development (collection of models) as a ‘feature’. Other possibilities of finer granularities of a ‘feature’ in Event-B were reported in our earlier work [23]. In feature modelling, features are atomic whereas we proposed composition and instantiation mechanism for features which are Event-B developments. After drawing a feature model, it could then be configured to model a particular instance of a product line. The feature model could be expressed with various constraints and how the features could be composed in a valid way to produce a PL member. We have developed a feature modelling tool (inspired by earlier feature modelling tools, e.g., [24]) as a plug-in to Rodin that can be used to draw feature models and configure these feature models that includes conflict resolution and composition of selected features. Our case-study examples showed that we can utilise existing Event-B (de)composition techniques to decompose a system into various features, refine them and recompose later to model the desired PL instance. The two case-studies modelled different systems in different styles to explore reusability and at the same time Event-B’s potential for feature-modelling. These also highlighted further tooling requirements and research questions. We have also proposed some guidelines that we think would be useful for SPL modelling using Event-B’s current tools and techniques.

### *1.3. Paper Organization*

Section 2 gives an overview of Event-B language syntax, refinement, generic instantiation and various composition approaches. Section 3 describes feature modelling concepts and notions, followed by our extension of these notions for feature modelling in Event-B in Section 4. Section 5 & 6 explain how we have modelled the two case-studies to address our research objectives. Section 7 explains the current state of feature modelling tool support in Event-B. Section 8 provides some guidelines for feature-oriented modelling and further tool support requirements. Related work is discussed in Section 9 followed by conclusion and future work in Section 10.

## **2. Event-B Language**

This section will discuss the syntax of the Event-B language with a simple example. We will also give a brief overview of refinement in Event-B modelling. This will be followed by a discussion on existing (de)composition

```

machine IntegralATM_0
sees IntegralATM_CO

variables bal

invariants
  @inv1 bal ∈ ACCOUNT → N

events
  event INITIALISATION
    then
      @act1 bal = ∅
    end

  event transfer
    any src_ac dest_ac am
    where
      @grd1 src_ac ∈ ACCOUNT
      @grd2 dest_ac ∈ ACCOUNT
      @grd3 am ∈ N
      @grd4 src_ac ∈ dom(bal)
      @grd5 dest_ac ∈ dom(bal)
      @grd6 src_ac ≠ dest_ac
      @grd7 am < bal(src_ac)
    then
      @act1 bal = bal ← {dest_ac ↦ (bal(dest_ac) + am), src_ac ↦ (bal(src_ac) - am)}
    end

  event deposit
    any acc am
    where
      @grd1 acc ∈ ACCOUNT
      @grd2 acc ∈ dom(bal)
      @grd3 am ∈ N
    then
      @act1 bal(acc) = bal(acc) + am
    end

```

Figure 1: Integral ATM abstract model

techniques in Event-B along with the existing reuse approach of generic instantiation. We will discuss the application of these existing approaches of Event-B later in the case-study Section 5.

### 2.1. Event-B Syntax

An Event-B model consists of a *machine* and can see multiple *contexts*. The machine specifies the behaviour or dynamic part of the system and the context contains static data which includes *sets*, *constants*, *axioms* and *theorems*. The sets define types whereas the axioms define properties of the constants such as typing etc. Theorems must be proved to follow from axioms. The machine *sees* context(s). State is expressed by machine *variables*. *Invariant* predicates provide typing for the variables and also specify requirements (e.g., safety or consistency) and any properties of a system that must always hold. The state transition mechanism is accomplished through *events* which modify the machine state. An event can have *guard* predicates which

must be true in order to enable the event to perform *actions*, e.g., assignment etc. Event *parameters* (also known as local variables) express input to, output from and local choice data within the event, as required. Variables are initialized by a special event called **Initialization** which is unguarded. An event has the following syntax:

$$e = \text{any } t \text{ where } G(t, v) \text{ then } A(v, t) \text{ end}$$

An event  $e$  having parameters  $t$  can perform actions  $A$  on variables  $v$  if the guards  $G$  on  $t$  and  $v$  are true. A model is said to be consistent if all events preserve the invariants, i.e., the invariant predicates must be true after any state updates. These invariant preservation properties, called proof obligations (POs), are the verification conditions automatically generated by the tool and then discharged using theorem provers to verify correctness of the model. Figure 1 shows an example of a complete Event-B model with a machine and its seen context. It has a variable  $bal$ , typed by the invariant  $inv1$ , as a partial function from set **ACCOUNT** to natural number to simply model an array of bank account balances. The  $bal$  is initialized as an empty set in the **Initialization** event. The set **ACCOUNT** is given in the context. There are two events, i.e., *transfer* and *deposit*, which update the variable  $bal$  when their respective guards become true. The local variables given in the **any** clause serve as input parameters here, e.g.,  $src\_ac$ ,  $dest\_ac$  and  $am$  in event *transfer*.

## 2.2. Refinement in Event-B

Refinement is a top-down development method and is at the core of Event-B modelling. We start by specifying a system at an abstract level and gradually refine by adding further details in each refinement step until the concrete model is achieved. A refinement is a development step guaranteeing every behaviour in the concrete model is one specified in the abstract model. It usually reduces non-determinism and each refinement step must be proved to be a correct refinement of the abstract model by discharging suitable refinement POs. Typically, we classify the refinement into horizontal and vertical refinements [12]. In horizontal refinement, we add more details to the abstract model to elaborate the existing specification or introduce further requirements of the system being modelled. In vertical refinement, the focus is on design decisions, i.e., transforming and enriching data types and the elaboration of algorithms. For example, data types become more concrete, i.e., move from set-theoretic (sets, relations) to machine-implementable

types (arrays, queues). In vertical refinement, the state of a concrete model is linked to the abstract model using *gluing invariants*. It is usually harder to prove vertical refinements compared to horizontal refinements since the gluing invariants increase PO complexity. A model is vertically refined after the horizontal refinement has been performed to introduce all the requirements of the system.

### 2.3. Generic Instantiation

Generic Instantiation [25] provides a reuse approach for Event-B models. The generic Event-B developments (called patterns) are instantiated to specific models through parametrisation achieved by refactoring. Such a pattern can include a refinement chain. This means that machine elements (e.g., variables, event names etc.) can be renamed and context parameters (e.g., sets, constants etc.) can be replaced with elements of the same type. This helps in reusing patterns to build models having similar characteristics and also maintains the validity of already discharged POs through to the instantiated models.

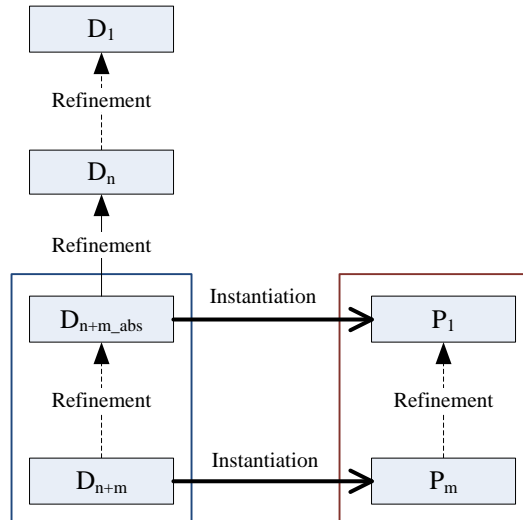


Figure 2: Instantiation of a generic chain of refinements [25]

Figure 2 shows an example of the generic instantiation for a generic pattern as a refinement chain ( $P_1$  to  $P_m$ ). A development  $D_1$  is refined by  $D_n$  which can be refined by instantiating pattern  $P_1$  through a user provided context resulting in  $D_{n+m\_abs}$ . The final refinement of pattern ( $P_m$ ) can be

instantiated to a specific development ( $D_{n+m}$ ) which turns out to be a refinement of  $D_{n+m\_abs}$ . Since the pattern has already discharged all the refinement POs, there is no need to prove the refinement POs for the instantiated developments (i.e., between  $D_{n+m\_abs}$  and  $D_{n+m}$ ). The final refinement of instantiated development ( $D_{n+m}$ ) can be further refined as usual. This work shows the reuse of generic patterns and the refinement POs. There is no tool support for generic instantiation as of yet and we suggest similar approach in our feature-oriented modelling framework discussed later but that requires more than just refactoring while instantiating Event-B developments.

#### 2.4. Decomposition & Composition in Event-B

**Decomposition:** When a model becomes too big to be easily refined, we need to decompose it into various sub-models (components) which can then be refined independently. In effect, this is complexity management by reducing the size of models, which keeps them understandable and reduces the number of POs to be proved for each model. This also allows the refinement of components in parallel by different teams. There are two types of decomposition in Event-B known as *shared-variable* decomposition (*SVD*) [26] and *shared-event* [27] decomposition (*SED*). Like the Event-B language, these techniques are influenced by earlier formalisms such as CSP [13] and Action Systems [11]. The refinement preserving nature of these decomposition techniques differentiates these from the feature-based decomposition with in the FOSD community.

In *shared-variable* decomposition style, shared variables are kept in all the components, and events are partitioned between components. Each shared variable  $v$  in each component  $C$  is affected by - i.e. has possible transitions defined by - every event  $e$  in every other component acting on that variable. To model this, for each such  $e$ , an *external event*  $e_{ext}$  is added to  $C$ . When a component is refined, shared variables and external events must not be refined. This type of decomposition corresponds to asynchronous shared-memory communication between components. Figure 3 (left) is an example of SVD where machine  $M$  is decomposed, with shared variable  $v2$ , by partitioning events into machines  $M1$  and  $M2$ . Thus event  $e3'$ , a new external event in  $M1$ , models the effect on  $v2$  of  $e3$  in  $M2$ . Similarly,  $e2'$  is an external event in  $M2$  modelling the effect on  $v2$  in  $M2$  of  $e2$ .

The *shared-event* style is based on shared events rather than shared variables. During the decomposition, variables are partitioned between components and shared events are split. A shared event is the one accessing



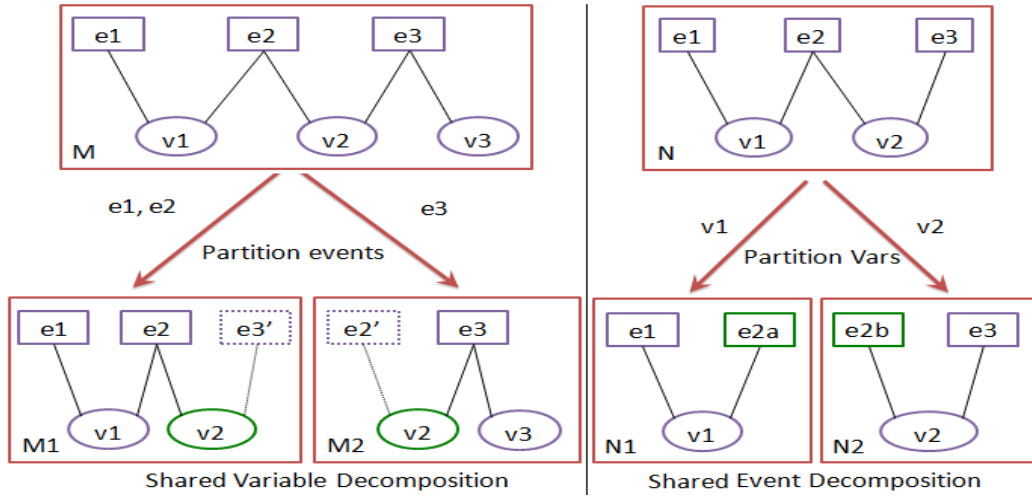


Figure 3: Decomposition types in Event-B

variables residing in different components. Figure 3 (right) is an example of SED where machine N is decomposed by partitioning variables into machines N1 (with  $v1$ ) and N2 (with  $v2$ ). Since event  $e2$  works on variables  $v1$  and  $v2$ , it will be split between N1 and N2. So, part of event  $e2$  ( $e2a$ ) that deals with variable  $v1$  becomes an event of N1 and its other part ( $e2b$ ) that deals with  $v2$  becomes event of N2. Event splitting is achieved by decomposing its parameters, guards and actions into two. This type of decomposition is considered appropriate for systems based on synchronous message passing.

Both the SVD and SED approaches have semantic support for modular refinement. This means that it has been shown for both approaches that decomposition preserves refinement: if we were to recompose components, even after further refinement steps, the composite would refine the single abstract model.

In practice the designer might choose to recompose - e.g., all code to run on a single processor - or might not - e.g., where component models are deployed on separate physical devices. The key point is that the final model is ‘correct by construction’. A decomposition plug-in [28] has been developed for the Rodin tool which can be used to demonstrate both styles of decomposition.

**Composition:** Since we are interested in composition, we would like to use the decomposition styles discussed above by inverting the decomposition method. For the shared-event style this is straightforward, whether one is

composing all, or just a subset of components, provided these do not have any shared state. For shared variable, composition is straightforward provided *all* components are included; if not, remaining external events are a problem. So, this brings up a tooling requirement to automatically generate external events for the components being composed. We could manually do this but it will be cumbersome and even more difficult when composing large number of components with many events. We will discuss this further in our example case-studies later.

*Fusion* [29] is another style of composition which allows the fusion of events when composing Event-B models having shared variables. During the fusion of two events, guards are conjoined and actions are concatenated. This style of composition, inspired by the above two decomposition styles, promises the support for reuse of models through composition as envisaged in feature-based development. The refinement preservation is also guaranteed as each of the abstract input feature events is refined by the concrete fused event.

### 3. Feature Modelling

#### 3.1. Introduction

Feature modelling is commonly used these days for SPL engineering. It was proposed as part of the feature-oriented domain analysis (FODA) method [30] in the early 90's. "Feature modelling is a technique for representing the commonalities and the variabilities among a set of systems in concise, taxonomic form" [19]. It can be considered as the activity of identifying externally visible characteristics of products in a domain [31]. It can also be used for scoping and developing domain-specific languages [32].

Feature modelling provides means to organise features into a feature model and configure them in order to build products of a product line. One of the many definitions of a *feature* is: "a logical unit of behaviour specified by a set of functional and non-functional requirements" [33] and usually referred as a property of the system that is of some value to the stakeholders. It is considered as a unit of reuse, specialization and composition (usually a piece of code in a programming language, e.g., a module) in SPL engineering. A feature has also been defined as an increment in program functionality [21].

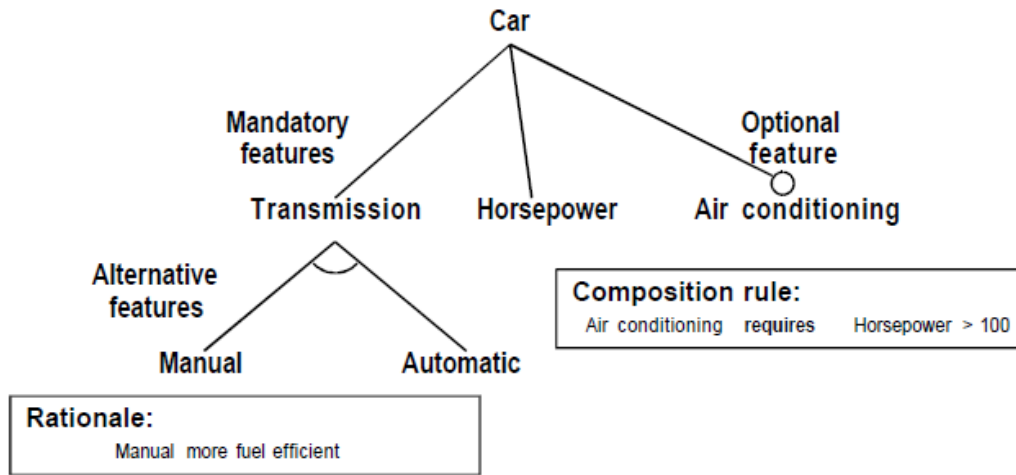


Figure 4: Example of a Feature Model (FODA style) [30]

### 3.2. Existing Feature Modelling Notations & Tool Support

There are various notations used for feature modelling which extend the original feature modelling notations from FODA. A feature model consists of one or more feature diagrams. The most basic of these notations include a feature diagram which represents a product family. A feature diagram is a hierarchical tree structure of all possible features that may occur in a product. The PL is the set of all distinct products that can be instantiated from the feature model. A feature can be optional, mandatory or mutually exclusive to another feature. Various composition rules are also supplied in textual form along with the feature diagrams. Figure 4 shows an example of a basic feature model drawn using FODA notations.

One of the common notations for feature modelling currently used is ‘cardinality-based feature modelling’ by Czarnecki et al. [32]. They extended the FODA notations by introducing feature and group cardinalities, feature attributes and feature diagram references. A feature diagram has a root feature containing further features. The cardinality of a feature or a group is provided as an interval (e.g., m..n). Features can refer to a feature diagram which improves reusability by referring to the feature diagram at various points in the feature model. This feature diagram reference is similar to our feature inclusion constraint discussed later in our approach.

Different configurations of features in a feature model will result in different instances of the system or members of the product line. This is done

through the feature configuration diagram where the user selects the desired features to be included in the feature model instance.

Several tools have been developed to support feature modelling for product line software engineering [24, 31]. Some of these focus on the demonstration of their extended notations and others implement the existing notations in different perspective and for different domains. Some of these also reflect on the improvements from the previously existing tools.

#### 4. A Feature Modelling Approach for Event-B

We adapted and extended the ‘cardinality-based feature modelling’ notation [32], so that we can build an Event-B specific feature modelling tool as a Rodin plug-in. The reason for doing so is because it was difficult to use existing feature modelling tools to specify Event-B features. A tool specific to Event-B would be able to adopt any modifications to the language as it is continually being improved. Also, this would enable us to make use of all the available plug-ins of the Rodin platform such as editors, provers, model checkers and animators. Our recent experiment of adding Event-B support to an existing feature modelling tool (FeatureHouse [34]) did not provide the same flexibility of feature configuration and composition as with our proposed feature modelling tool. It has proved difficult to integrate independently developed tools within the Rodin. One reason was an incompatibility of model interchange format between FeatureHouse (plain text) and Rodin (Event-B syntax aware XML). We would also require refactoring support while composing features and prefer to use a graphical feature composition tool rather than the script-based composition approach of FeatureHouse. So, our proposed feature modelling approach will be fully supported by a state of the art formal method.

##### 4.1. Our Feature Modelling Notations

We define a *feature* to be an Event-B development which consists of a machine with various refinements and their seen context(s). This definition of feature is different to the one presented in our earlier work [35], where we considered a single machine and its context as a feature. Since an Event-B feature can now have a chain of refinements, that makes it more difficult to deal with composing features having various refinement levels. We will discuss this later in our case-study examples.








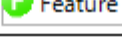
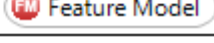
	Mandatory		Group with cardinality
	Optional		Includes
	OR Group		Excludes
	XOR Group	 Feature	 Feature Model

Figure 5: Our Feature Modelling Graphical Notations

The graphical notations used in our feature modelling framework are given in Figure 5. A feature model consists of a tree structured feature diagram whose root feature takes the name of the model. The filled circle on a feature shows that it is a mandatory feature and optional otherwise (empty circle). The features with a triangle attached represent group features which are containers for other features and specify any constraints on the selection of features within that group. One such is the *cardinality* constraint that indicates how many of the features in the group must be present in a particular instance. An empty triangle means exclusive OR (XOR) or otherwise a group with cardinality (e.g., ‘2..4’) and the filled triangle means OR, i.e., the cardinality ‘1..k’, where k is the number of features in that group. There are three types of connections that can be used to connect various model elements: child features, includes and excludes. The *includes* and *excludes* serve as constraints in the feature model. A feature can include other features, i.e., selecting that feature must also select the included features. Similarly, a feature can exclude other features and it is mutually exclusive, which means you can not have any two features with excludes connection between them in the configured instance. The leaf level features are actually mapped to Event-B developments during configuration.

Figure 6 shows an example feature model drawn using our feature modelling notation. The root feature PC has a group of five features with cardinality “5..5” which means an instance must select all five of the group features. The features *BeltCrane* and *AdvCrane* of the *CraneType* group are mutually exclusive (i.e., cardinality ‘1..1’ shown by an empty triangle) which means both of these can not be present in a particular variant of PC derived from this feature model. By selecting *BeltCrane* feature, both the *Deposit-Belt* and *Crane* become mandatory due to group cardinality constraint, i.e., ‘2..2’.

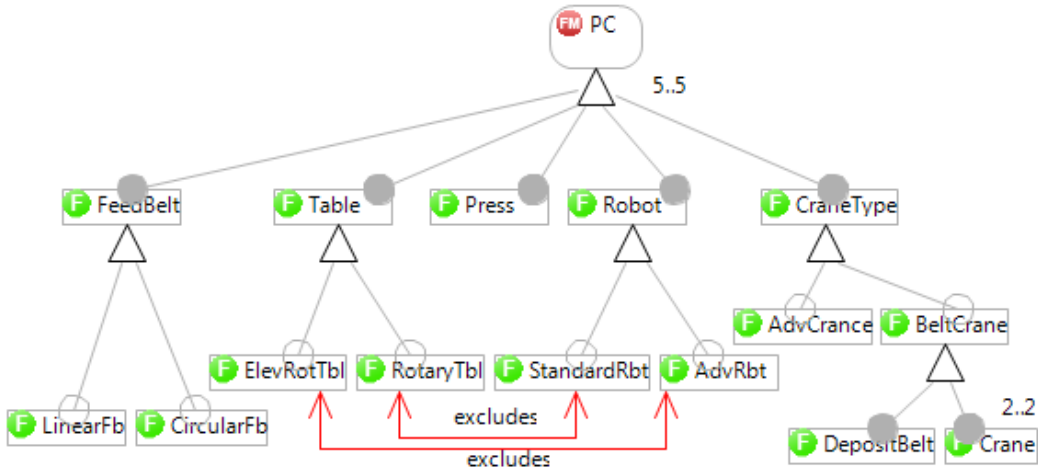


Figure 6: Example Feature Model

Our main extension to existing notations is that of including the refinement concept of Event-B. We consider an Event-B development as a feature, where each refinement model can be composed during the configuration for product line instantiation. We also provide *includes* and *excludes* constraints. Our feature modelling notation slightly differs from [36], as we allow feature refinements and a feature can exclude other features in a feature model. On the other hand, that work considers the reuse of proof obligations for invariant preservation and feasibility; we have not yet examined this.

## 5. Case-Study Experiments

The case-study experiments were carried out to explore whether existing Event-B tools and techniques can be used for feature-oriented modelling in Event-B, to gain the benefits of reusability. We would see how the Event-B methodology of single system, top-down refinement based development with decomposition can be combined with multi-system, side-ways in, compositional, reuse-oriented approach of feature modelling for SPL engineering. The case-studies would also show the opportunities and constraints for feature-based working in Event-B and would help in suggesting tools and techniques for future.

We have modelled two case-studies in Event-B, i.e., production cell (PC) and ATM. Our first case-study (PC) suits the typical top-down refinement approach of Event-B. At first sight, this case study does not show obvious

reuse potential in terms of functional requirements features; variability arises as different possible connection topologies between PC components. We then model it in a different way by considering a fine-grained view of features as controllers that were generic and revealed more reuse opportunity for PC modelling. We would explain how we modelled PC in different ways to explore Event-B’s capability for feature modelling using its decomposition and genericity techniques to exploit reuse. In order to support our findings of the PC case-study, we then modelled the second case-study(ATM). ATM example is more reuse-oriented in term of functional features and suits the traditional feature modelling approach where we can have a product line of ATM systems having different features. This has revealed a pattern of modelling that could be used as a guideline for SPL modelling in Event-B for future users. Both case-studies will use existing (de)composition approaches of Event-B discussed earlier.

We have developed a prototype feature composition tool [37] which was used for composing Event-B feature during the case-study experiments. This tool allows the free-style composition (including fusion) in whatever ways the user wants but it does not automatically discharge proof obligations for the composite model. This means the user need to reprove the composite model to make sure that the composition was performed correctly. The case-study results will show how we can avoid reproof effort by following a particular modelling pattern involving different (de)composition styles.

### 5.1. Production Cell

The production cell (*PC*) [38] is an example of a reactive system which has been modelled in more than 30 formalisms [39, 40, 41, 42]. It has also been specified in the B formal method which is a predecessor of the Event-B language [43]. It is a metal processing production line where metal blanks are routed to a press for forging, then routed away from it after processing. Figure 7 shows the top view of the production cell plant. Metal blanks enter into the system through the feed belt and are dropped on to the elevating-rotary table when the table is empty and in the loading position. The table elevates and rotates to a position so that the first robot arm can pick up the blanks as the robot arms are positioned at a different horizontal plane. The robot rotates anti-clockwise to drop the blanks in the press. The press forges the blanks which are picked up by the second robot arm and then dropped on to the deposit belt. A moving crane then picks the blanks from the deposit

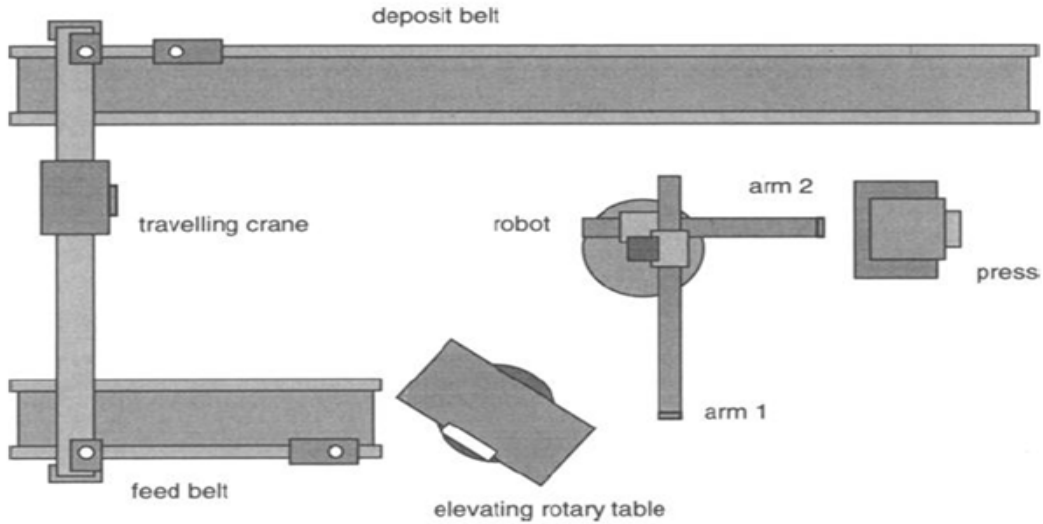


Figure 7: Production Cell Plant [38]

belt, that have not been forged properly, and brings them back to the feed belt for reprocessing.

We have modelled PC in three ways, i.e., physical component-based, controller-based (functional) and a domain-specific modelling approach based on static variability as described below in detail. This allows us to use different methods of modelling the same system in Event-B and analysing our approaches to feature-based modelling using existing tools and techniques in Event-B.

**PC Component-based:** In the physical component-based modelling approach, we started with an abstract model of the production cell and refined up to a few levels by adding more details in each refinement step. At the abstract level (see Figure 8), we only have one event *Operate* which models the processing of blanks from ‘forged’ to ‘unforged’ state through the variable ‘blanks’. Initially, all the blanks are set to ‘unforged’. We refined this model by introducing various physical components of PC (e.g., table, robot, press etc.) and where the blanks are positioned across these components. This is further refined by modelling operations taking place at different components of PC, e.g., the elevation and rotation of table etc. In the next refinement, we introduced control functionality of robot arms and the movement of belts for delivering blanks. Some safety requirements were also modelled such as arms should not be extended while the robot



```

machine PC_0 sees PC_CO
variables blanks
invariants
  @inv1 blanks ∈ BLANKS → STATUS
events
  event INITIALISATION
  then
    @act1 blanks = unfBlanks
  end
  event Operate
  any b
  where
    @grd1 b ∈ dom(blanks)
    @grd2 blanks(b) ≠ forged
  then
    @act1 blanks(b) = forged
  end
end

```

Figure 8: PC Abstract Machine

is not positioned correctly. Another refinement was done to introduce the functionality of the ‘crane’ component. These were all horizontal refinement steps.

At this stage, the model became quite large and was difficult to refine further as a whole. So, we decomposed the model into various physical components (sub-models) of the PC (i.e., *feed belt*, *table*, *robot*, *press*, *deposit belt* and *crane*). We used both types of decomposition, i.e., shared-variable (*SVD*) and shared-event decomposition (*SED*). At first, it seemed appropriate to use SVD since different components were sharing variables, e.g., all components shared the variable *blanks*, which models the status of blanks at any component. So, during the decomposition, events related to a particular physical component became events of that sub-model and any events of the sub-model involving the shared-variable became external events in all the other components. For example, event *loadTable*, moved to the *table* sub-model, became an external event in all the sub-models for other physical components of PC and so on.

In order to explore whether we can use SED to decompose the integral model into sub-models, we had to prepare the model to be decomposed using the SED style. For this, we had to partition(localize) the variable *blanks* for each component (i.e., *blanksOnFb*, *blanksOnTbl* etc.), so that there is no more shared-variable. In this case, we partitioned the variables into various sub-models along with their related events. Figure 9 shows how an event

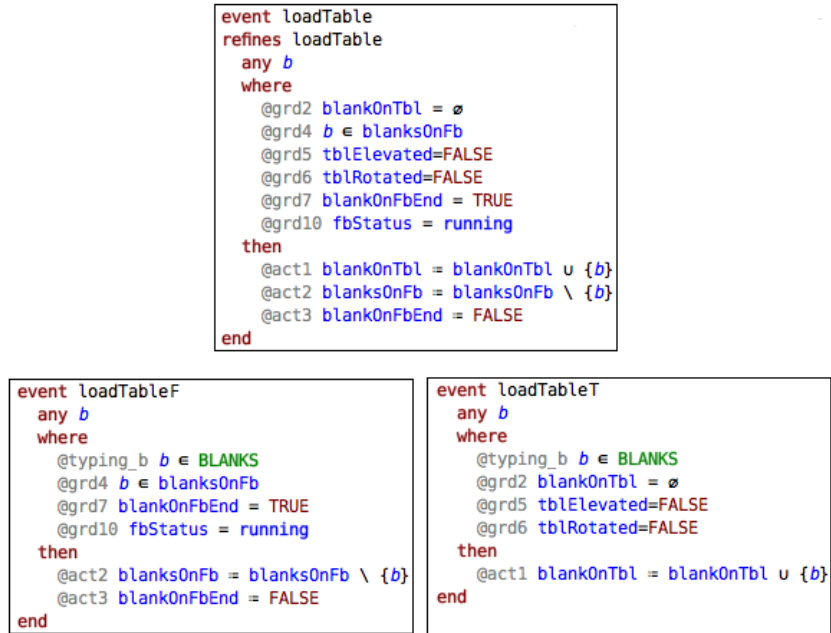


Figure 9: Event Splitting for SED

*loadTable* is split into two events, i.e., *loadTableF* and *loadTableT* for *table* and *feed belt* components. We simply split the guards and actions into two. If a guard or action of an event is complex and can not be split then it must be simplified in the preparatory step to be split into two. Note that we had to do vertical refinement in order for us to perform SED unlike SVD where we only carried out horizontal refinements before the decomposition. So, it depends on the type of system being modelled and for distributed systems, the SED approach seems more appropriate.

After decomposing the model into sub-models, we could then refine each of these sub-models independently. In case of SVD, we had to maintain the restrictions of the SVD style while refining these sub-models, i.e., to ensure that the shared variables and external events were not refined. We further refined the ‘press’ sub-model vertically by introducing actuators and sensors. This involved another three levels of refinement and was done using a refinement pattern for control systems [44]. Other sub-models could also be refined similarly.

This gives us a product of PC which models a particular topology (see Figure 10) and how the physical components are connected to model the pro-

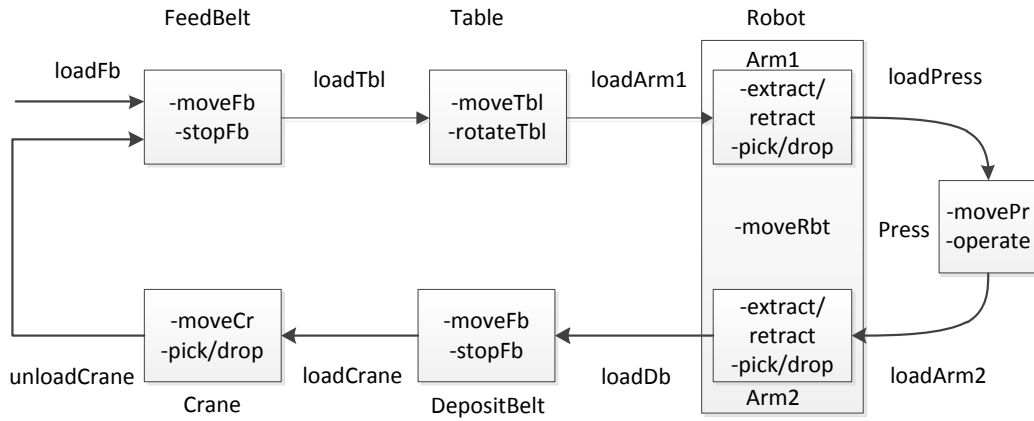


Figure 10: PC Topology 1

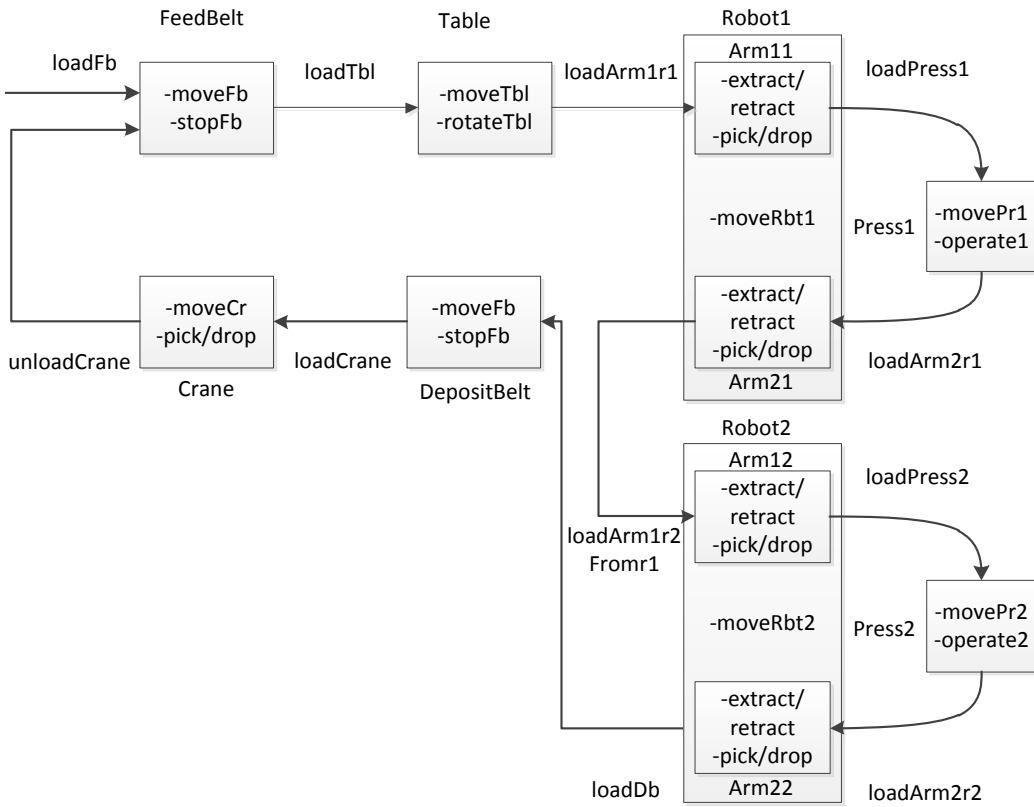


Figure 11: PC Topology 2

duction cell plant. The figure is designed to visualise the topology in terms of events and helps in reuse for instantiation of alternative topologies. The events placed inside the boxes are local for the components and those placed between the boxes represent shared events containing topological information for modelling the connectivity of the components. This is an example of domain specific instance modelling with Event-B. Each different topology is an instance of PC product line and we can build more variants of PC by selecting a different configuration (or topology) of these physical components. For example, if we want to model a production cell with two press components for processing blanks twice and using two robots. We can call this ‘topology 2’ where *robot1* picks blanks from the *table* and drops onto *press1* and *robot2* takes the blank from *robot1* which picks it from *press1* and drops on *press2* (see Figure 11). Here we are interested in exploring to what extent we can reuse the models of *topology1* while modelling *topology2* and hence the proof effort. For *topology2*, we had to do instantiation and refactoring to simply duplicate the functionality of existing components. This means that we would not have to reprove the models which have already been proved for *topology1*. This is because renaming of elements would not affect the POs and is currently supported by the refactory plug-in [45]. We only had to prove the POs generated for any additional information modelled in the second topology. For example, we specify that *arm1* of *robot2* collects the blank from *arm2* of *robot1* unlike picking it from the *table* in *topology1*.

Figure 12 shows the refinement architecture for modelling the two topologies and their components as achieved after decomposition. An example of event instantiation while modelling *topology2* after *topology1* is shown in Figure 13 where event *loadPress* is duplicated for the two presses (events: *loadPress1* & *loadPress2*). This type of instantiation and refactoring has no proof burden. The POs for *topology2* were discharged in the same way as *topology1*. Figure 14 shows the number of POs for both topology developments at different refinement levels and how these were discharged, i.e., automatically or interactively. The Figure would be more interesting if we could reflect the percentage of POs reused when modelling *topology2* after *topology1*. So, this shows that we can reuse the existing models and their proofs if we have tools to automate the instantiation and refactoring. Hence this exercise generated additional tooling requirements which are discussed later.

**PC Controller-based:** In the controller-based PC modelling, the functional requirements to model the behaviour of each controller were grouped

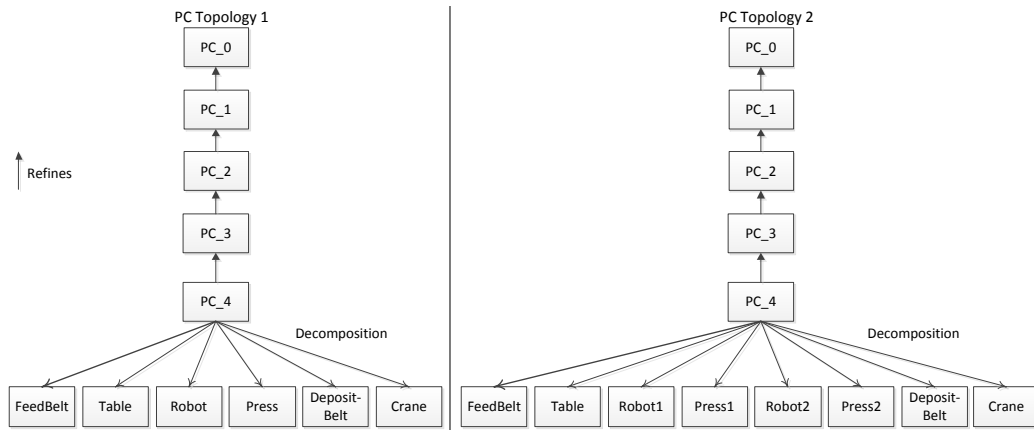


Figure 12: Refinement architecture for modelling the two topologies

```

event loadPress
  any b
  where
    @grd1 b∈BLANKS
    @grd2 pr ∈ran(position)
    @grd3 b∈dom(position)
    @grd4 position(b) = arm1
    @grd5 pressPos=mid
    @grd6 robotPos=pos2
  then
    @act1 position(b) = pr
  end

event loadPress1
  any b
  where
    @grd1 b∈BLANKS
    @grd2 pr1 ∈ran(position)
    @grd3 b∈dom(position)
    @grd4 position(b) = arm1r1
    @grd5 press1Pos=mid
    @grd6 robot1Pos=pos2
  then
    @act1 position(b) = pr1
  end

event loadPress2
  any b
  where
    @grd1 b∈BLANKS
    @grd2 pr2 ∈ran(position)
    @grd3 b∈dom(position)
    @grd4 position(b) = arm1r2
    @grd5 press2Pos=mid
    @grd6 robot2Pos=pos2
  then
    @act1 position(b) = pr2
  end

```

Figure 13: Event instantiation example for PC topology2

	PC Topology 1				PC Topology 2			
Ref. level	Auto	Manual	Reviewed	Total	Total	Auto	Manual	Reviewed
PC-0	3	-	-	3	6	6	-	-
PC-1	36	1	-	37	56	55	1	-
PC-2	22	4	-	26	26	16	10	-
PC-3	41	4	-	45	59	55	4	-
PC-4	10	-	-	10	8	8	-	-
PC-5	84	-	-	84	190	188	2	-
PC-6	20	-	-	20	40	40	-	-
PC-7	188	-	8	196	432	401	11	20
Total	404	9	8	421	817	769	28	20

Figure 14: POs for two topology developments

together as a feature. So, the requirements specification was decomposed into various controller features. We also generalised the requirements for each of the controller so that we could model generic controllers which could then be specialised and reused for modelling various controllers of different physical components of PC. Hence, the controller-based modelling of PC was a result of decomposition plus generalization. Table 1 shows part of the requirements specification for the *table* feature of component-based PC and the *movement* feature for controller-based PC. This shows how we can define the feature in terms of requirements for two styles of modelling the PC while making the features more reusable. The compositional requirements are modelled while actually composing various components, this may include topological information and how components are connected together. The controller-based PC models consisted of *loader*, *movement*, *rotation* and *magnet* controllers. A member of PC product line could be modelled by instantiating and composing these controller-based reusable features. These features were then refined independently. We discuss the refinement of *magnet* and *movement* features below where we introduced sensors and actuators in various refinement steps using the pattern for refining control systems as suggested in [44].

**Magnet Controller:** At the abstract level, we have events for picking and dropping of blanks by a component. A component which has not already

Table 1: Requirements description for table and movement features of PC

<p><u>Table Component</u></p> <ul style="list-style-type: none"> <li>- Move table upwards/downwards</li> <li>- Rotate table clockwise/anti-clockwise</li> <li>- Table must not rotate when its at low position</li> <li>- Table must not move down if it is rotated (rotate backward first and then then move down) or if it is already not elevated</li> <li>- The table must not rotate clockwise if it is in a position to deliver blanks (unloading position)</li> <li>- The table must not rotate at all if it is not elevated</li> </ul> <p><u>Compositional Requirements</u></p> <ul style="list-style-type: none"> <li>- Drop blank on table from feed belt when it is in the loading position (not elevated and not rotated)</li> <li>- Robot picks blank from table when it is in unloading position (elevated and rotated)</li> </ul>	<p><u>Generic Movement Controller</u></p> <ul style="list-style-type: none"> <li>- Move a component from position A to position B and vice-versa</li> </ul> <p><u>Instantiation Requirements</u></p> <ul style="list-style-type: none"> <li>- Extract/Retract Arm1</li> <li>- Extract/Retract Arm2</li> <li>- Move Feed Belt/Deposit Belt</li> <li>- Move the Table upwards/ downwards</li> <li>- Move Press to upper/lower/middle position</li> <li>- Move Crane To and from Feed Belt/Deposit Belt</li> </ul> <p><u>Compositional Requirements</u></p> <ul style="list-style-type: none"> <li>- Extract/Retract Arm1 if robot is facing table or facing press while press is in middle position</li> <li>- Extract/Retract Arm2 if robot is facing deposit or facing press while press is in lower position</li> <li>- Table must not move down if its rotated (rotate backward first and then move down) or if it is already not elevated</li> <li>- The press must not move downward if it is in lower position and must not move upward if it is in upper position</li> <li>- Crane should only move towards feed belt if it is positioned on deposit belt and vice-versa</li> </ul>
--	---

picked a blank can do so and a component which has picked a blank can drop it. The feature will be instantiated to a specific component such as a *crane* or a *robot arm*. The model is quite abstract and the details are added later in the refinements and during specialization. In the first refinement, we added sensor for magnet which informs the controller whether a blank has been picked up or dropped off. An electromagnet switch acts as an actuator for the magnet which performs the pick and drop of blanks. We have events for starting and stopping the magnet and switching the sensor on and off. In the second refinement, we differentiate between the actual and sensed values of the sensors. This is done to model the system closer to reality, as the actual value of the sensors at some point in time will be different from the sensed values. Similarly, in the third refinement, we refine the actuation where controller sets the actuation of the motor before the motor can be actuated. Here we split the actuation events into two, i.e., an event for setting the actuation of magnet by the controller and the event for magnet to actuate accordingly.

***Movement Controller:*** At the abstract level, we have events for moving a physical component forward and backward between two positions.

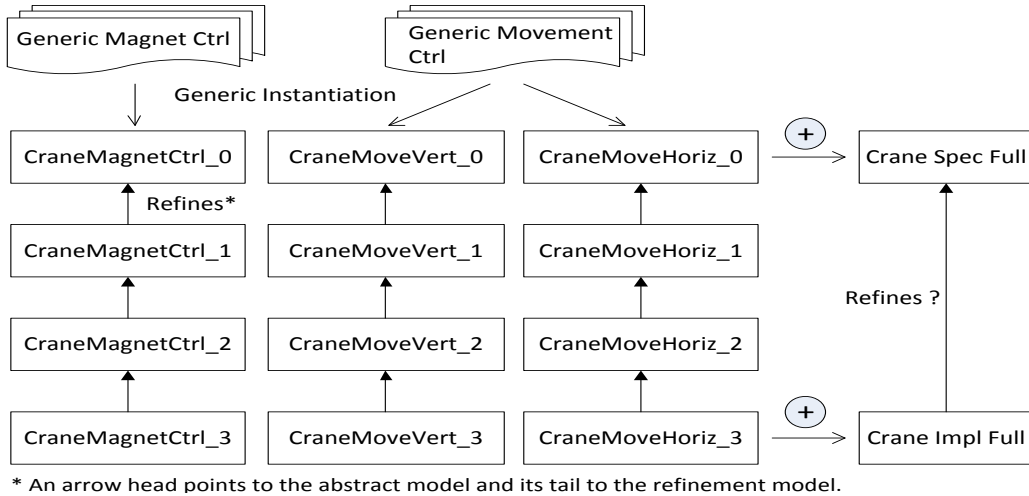


Figure 15: Crane Instantiation

The feature will be instantiated to a specific component such as a *press* or a *crane*. During the first refinement, we added sensors for the two positions and a motor for moving backward and forward. Events were added for starting and stopping the motor at different positions and switching the sensors on and off. In the second refinement, we differentiate between the actual and sensed values of the sensors as discussed earlier. Using the same refinement style, at third level of refinement, we differentiate between setting the motor's actuation by the controller from its actual movement.

### ***Instantiation & Composition:***

The *magnet* and *movement* controllers provide us refinement chains of generic Event-B models for the two features. In order to model any component of the PC, we need to instantiate and compose these chains of models. The composition is done in two phases. In the first phase, we compose instantiated models to build a component (e.g., *crane* or *robot*) and in the second phase, we compose all the components while providing topological information. For example, if we want to model the *crane* component, we have to specialize one instance of the *magnet* controller to pick and drop blanks and two instances of the *movement* controllers for moving the *crane* horizontally and vertically, as shown in Figure 15. In this example, we have three refinements in each development which align well during the composition. This alignment issue needs to be explored further to address the composition of Event-B developments having different number of refinements. Figure 16



<pre> event PickBlank   any b   where     @grd1 X comp X ∈ ran(position)     @grd2 b ∈ dom(position)   then     @act1 position(b) = X comp X   end </pre>	<pre> event CranePickBlank   any b   where     @grd1 crane ∈ ran(position)     @grd2 b ∈ dom(position)   then     @act1 position(b) = crane   end </pre>
---	--

Figure 16: Event Specialization for Crane

shows a simple example where event *PickBlank* of *magnet* controller is specialized for the *crane* component. Here the generic model parameter *XcompX* is replaced by *crane* provided both of these are of the same type. For now, we use *X...X* as a syntactic convention to model a generic parameter, given that the current Rodin tool does not support generics.

The composition of abstract level models from each refinement chain would give us an abstract specification for the *crane*. We also had to do some guard strengthening and add some invariants during the composition. The composition of implementation level models for each refinement would provide us with the implementation of the *crane*. Again extra guards for events and invariants were needed. Figure 17 shows two events from *magnet* and *movement* controllers for picking up blanks by *crane* and the movement of *crane* towards *feed belt* (before composition). Figure 18 (after composition) shows these events with extra guards added during the composition. For example, *grd3* of *CranePickBlank* event specifies that the *crane* can only pick a blank when it is positioned on the deposit belt. Similarly, *grd2* of *moveToFB* event in Figure 18 specifies that the *crane* can only move towards the *feed belt* if it has picked up a blank. The guard *grd2* of *CranePickBlank* event means it can pick any blank in the system. When we finally compose all the components to model the entire PC in the second phase of composition, we will need to strengthen this guard to say that the *crane* can only pick a blank from the *deposit belt*. Here we would need to give topological information of PC in terms of how different components are connected to each other as discussed in the components-based PC example earlier.

We call this style of composition ‘feature composition’ where additional information can be added during the composition. As of yet, this style of composition does not guarantee refinement preservation between the com-

<pre> <b>event</b> CranePickBlank      (before) <b>any</b> b <b>where</b>   @grd1 crane<math>\in</math>ran(position)   @grd2 b<math>\in</math>dom(position) <b>then</b>   @act1 position(b) = crane <b>end</b> </pre>	<pre> <b>event</b> moveToFB          (before) <b>where</b>   @grd1 cranePosHorz=posDB <b>then</b>   @act1 cranePosHorz = posFB <b>end</b> </pre>
---	--

Figure 17: Events of Magnet and Movement Controllers

<pre> <b>event</b> CranePickBlank      (after) <b>any</b> b <b>where</b>   @grd1 crane<math>\in</math>ran(position)   @grd2 b<math>\in</math>dom(position)   @grd3 cranePosHorz = posDB <b>then</b>   @act1 position(b) = crane <b>end</b> </pre>	<pre> <b>event</b> moveToFB          (after) <b>where</b>   @grd1 cranePosHorz=posDB   @grd2 crane<math>\in</math>ran(position) <b>then</b>   @act1 cranePosHorz = posFB <b>end</b> </pre>
---	--

Figure 18: Guard strengthening of events during composition

posed abstract and implementation models (see ‘refines?’ in Figure 15). In order to deal with this kind of composition, we need support for proof reuse. By this we mean to find a way of automatically discharging composite POs with the help of already discharged POs of the components being composed. This requires further work. We will discuss an alternative approach in ATM example where we can avoid reproof by following a modelling pattern. In comparison to the component-based approach discussed earlier, this style of modelling SPLs in Event-B seems more appropriate because it provides more reuse opportunities.

The shared-event composition could not be applied here due to the shared state between the components being composed. The shared-variable composition approach is too constraining and could only be used here if we start with an abstract model containing the functionality of both the *magnet* and *movement* features. We could then decompose these into two, refine each of these, instantiate for the crane and compose to build the crane model. The ATM case-study discussed in Section 6 further explores these issues and suggests the modelling style through which we could use existing techniques of Event-B to achieve partial reuse of existing specifications, when modelling variants of a product line.

**PC – Domain-specific SPL modelling through contexts:** We also modelled a generic component-based PC which supports the two topologies mentioned earlier. The variability is provided through the context which means the machine for both the topologies will remain same and we could have a different topology by just switching the contexts. We modelled the topology of PC in the context, i.e., we specify in the context what are the physical components and how these could be connected to each other. The machine is modelled in a generic way, e.g., an event *loadComponent* is used to model the loading of blanks onto a PC component and an event *passBlankBtwCpts* for moving a blank from one component to another. This is shown in Figure 19. The invariant *inv1* defines the position of blank at a component. The topology is given in the contextual part at the bottom where *cptGraph* defines the components graph, i.e., valid ways of connecting different components. This is then used in *grd7* of the *passBlankBtwCpts* event to make sure that the blank is passed between connectable components.

The advantage of modelling in this way is that we will not need to reprove a variant of PC resulting from static variability through context switching. The disadvantage is that the modeller may not visualise various events of the machine for a particular topology as this machine may not be instantiated. This could be a useful domain modelling activity for exploring variability of a product line in a distributed environment.

**Evaluation:** By modelling the PC in three different ways, we have explored to what extent we can use existing Event-B tools and techniques for feature-based product line modelling. This also enabled us to figure out the requirements for future tooling and techniques (discussed later) that can further facilitate such development approach to benefit from reuse of existing models and their proofs.

The first style of modelling – component-based – is a natural approach of modelling in Event-B which used both types of decomposition techniques to reduce the complexity of modelling and proving by decomposing large models into smaller sub-models. It also showed that we could model variants of PC product line by configuring the different topology of physical components of PC. This is not a typical example for feature-based modelling as the entire model needs to be redone again for building a second topology after doing the first one. But this allowed us to explore requirements for instantiation and refactoring tool support that could be useful in automating this approach.

The second approach of controller-based modelling is more feature-oriented as we have modelled generic reusable features that could be instantiated and

```

invariants
@inv1 blankPosition ∈ BLANKS ↔ CPT
event loadComponent // load a blank on to a Component
  any b cpt
  where
    @grd1 b ∈ BLANKS
    @grd2 cpt ∈ CPT
    @grd3 cpt ∉ ran(blankPosition) // cpt is vacant
  then
    @act1 blankPosition(b) = cpt
  end
event passBlankBwCpts // pass a blank from one to another component
  any b cpt1 cpt2
  where
    @grd1 b ∈ BLANKS
    @grd2 cpt1 ∈ CPT
    @grd3 cpt2 ∈ CPT
    @grd7 cpt1 ↦ cpt2 ∈ cptGraph
    @grd4 b ∈ dom(blankPosition)
    @grd5 blankPosition(b) = cpt1 // blank is on cpt1
    @grd6 cpt2 ∉ ran(blankPosition) // cpt2 is vacant
  then
    @act1 blankPosition(b) = cpt2
  end

constants fb tbl arm1 arm2 pr db cr cptGraph
sets CPT POS
axioms
@axm1 partition(CPT, {fb}, {tbl}, {arm1}, {arm2}, {pr}, {db}, {cr})
@axm3 cptGraph ∈ CPT ↔ CPT
@axm4 cptGraph = {fb ↦ tbl, tbl ↦ arm1, arm1 ↦ pr, pr ↦ arm2, arm2 ↦ db, db ↦ cr}

```

Figure 19: Example of domain-specific modelling through context

composed in different ways to model different PC components and hence benefit from their reuse.

The third approach of modelling static variability through context switching allows to evaluate the scope of a product line and without doing proving effort upfront. This in another way of modelling component-based PC and suits the product line development approach as we can figure out the common base for different variants (topologies) of the PC, and the configuration or the variability is embedded in the context. This could be useful to foresee how a product line would evolve for a particular domain and later on this could be modelled in one of the two styles mentioned to build a database of reusable features.

## 6. ATM Case-Study

The auto teller machine (*ATM*) provides services to bank customers using their ATM cards issued by the bank. There are some basic services provided by an ATM such as cash withdrawal, view account balance and card activation related services. Other services can also be provided by ATMs which vary for different banks and ATM locations, e.g., mobile top up, cash deposit and emergency cash withdrawal etc.

We can build a product line of ATMs to manage variability and benefit from reuse while building various ATMs providing different features from a shared set of available features. Figure 20 shows a feature model for ATM product line. A different configuration of features will result in a variant of an ATM. We have modelled some ATM features in Event-B to see if existing tools and techniques are capable enough for our feature-oriented modelling framework in Event-B or whether we can find other patterns where these existing approaches fall short. Hence, we can propose ways to handle those situations and suggest any requirements for the tools and techniques to be built in the future to complement the feature-based development in Event-B. In this Section, we will discuss balance transfer and deposit features to model an ATM product and then modelling another variant of ATM product line by extending existing ATM product to provide cash withdrawal feature as well. Although some ATM requirements have previously been modelled in Event-B [46], we have used a different set of features and modelled in a different way to experiment with our reuse approach.

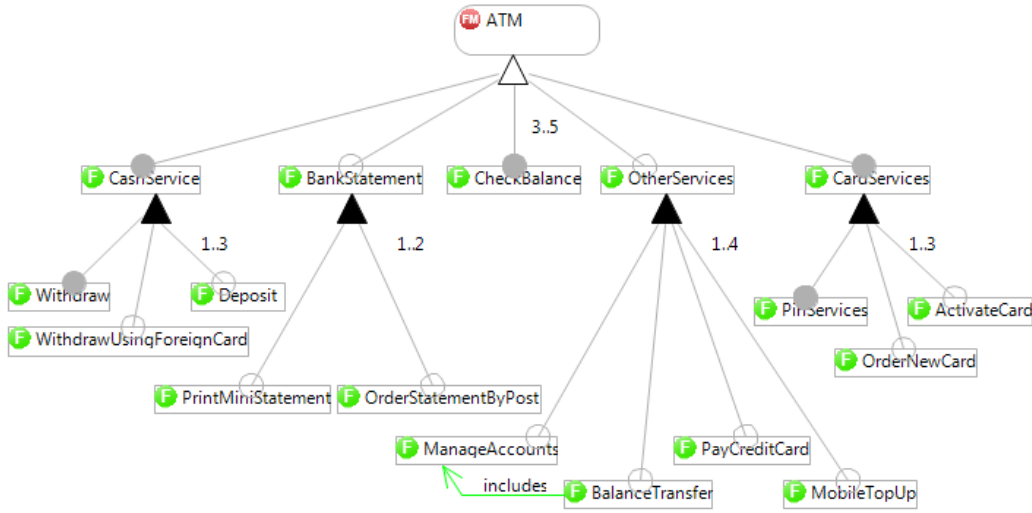


Figure 20: ATM feature model

### 6.1. Event-B Modelling of ATM Features

We started modelling the deposit and balance transfer (which we will simply call “transfer”) features independently and refined each of them to introduce all their requirements. At the abstract level, the deposit feature contains an event for deposit which increments the account balance (*bal*) by an amount. Similarly, the transfer feature contains an event for transferring balance that decrements the source account and increments the target amount by an amount (see Figure 1). Both features contain external events, i.e., deposit feature has external event *transfer* and transfer feature has external event *deposit*. These external events must not be refined along with the shared variable as a restriction of SVD. These features are then refined horizontally and vertically as discussed below.

The first refinement of the balance transfer feature refines the *transfer* event for a successful transfer of money and another event is introduced when the transfer fails due to the account balance being less than the transfer amount. The second level of refinement introduces a request and response mechanism between the ATM and the Bank. Here the ATM sends a balance transfer request to the bank, which responds after a successful or failed transfer event takes place and then the ATM displays the transfer status. The third level of refinement further refines the request and response mechanisms by partitioning the request event for sending and receiving the request and

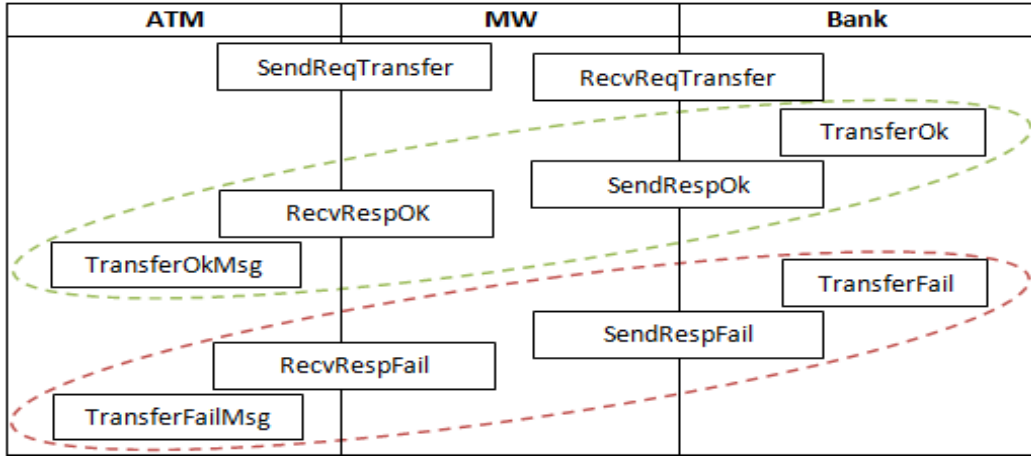


Figure 21: Balance Transfer fourth refinement model with events ready for architectural decomposition

similarly for the response event. The fourth refinement introduces the middleware (*MW*) between the ATM and the Bank. This allow us to make an architectural decomposition of the balance transfer feature into ATM, MW and the Bank where MW is used for communicating between the two. The recomposition of these (ATM, MW, Bank) would refine the feature being decomposed (fourth refinement).

Figure 21 shows the architecture of this refinement model including all the events and which component these would belong to after decomposition. It also shows the sequence of events. An ATM sends a balance transfer request through the MW which is received by the bank. The bank then sends a response for a successful or failed transfer through the middleware. The ATM finally displays the transfer status accordingly. We used SED here and the components synchronise using the shared-events. Each of these three components can be further refined.

Similarly, we refined the deposit feature resulting in three components, i.e., ATM, MW and the Bank. Figure 22 shows the development and composition structure for the deposit and balance transfer features of the ATM. In the figure, asterisk (\*) denotes a model with external events, and *bal* indicates the model's shared variable. We need external events in order to later compose these features using SVC after several refinement steps, to make sure the composition is correct by construction and hence not need reprov- ing. This means we need to make sure that the two features were a result

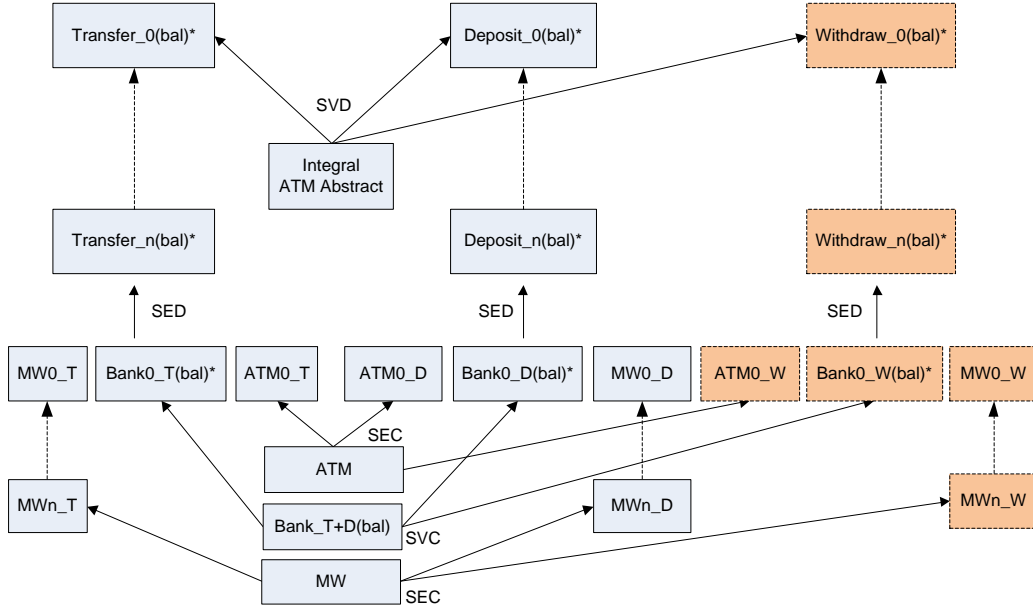


Figure 22: Refinement & (de)composition architecture for ATM features

of SVD of an integral abstract model. Figure 23 shows the SVD of integral abstract model of Figure 1 with external events and shared variable. This generated a tooling requirement to automatically generate external events for the features that we would like to compose using SVC and hence saving time to manually do so. Note that in this case study, the shared variable *bal* and its corresponding external events are localized in the Bank component.

Now that we have the same architectural decomposition (ATM, MW, Bank) for each feature, we would like to compose these models pairwise (i.e.,  $Bank_{T+D} = Bank_T + Bank_D$ , etc.) for implementation purposes. In general, the task would of course be more complex, involving more than two features. In our case, where the shared variable *bal* is localized into the two architectural Bank components, intuition suggests that these can be composed, with the composite Bank refining each component Bank. This is because each Bank's external events are exactly "cancelled out", or implemented, by the other Bank's actual events. This assertion remains to be proved in general for this pattern of mixed decomposition-recomposition.



<pre> machine Transfer_0 sees Context_Transfer0 variables bal // Shared variable, DO NOT REFINE invariants   theorem @typing_bal bal ∈ P(ACCOUNT × Z)   @compA_inv1 bal ∈ ACCOUNT ↔ N events   event INITIALISATION   then     @act1 bal = ∅   end   event transfer   any src_ac dest_ac am   where     @grd1 src_ac ∈ ACCOUNT     @grd2 dest_ac ∈ ACCOUNT     @grd3 am ∈ N     @grd4 src_ac ∈ dom(bal)     @grd5 dest_ac ∈ dom(bal)     @grd6 src_ac ≠ dest_ac     @grd7 am &lt; bal(src_ac)   then     @act1 bal = bal + {dest_ac ↦       (bal(dest_ac) + am), src_ac ↦(bal(src_ac) - am)}   end   event deposit // External event, DO NOT REFINE   any acc am   where     @grd1 acc ∈ ACCOUNT     @grd2 acc ∈ dom(bal)     @grd3 am ∈ N   then     @act1 bal(acc) = bal(acc) + am   end end </pre>	<pre> machine Deposit_0 sees Context_Deposit0 variables bal // Shared variable, DO NOT REFINE invariants   theorem @typing_bal bal ∈ P(ACCOUNT × Z)   @compA_inv1 bal ∈ ACCOUNT ↔ N events   event INITIALISATION   then     @act1 bal = ∅   end   event transfer // External event, DO NOT REFINE   any src_ac dest_ac am   where     @grd1 src_ac ∈ ACCOUNT     @grd2 dest_ac ∈ ACCOUNT     @grd3 am ∈ N     @grd4 src_ac ∈ dom(bal)     @grd5 dest_ac ∈ dom(bal)     @grd6 src_ac ≠ dest_ac     @grd7 am &lt; bal(src_ac)   then     @act1 bal = bal + {dest_ac ↦       (bal(dest_ac) + am), src_ac ↦(bal(src_ac) - am)}   end   event deposit   any acc am   where     @grd1 acc ∈ ACCOUNT     @grd2 acc ∈ dom(bal)     @grd3 am ∈ N   then     @act1 bal(acc) = bal(acc) + am   end end </pre>
---	---

Figure 23: ATM Integral Model Decomposed using SVD into Transfer & Deposit Features

### 6.2. Modelling ATM product by reusing existing features

After building an ATM with two features, we want another ATM product having a cash withdrawal feature as well (as shown by dotted lines in Figure 22). We elaborate the top-level integral model to include the withdrawal feature and decompose it into three components (i.e., deposit, transfer and withdrawal). Provided the new feature is *separable* - in the sense that in the SVD refinement the other two features remain unchanged - then all we have to do is refine the withdrawal component. This means that both the deposit and transfer features would now contain external event *withdraw* of withdrawal feature. Since the deposit and balance transfer components have already been proved, new POs will only be generated for the newly added external event acting on shared variable *bal*. These new POs will only be generated in the abstract models of deposit and transfer features no matter how many refinements exist because of the restriction of SVD that external extents and shared variables should not be refined. Hence, we will only have to discharge these small number of POs when reusing existing models. So, If we can have a tool for analysing and automatically generating external events in the existing components for the newly added components, then we could reduce the amount of POs needed to be discharged.

### 6.3. Evaluation

We have examined a specific pattern of mixed decomposition-recomposition - SVD followed by SED and then SVC/SEC in a single development. It appears possible to do this provided shared variables and their associated external events become localized in one of the shared-event components. At present, there is no SVC tool support which means that we have manually done this composition using our feature composition tool that does not guarantee correct composition without reproof. Should this pattern be validated theoretically, other architectural possibilities should emerge: e.g., an ATM-specific shared variable as well as a Bank-specific one in the same development. Interesting avenues of future work are indicated. We can generalise this pattern where the shared variables and their associated external events must be localised in exactly the same component in each of the feature developments, i.e., if a shared-variable 'a' goes into component1 and shared-variable 'b' goes into component2 in feature1 then these variables 'a' and 'b' should be localised to components 1 & 2 respectively in feature2 and so on. Figure 24 shows this general pattern. We have to use SVC while composing Nr2(a) and Pr2(a) as these two components have external events

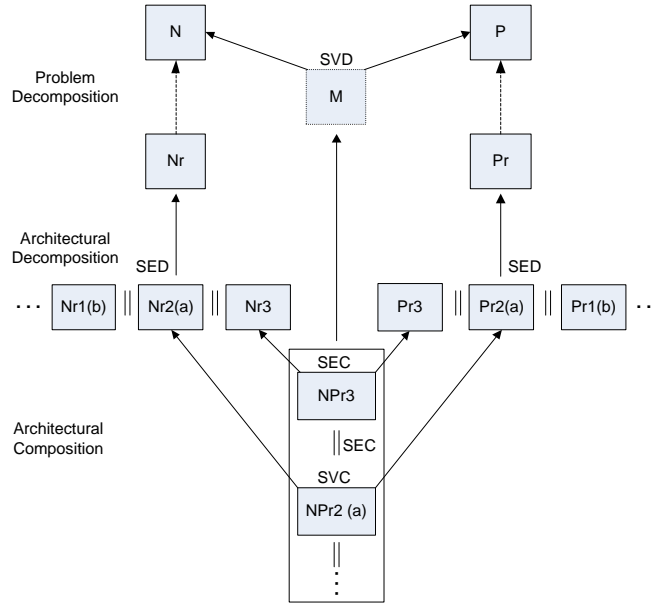


Figure 24: Feature-oriented Refinement & (De)Composition Pattern

and shared-variable ‘a’. We can compose  $Nr3$  and  $Pr3$  using SEC since both components are disjoint. This pattern needs to be experimented with more complex example case-studies.

## 7. Current Feature Modelling Tool Support for Event-B

Based on our feature modelling notation discussed in Section 4, we have developed a feature modelling tool to specify feature models of product lines and to configure the feature models to generate their instances. The tool is open source, developed in Eclipse using Java and integrated as a plug-in to the Rodin platform. It consists of two parts, i.e., feature model editor and configuration editor, discussed next.

### 7.1. Feature Model Editor

Our feature modelling tool includes a graphical feature model editor developed using GMF (Graphical Modelling Framework) [47]. This editor can be used to draw feature models for product lines in a free form tree structured hierarchy and uses our graphical notations of feature modelling. There is a three way validation mechanism provided by the editor to ensure that valid

feature models are drawn which could then be configured. The division is based on how this is implemented by the tool. Firstly, feature models should conform to the feature metamodel. Secondly, it does not let the user draw a feature model that *violates* any of the following validation cases:

- A feature model may not have cycles, i.e., a feature  $x$  having a child feature  $y$ , which is a parent feature of  $x$ .
- A feature may not include and exclude the same feature.
- An Event-B feature must be a leaf feature in the feature model tree and should not have further features or groups.

Thirdly, the editor validates the feature model when it is saved based on the following validation cases and warns the user if any of the cases is violated by the model. This differs from the above as it lets the user save an inconsistent/incomplete model to be completed later. Note that we have not yet explored automated feature model verification and the generated instance validation as proposed by Sun et al. [48].

- A feature  $x$  may not include features  $y$  and  $z$ , which exclude each other. If this scenario is present in the feature model, it is not possible to generate a valid instance of the feature model.
- A feature may not exclude any of its ancestor features.
- A feature may not be unreachable during the configuration or may not be selected in any valid configuration. An example would be a group of two features with group cardinality ‘1..1’ and one of the features is mandatory. In this case, the user will never be able to have the optional feature selected in a valid configuration as doing that will violate the group cardinality due to the other mandatory feature.
- A feature model may not have orphan features.

A metamodel provides a language or notations for building the models. The feature models built using the feature model editor are transformed into their equivalent EMF metamodels at run-time and for each product family. This model to metamodel transformation is needed in order to instantiate

the feature models and this forces the instances to conform to their meta-model. So, a feature model is an instance of the metamodel defining our feature modelling notations and it also serves as a metamodel for any of its instances. The transformation is done using Epsilon Transformation Language (ETL) [49]. After transforming feature models into metamodels, for different product lines, these are then used as an input to the feature configurator for instantiation discussed next. This model transformation process is internal to the tool and not visible to the modeller. This is shown in Figure 25 as part of the feature modelling tool architecture.

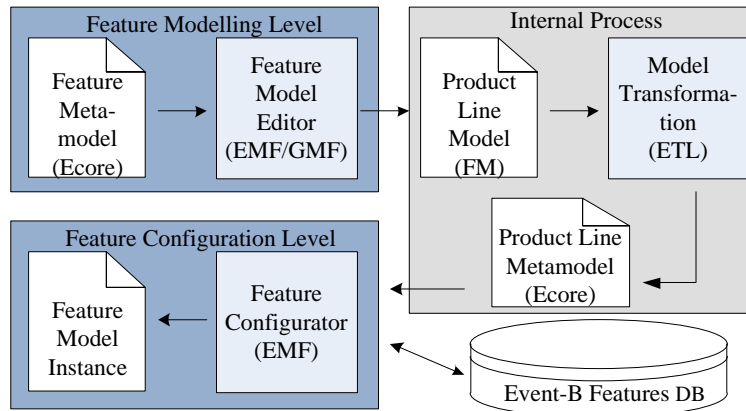


Figure 25: Feature Modelling Tool Architecture

## 7.2. Feature Configurator

The feature configurator is a collapsible tree-structured editor (screenshot in Figure 26) that allows the user to configure a feature model by selecting features that they want to include in a particular product. The editor then highlights any conflicts and provides options for resolving these conflicts automatically/interactively. The selected features (Event-B models) are then composed into a single composite Event-B model. The configurator enforces some of the constraints provided in the feature model. It automatically selects the mandatory features and highlights any violation of cardinality constraints. Whenever a feature is selected, the tool automatically selects/deselects features specified using includes/excludes constraints. The configurator also shows Event-B elements of machines and contexts (e.g., variables, events etc.) for the Event-B features which is different to other feature modelling tools (e.g., FeatureHouse [34]).



Figure 26: Feature Configurator Screenshot

At the moment it detects naming conflicts within Event-B models (e.g. variables or events having same name). It provides ways to automatically resolve these name clashes either by making them disjoint through refactoring or by simply deselecting repeating entries in multiple features. It also helps the user in automatically selecting any dependencies, for example, if an event is selected, it can then select the related variables and their invariants to build the correct model.

Once all the desired features are selected and conflicts are resolved, these are composed to generate a composite Event-B feature. All the machines are merged into a machine and all the contexts are merged into a context. This can be called as a structured cut-and-paste composition. The editor also enables the user to merge multiple events into a single event. This concatenates the actions and conjoins the guards to maintain invariant preservation. This composition of Event-B features into a composite feature is required in order for us to reason about the complete model of the generated instance, e.g., using animation, theorem provers etc.

The feature modelling tool was quite useful while modelling our case-study examples as we had to compose models in different ways to experiment with our modelling approaches and to use existing decomposition techniques, specifically SVD, since there is no tool support for shared-variable composition. The tool will be extended further to include the requirements we have

generated during this research, presented in the next Section.

## 8. Guidelines for Feature Modelling in Event-B & Tooling Requirements

Our case-study experiments reveal some patterns of modelling that could be used as a guideline for specifying SPLs in Event-B. These patterns make use of Event-B's existing tools and techniques. If we could develop further tool support as discussed later, then this whole process could be automated saving lot of user time and effort. At the moment, the lack of proper tool support meant that we had to do lot of things manually, e.g., trivial refactoring and instantiation of a generic model into two disjoint models. In the following we present some guidelines for formal feature-based modelling in Event-B which generalise our approach of modelling the two case-studies discussed earlier.

By grouping a list of requirements into small features as we did for controller-based PC modelling, making each feature as a stand alone generic development, we can reuse and specialise these features in different configurations to build variants of a PL. This could also be done by first modelling the system up to a few refinements as a whole and then trying to figure out how the system could be decomposed in terms of requirements/functionality. In this way, the modeller will be able to identify earlier on what sort of information would be required in order to compose these features so that the generic features could be modelled with that input/output connector information as in component-based software development. This would serve as a domain-analysis step to predict reusability of features before actual modelling them as reusable features.

The decision to use top-down usual modelling approach of Event-B, bottom-up approach of SPLE or the middle-in approach of the ATM example would vary for different domains. The domain expert or system modeller would be in a better position to judge which approach would suit more for a particular system. We have used both top-down and bottom-up approach in PC case-study and middle-in approach in the ATM case-study.

Based on experience of the two case-studies modelled in different ways, if the user is planning to model a PL and would need to build many variants of the PL in future, then its probably useful to take the middle-in approach and make the features as generic as possible. If the abstract models of the features being modelled initially could be considered as a result of SVD, then

their recomposition at any later refinement will be correct by construction. These features later on could also be decomposed using SED for architectural decomposition and that would not violate the correctness promise of SVD if the same pattern of modelling is followed as in ATM example, i.e., the shared-variables and their corresponding external events are localised in the same components across the different features as shown in the ATM modelling pattern (see Figure 24 ). If this pattern is not followed and feature are modelled as generic reusable models independently and later on composed and additional composition time information is supplied (discussed as Feature Composition earlier), then the user need to reprove the composite model. This could only be helpful if we have a mechanism for proof reuse that would reduce the reproofing effort.

Also, if the features being developed can use generic placeholders which could be filled during composition, then that would reduce effort of reproofing by simple refactoring. The generic placeholder could be specified, e.g., X...X at the moment as Event-B tool will not raise an error, so the composition tool would force the user to fill these placeholder at the composition time.

### *8.1. Future Tool Support Requirements*

In order to make use of the feature modelling approach suggested, the following tool support will be needed to facilitate the modeller.

- **Refactoring Support:** During the composition of features, allow the user to add new variables, invariants and guards. These additional variables, guards and invariants should be static-checked for errors and instantly reported to the user. The user should be able to guide the refactoring, i.e., when refactoring a machine, allow the user to suggest prefixes. This should also support instantiation of generic refinement chains by refactoring. The refactored elements should also be static-checked for any errors. The tool should provide support for generic placeholders. For example, an element name (e.g., vars, events etc.) must be provided by the user at the time of composition when a model contains any placeholder (e.g., X...X).
- **Feature Modelling & Topology Tool:** The tool should allow the user to specify cardinality for feature instantiation (cloning), so that a feature could be replicated a number of times in a particular



instance. For example, in PC feature model, the feature ‘press’ could have a cardinality ‘1..n’, which means any instance of the PC could have at least one and up to ‘n’ presses. This must be then supported in the configuration editor and would also need user guided refactoring support as discussed above.

The user should be able to annotate the feature model, i.e., to specify composition rules, more complex constraints and other information that they think might be helpful during the configuration process. For example, the composition rule and rationale provided in the example feature model of Figure 4. This would help the modeller during the selection of a particular feature while instantiating a product line.

After drawing a feature model, allow the user to draw a topology graph based on that feature model to visualize the feature model instance and how different features will be connected to each other in the graph, as discussed in PC. The features are drawn as nodes and the user should be able to select the transitions between these nodes by selecting an event from the available features in the feature machine. This may be also be achieved by adding extra constraints to the feature model that specifies connection between components. So the user specifies which component is connected to which other components and this information could be used to draw transitions between different nodes. At the moment, the feature modelling tool only supports two constraints, i.e., includes and excludes.

- **SVC Tool:** There should be a tool for composing models using the SVC style. This may also include support to automatically generate external events and shared-variables so that the models could be composed using shared-variable composition style. Also, it would be helpful for the modelled to have a single tool that could be used to compose models using any of the composition styles, i.e., SEC, SVC, fusion and feature composition.
- **Composition Replay Support:** All the composition decisions must be saved so that the user can come back later to modify or redo the composition without losing additions or refactoring etc. It would be useful to have a script of all the configuration decisions so that the user can replay the configuration by editing the script (may be doing some refactoring etc.). This will be like an audit trail of all the actions

performed during the configuration. This may also include graphical representation of the actions performed during the composition to visualise more complex multi-step compositions.

## 9. Related Work

As far we know, there is no tool support for feature-oriented modelling in Event-B that can be used to specify product lines and reuse existing features to model different variants of a SPL. Formal SPL development has been researched for quite a while now [50], but there is no standard tool and technique for doing so. HATS [51] provides a methodology for applying formal methods at different stages of SPL development cycle. This seems to be a promising work to bring together the domains of formal methods and SPL. Lau et al. [52] proposed component-based verification approach which allows the composition of existing verified components and support proof reuse. This is different to our approach because an Event-B component is not a single model but a chain of refinements. We need to compose models at different refinement levels and also preserve the refinement relationship between the abstract and concrete composite models. Mannion [53] has used first-order logic for PL model validation by representing requirements and their relationship as a logical expression. Some work on formal feature modelling has been done by Sun et al. [48] which provides automated feature model verification and the validation of generated PL instances.

There are a number of feature modelling tools that can be used for modelling SPLs. FeaturePlugin [24] was developed to support the ‘cardinality-based feature modelling’ as discussed above. This is an Eclipse plug-in providing EMF based tree structured editor for building feature models, their specialization and configuration. CaptainFeature [54] is another similar tool and the major difference to the above is in the rendering style for feature models. XFeature [55] is another Eclipse-based tool that requires the feature models to be expressed in XML using the XML editor. The editor is supported by an XML Schema representing the metamodel of feature modelling notations. Pure::Variants [56] is a commercial tool for SPL development which does not use cardinality based notations and provides constraint-oriented configuration through a Prolog-based constraint solver. FeatureHouse [34] is another tool that allows the composition of artefacts and supports various languages with option to include more. Our experiment to use Event-B language in FeatureHouse did not seem feasible because integration of FeatureHouse in the

Rodin was not straightforward and we also found it less usable for Event-B modellers in comparison to our proposed feature modelling tool.

Some work has been done by Sorge et al. [36] which deals with invariant POs for composing features. This does not support feature refinement and event fusion which is required to complement our feature modelling framework. Verification of a PL variant through proof composition [57] has been suggested where the proofs for the composite model (PL variant) are generated by composing proofs of selected features. These proofs can then be checked by a proof assistant to validate the PL member.

## 10. Conclusion & Future Work

We have given an overview of our research on feature-oriented reuse approach using Event-B and to what extent this is possible using existing Event-B's tools and techniques. Although, it seems possible to model SPLs using existing techniques with some restrictions, we still need more tool support to gain full benefits of formal SPL engineering. Our case-study examples have revealed some patterns of modelling that could be used as a guideline for SPLE using existing (de)composition approaches and to minimise the proof effort while reusing already proven Event-B features. We have also generated some tooling requirements that could facilitate the user for feature-modelling in Event-B. Some interesting research questions came up during this work that would need to be addressed in the future. For example, to what extent our suggested modelling pattern could be generalised. This would require further work where we could apply these guidelines and pattern on more complex case-studies.

We also need to do some research on a proof reuse approach that would allow more complex and freer form of composition with minimal proof burden for the modellers. We would like to see if the proof composition approach of [57] can be applied to Event-B using the case-studies presented. The issue of feature interactions [58] also needs to be explored for feature-oriented modelling in Event-B.

## 11. Acknowledgement

We are grateful to Todor Spasov (MSc Student), Chris Franklin and Nikola Milikic (University of Southampton Interns), for their contribution towards the development of prototype tools to support our research. This

work is partly supported by the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) [www.deploy-project.eu](http://www.deploy-project.eu).

## References

- [1] J. R. Abrial, *The B-book: assigning programs to meanings*, Cambridge University Press, New York, NY, USA, 1996.
- [2] R. W. Butler, *What is formal methods?*, Technical Report, NASA, 2001.
- [3] E. M. Clarke, J. M. Wing, *Formal methods: state of the art and future directions*, *ACM Comput. Surv.* 28 (1996) 626–643.
- [4] J. P. Bowen, M. G. Hinchey, *Ten commandments of formal methods ...ten years later*, *Computer* 39 (2006) 40–48.
- [5] J. Woodcock, P. G. Larsen, J. Bicarregui, J. Fitzgerald, *Formal methods: Practice and experience*, *ACM Comput. Surv.* 41 (2009) 19:1–19:36.
- [6] J. P. Bowen, M. G. Hinchey, *The use of industrial-strength formal methods*, in: *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*, IEEE Computer Society, Washington, DC, USA, 1997, pp. 332–337.
- [7] J. C. Bicarregui, J. S. Fitzgerald, P. G. Larsen, J. C. Woodcock, *Industrial practice in formal methods: A review*, in: *Proceedings of the 2nd World Congress on Formal Methods, FM '09*, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 810–813.
- [8] J.-R. Abrial, *Formal methods in industry: achievements, problems, future*, in: *ICSE '06: Proceedings of the 28th international conference on Software engineering*, ACM, New York, NY, USA, 2006, pp. 761–768.
- [9] J. P. Bowen, *Formal Specification and Documentation using Z: A Case Study Approach*, International Thomson Computer Press, 2003.
- [10] D. John, *The VDM-SL Reference Guide*, Routledge, UK, 1991.
- [11] R. Back, J. von Wright, *Trace refinement of action systems*, in: B. Jonsson, J. Parrow (Eds.), *Concurrency Theory*, volume 836 of *LNCS*, 1994, pp. 367–384.

- [12] J.-R. Abrial, *Modeling in Event-B: System and Software Engineering*, Cambridge University Press, first edition, 2010.
- [13] C. A. R. Hoare, *Communicating sequential processes*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.
- [14] J. Bergstra, J. Klop, Algebra of communicating processes with abstractions, in: *Theoretical Computer Science*, volume 33, Netherlands, pp. 77–121.
- [15] R. Milner, *Communication and concurrency*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989.
- [16] Eclipse - An open development platform, <http://www.eclipse.org>, August 2011. October 2009.
- [17] J.-R. Abrial, M. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, L. Voisin, Rodin: An open toolset for modelling and reasoning in event-b, *International Journal on Software Tools for Technology Transfer (STTT)* 12 (2010) 447–466.
- [18] P. Clements, L. Northrop, *Software Product Lines : Practices and Patterns*, Addison-Wesley Professional, 2001.
- [19] K. Czarnecki, M. Antkiewicz, C. H. P. Kim, S. Lau, K. Pietroszek, Model-driven software product lines, in: *OOPSLA '05*, ACM, NY, USA, 2005, pp. 126–127.
- [20] A. Classen, P. Heymans, P.-Y. Schobbens, What's in a feature: a requirements engineering perspective, in: *Proceedings of FASE'08/ETAPS'08*, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 16–30.
- [21] D. Batory, Feature models, grammars, and propositional formulas, in: *SPLC '05: Proceedings of the 9th International Software Product Line Conference*, Springer, 2005, pp. 7–20.
- [22] C. Snook, M. Poppleton, I. Johnson, Rigorous engineering of product-line requirements: a case study in failure management, *IST 50* (2008) 112–129.

- [23] M. Poppleton, B. Fischer, C. Franklin, A. Gondal, C. Snook, J. Sorge, Towards Reuse with “Feature-Oriented Event-B”, McGPLe: Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, Nashville, TN, 2008.
- [24] M. Antkiewicz, K. Czarnecki, Featureplugin: feature modeling plug-in for eclipse, in: eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange, ACM, New York, NY, USA, 2004, pp. 67–72.
- [25] R. Silva, M. Butler, Supporting Reuse of Event-B Developments through Generic Instantiation, in: ICFEM, pp. 466–484.
- [26] J.-R. Abrial, S. Hallerstede, Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B, *Fundam. Inf.* 77 (2007) 1–28.
- [27] M. Butler, Synchronisation-based Decomposition for Event-B, in: RODIN Deliverable D19 Intermediate report on methodology.
- [28] R. Silva, C. Pascal, T. S. Hoang, M. Butler, Decomposition tool for event-b, in: Workshop on Tool Building in Formal Methods - ABZ Conference.
- [29] M. Poppleton, The composition of event-b models, in: ABZ2008: Int. Conference on ASM, B and Z, volume 5238, Springer LNCS, 2008, pp. 209–222.
- [30] K. Kang, S. Cohen, J. Hess, W. Nowak, S. Peterson, Feature-Oriented Domain Analysis (FODA) Feasibility Study., Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1990.
- [31] K. Lee, K. C. Kang, J. Lee, Concepts and guidelines of feature modeling for product line software engineering, in: ICSR-7, Springer-Verlag, UK, 2002, pp. 62–77.
- [32] K. Czarnecki, S. Helsen, U. Eisenecker, Staged configuration through specialization and multilevel configuration of feature models, *Software Process: Improvement and Practice* 10 (2005) 143–169.

- [33] J. Bosch, Design and use of software architectures: adopting and evolving a product-line approach, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 2000.
- [34] S. Apel, C. Kastner, C. Lengauer, FEATUREHOUSE: Language-independent, automated software composition, ICSE '09, USA, pp. 221–231.
- [35] A. Gondal, M. Poppleton, M. Butler, C. Snook, Feature-Oriented Modelling Using Event-B, in: SETP-10, Orlando, FL, USA.
- [36] J. Sorge, M. Poppleton, M. Butler, A Basis for Feature-oriented Modelling in Event-B, in: ABZ2010.
- [37] A. Gondal, M. Poppleton, C. Snook, Feature composition - towards product lines of Event-B models, in: MDPLE'09, CTIT Workshop Proceedings, 2009.
- [38] T. Lindner, Task description, in: C. Lewerentz, T. Lindner (Eds.), Formal Development of Reactive Systems, volume 891 of *Lecture Notes in Computer Science*, Springer, 1995.
- [39] C. Lewerentz, T. Lindner, Case study 'production cell': A comparative study in formal specification and verification, in: M. Broy, S. Jähnichen (Eds.), Tech. Rep., Forschungszentrum Informatik, volume 1009 of *LNCS*, LNCS 891, Springer-Verlag, 1994, pp. 1–54.
- [40] M. Ouimet, K. Lundqvist, Modeling the Production Cell System in the TASM Language, Technical Report, Massachusetts Institute of Technology, 2007.
- [41] D. Paun, M. Chechik, B. Biechelle, Production cell revisited, 1998.
- [42] H. Dierks, The production cell: A verified real-time system, in: B. Jonsson, J. Parrow (Eds.), Formal Techniques in Real-Time and Fault-Tolerant Systems, volume 1135 of *LNCS*, Springer, 1996, pp. 208–227.
- [43] E. Sekerinski, Program Development By Refinement, Springer-Verlag, 1998.
- [44] M. Butler, Towards a cookbook for modelling and refinement of control problems (2009).

- [45] R. Silva, Renaming framework, 2011. [http://wiki.eventb.org/index.php/Refactoring\\_Framework](http://wiki.eventb.org/index.php/Refactoring_Framework).
- [46] M. Y. Said, Methodology of refinement and decomposition in UML-B, Ph.D. thesis, School of Electronics and Computer Science, University of Southampton, 2010.
- [47] Online, Graphical Modeling Framework (GMF), Project Website: <http://www.eclipse.org/modeling/gmf/>, 2011.
- [48] J. Sun, H. Zhang, H. Wang, Formal semantics and verification for feature modeling, in: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, Washington, DC, USA, 2005, pp. 303–312.
- [49] D. Kolovos, R. Paige, F. Polack, The epsilon transformation language, 2008, pp. 46–60.
- [50] T. Kishi, N. Noda, Formal verification and software product lines, Commun. ACM 49 (2006) 73–77.
- [51] D. Clarke, N. Diakov, R. Hähnle, E. B. Johnsen, G. Puebla, B. Weitzel, P. Y. H. Wong, HATS-a formal software product line engineering methodology, in: Proc. Intl. Workshop on Formal Methods in SPL Engineering, Southa Korea.
- [52] K.-K. Lau, Z. Wang, A. Wang, M. Gu, A component-based approach to verified software: What, why, how and what next?, in: X. Chen, Z. Liu, M. Reed (Eds.), Proc. 1st Asian Working Conference on Verified Software, pp. 225–229. UNU-IIST Report No. 347.
- [53] M. Mannion, Using first-order logic for product line model validation, in: Proceedings of the Second International Conference on Software Product Lines, SPLC 2, Springer-Verlag, London, UK, UK, 2002, pp. 176–187.
- [54] C. E. T. Bednasch, M. Lang., CaptainFeature, <https://sourceforge.net/projects/captainfeature/>, 2002-2004.
- [55] A. Pasetti, O. Rohlik, Concept for the XFeature Tool, Technical Report 1.3, P&P Software GmbH, 2008.



- [56] P.-S. GmbH, Variant Management with pure::variants, Technical Report, 2006.
- [57] T. Thum, I. Schaefer, M. Kuhlemann, S. Apel, Proof composition for deductive verification of software product lines, IEEE International Conference on Software Testing Verification and Validation 0 (2011) 270–277.
- [58] M. Calder, M. Kolberg, B. Evan H. Magill, Feature interaction: a critical review and considered forecast, Computer Networks 41 (2003) 115–141.