Feature-Oriented Modelling Using Event-B

Ali Gondal, Michael Poppleton, Michael Butler, Colin Snook
School of Electronics and Computer Science
University of Southampton, Southampton, SO17 1BJ, UK
Email: {aag07r, mrp, mjb, cfs}@ecs.soton.ac.uk

***Abstract*—Event-B is a formal method for specification and verification of reactive systems. Its Rodin toolkit provides comprehensive support for modelling, refinement and analysis using theorem proving, animation and model checking. There has always been a need to reuse existing models and their associated proofs when modelling related systems to save time and effort. Software product lines (SPLs) focus on the problem of reuse by providing ways to build software products having commonalities and managing variations within products of the same family. Feature modelling is a well know technique to manage variability and configure products within the SPLs. We have combined the two approaches to formally specify SPLs using Event-B. This will contribute the concept of formalism to SPLs and reusability to Event-B. Existing feature modelling notations were adapted and extended to include refinement mechanism of Event-B. An Eclipse-based graphical feature modelling tool has been developed as a plug-in to the Rodin platform. We have modelled the 'production cell' case-study in Event-B, an industrial metal processing plant, which has previously been specified in a number of formalisms. We have also highlighted future directions based on our experience with this framework so far.**

I. Introduction

Formal Methods provide mathematically based languages, tools and techniques for specifying and verifying systems during construction. They allow identification of inconsistencies, ambiguities and defects earlier in the software development life-cycle and reduce the need for unit and integration testing [1]. Successful application of formal methods can be seen in aerospace, transportation, defence and medical sectors [1], [2]. Improvements in formal specification languages, verification techniques and robust tools are ongoing, in particular, by the DEPLOY[1] and RODIN[2] projects, including industrial partners such as Bosch, SAP, Siemens and Space Systems Finland. These projects are developing tools, as well as strengthening the theoretical base, for the formal specification language Event-B [3].

Event-B [3] is a formal specification language, a successor of Abrial's classical B [4]. It was developed as part of the RODIN and earlier EU projects. The DEPLOY project, along with industrial partners, is currently focused on deploying this work into the industry. Event-B is based on set theory and first-order logic and allows the specification and verification of reactive systems. This is supported by integrated Rodin toolkit comprising editors, theorem provers, animators and model checkers.

[1]DEPLOY - Industrial deployment of system engineering methods EU Project IST-214158. http://www.deploy-project.eu

[2]RODIN - Rigorous Open Development Environment for Open Systems: EU Project IST-511599. http://rodin.cs.ncl.ac.uk

A Software Product Line (SPL) refers to a set of related products built from a shared set of resources having a common base [5]. Further members of the product line are built by reusing existing core components and adding extra functionality. SPLs provide the benefits of reusability in reducing the time to market, lower costs and reduce effort involved in product development. Feature modelling [6] is a well known technique for building SPLs. It has been used in many domains, product line projects and organizations where a feature (usually written in a programming language) is the unit of reuse, specialization and composition. It provides ways to manage variabilites and commonalities within a product line. Feature modelling was suggested as part of the Feature-Oriented Domain Analysis (FODA) [7] in the early 90's and several tools have been developed to support feature modelling for SPL engineering [6].

Our objective is to combine feature modelling with Event-B to formally specify software product lines, in order for SPLs to benefit from automated verification that formal methods provide. As a lot of time and effort is involved in specifying and verifying formal models, this can certainly help in reusing existing specifications and already proven models when building similar system. This can be encouraging for SPL community to use formal methods as we have adapted and extended existing feature modelling notations. Our earlier work on feature composition ([8], [9]) is now part of the formal product line development framework presented here. This paper highlights our contribution of extending existing feature modelling notations and the development of a feature modelling tool which can be used to build feature models of product lines with variability and constraints embedded in it. These feature models can then be configured to build instances of the product lines. A number of interesting issues came up during the development and then later on while applying it to the example case-study discussed later.

Section 2 gives a technical introduction of Event-B language. Feature modelling is discussed in Section 3 followed by the tool development discussion in Section 4. Section 5 discusses our experience on the application of case-study example to the tool. Related work and conclusion/future work is given in Sections 6 and 7 respectively.

II. Event-B Language

An Event-B model consists of a *machine* and multiple *contexts*. The machine specifies behavioural or dynamic part of the system where as the context contains static data which includes *sets*, *constants*, *axioms* and *theorems*. The set defines types where as the axioms give properties of the constants such as typing etc. Theorems must be proved to follow from axioms. The machine includes context(s) using the *sees* clause. The *variables* provide state space of the machine and their typing is given as *invariants* which are predicates and also specify correctness properties that must always hold. The state transition mechanism is accomplished through *events* which modify the variables. An event can have conditions known as event *guards* which must be true in order for the event to take place. It can also have *parameters* (also called as local variables). The machine variables are initialized using a special event called *INITIALIZATION* which is unguarded. An event has the following syntax:

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

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. This is checked by the tool using proof obligations (POs). POs are the verification conditions automatically generated by the tool and then discharged using theorem provers to verify correctness of the model.

Refinement is at the core of Event-B modelling where we start by specifying the system at an abstract level and gradually add further details in each refinement step until the concrete model is achieved. Refinement is usually considered to reduce non-determinism and each of the refinement steps must be proved to be the correct refinement of the abstract model by discharging the refinement POs. Typically, we classify the refinement into horizontal and vertical refinements. In horizontal refinement, we add more details to the abstract model to include further requirements where as in vertical refinement the focus is on the design decisions. Refinement can also be categorized as *event* and *data* refinements. In case of event refinement, when a new event is added in the refined model, it is said to refine *skip* which means it can not change any of the abstract state variables. New variables are related to the abstract ones using 'gluing invariants'. Both types of refinement can take place in a single refinement step.

III. Software Product Lines (SPLs)

A software product line is a set of related products built from a shared set of resources while having significant variability to meet the user requirements. Variability can be a particular functionality that differentiates products within the product line. The major advantage of SPLs approach is that of reusing existing artefacts to build similar systems with slight variations. The benefits of reusability are obvious when applied in a systematic way. These include less time to market in order to compete with the fast paced competitors, increased productivity and lower development costs by reusing already built components. It also improves on quality as existing components repeatedly go through testing and other v&v techniques. There are obviously some issues while reusing existing components such as integration problems but these can be overcome by carefully planning and designing the systems and building the components that can be reused in the future and that is where the concept of SPLs play its part. SPLs have been practiced in the industry for a couple of decades now and some success have been reported [5].

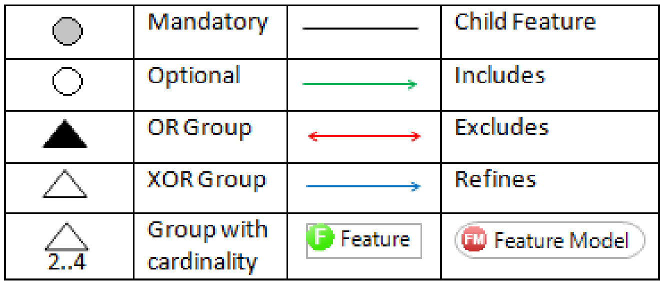
There are a number of approaches for specifying product lines. These include use-case driven and feature-based approaches. The use-case approach is more suitable for SPLs developers as it includes implementation level details and the feature-based approach is more suitable for non-development stakeholders where features are generally black-box objects and lower level technical details are abstracted [10]. We have selected the feature-based approach widely known as "feature modelling", as it seems more practical to combine with our formal methods domain.

A. Feature Modelling in Event-B

Feature modelling provides means to organize features and configure them in order to build products of a product line. The *feature* has been defined as "a logical unit of behaviour specified by a set of functional and non-functional requirements"[11] and usually referred as a property of the system that is of some value to the stakeholders. We define *feature* as an Event-B model which consists of a machine and multiple contexts. We have adapted the cardinality-based feature modelling notations[12] and extended it to add the refinement mechanism required for Event-B modelling. The reason for doing so is that the existing tools are not flexible enough to give Event-B semantics to the features and it is also hard to integrate with the Rodin platform. This can also be referred as domain-specific feature modelling.

Feature models are used to specify a product line and are represented using tree-structured feature diagrams. These include variability among the product line members and the ways in which these feature models can be instantiated to generate various products.

The graphical notations used in our feature modelling framework are given in Figure 1. A feature model consists of a tree structured feature diagram which has a root feature that gives it a name and can have many features. The filled circle on a feature shows that it is mandatory feature and optional otherwise. The features with a triangle attached represent group features which are containers for other features and specify any constraints on the 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. Non-filled triangle means alternative or the cardinality stated with it and the filled triangle means OR, i.e. the cardinality '1..k', where k is the number of features in that group. There are four types of connections that can be used to connect various model elements: features, includes, excludes and refines. 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. Our major extension to existing notations is the refinement concept of Event-B. A feature can be refined by multiple features. This implicitly puts a constraint on the selection of features in the refinement chain as only one of the features in the chain can be present in the generated feature instance. Any feature in the tree (except root), which does not have a features connection below it, is actually mapped to an Event-B feature during configuration, i.e. leaf level features or refined features (even non-leaf).



●	Mandatory	───	Child Feature
○	Optional	→	Includes
▲	OR Group	↔	Excludes
△	XOR Group	→	Refines
△ 2..4	Group with cardinality	F Feature	FM Feature Model

Fig. 1. Feature Modelling Graphical Notations

Figure 2 shows an example feature model drawn using our feature modelling notations. The root feature PC has a group of six features with cardinality “4..6” which means an instance must select at least four of the group features. The features *table* and *depositBelt* of the PC group are optional and rest of the group features are mandatory having filled circles. The feature *crane* includes *depositBelt* which means an instance of the feature model having *crane* feature must have *depositBelt*. Where as, using the feature *advCrane* (a refinement of *crane*) excludes *depositBelt*, which means both of these can not be present in a particular variant of PC derived from this feature model.

We developed an EMF[3] [13] metamodel based on these feature modelling notations as shown in Fig. 3. A *Feature-Model*, representing the system being modelled, has a name and may contain features or feature groups. A *Feature* also has a name and may have constraints. It can also contain further features or groups. Similarly, a *Group* may have features and any constraints as required. This metamodel has been developed with a view to possible future extensions to our feature modelling tool which have not been included in the current version such as annotating the feature models with composition rules and other constraints.

Feature modelling in Event-B can be used in two ways. Firstly, when the Event-B specification already exists and we want to use feature modelling to add variability and to allow

[3]Eclipse Modelling Framework (EMF), a part of Model Driven Architecture (MDA), is a Java based framework which provides facility for designing and implementing structured models

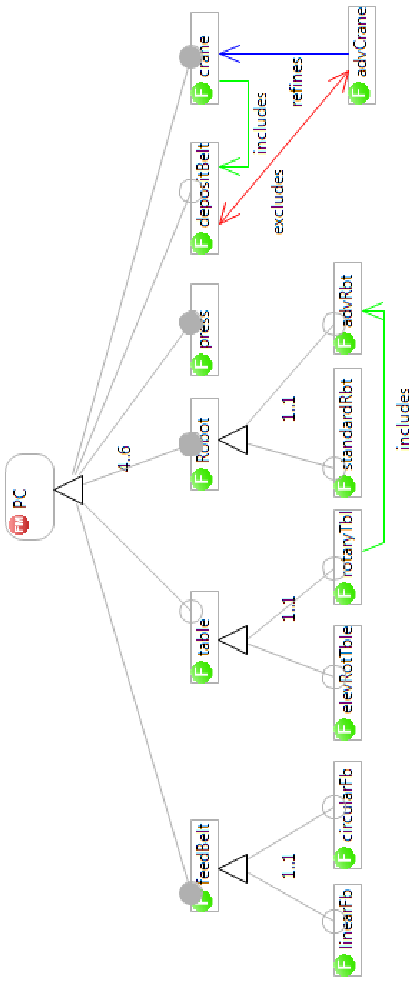


Fig. 2. Feature Model Example

users to configure these features in different ways to build similar systems. Secondly, where we start system development by building feature model of the system and then write Event-B specification afterwards. The later approach can also be used as a design activity during the system development and allows user to design the model in a way to be more reuse-oriented. It also helps in visualizing the refinement tree for the Event-B features to be specified.

1) Feature Model Validation: Model validation is an important issue that must be addressed in any model-driven development. Hence, we need to provide a way to make sure that our models are valid. By valid we mean that they conform to certain defined criteria or rules. In our feature modelling framework, the feature models must conform to the metamodel (which defines our feature modelling notations) and any constraints given in the metamodel must be satisfied. Also, there are other constraints that can not be expressed in a simple metamodel and requires the use of a constraints language (this will be considered in the future if needed based on the experience with the current tool). Following are the properties that must be satisfied in order to build consistent

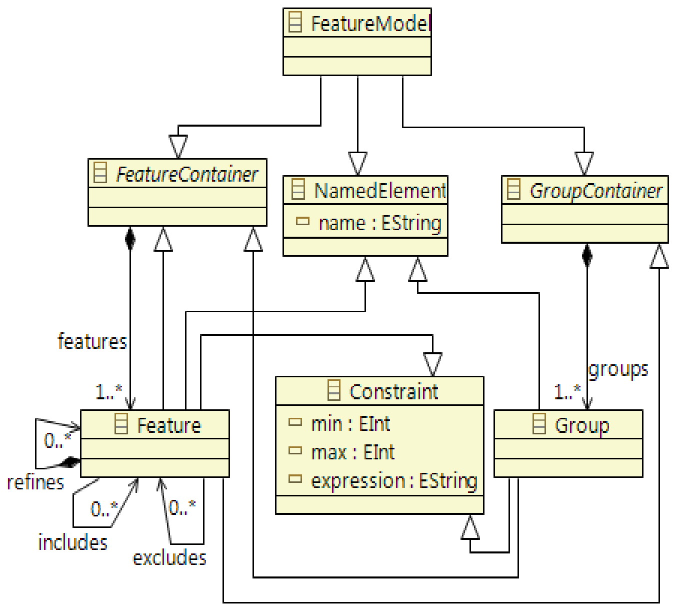


Fig. 3. Feature Modelling Metamodel

and correct feature models. To be more precise, a correct feature model conforms to its metamodel, does not violate any constraints and can be instantiated.

- A feature model may not have cycles, i.e. a feature x has a child feature y, which is a parent feature of x.
- A feature may not include and exclude the same feature.
- A feature refined by other feature should not have further features or groups as it represents an Event-B feature that can not have further features. It can only have further refinement features.
- 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. All the features must be connected to the feature tree in order for these features to be configured as part of the feature model.

IV. Tool Support

We have developed a feature modelling tool (FMT)[4] 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. Figure 4 shows architecture of the tool. It consists of a graphical feature model editor, a model transformation module and a feature configurator, discussed below.

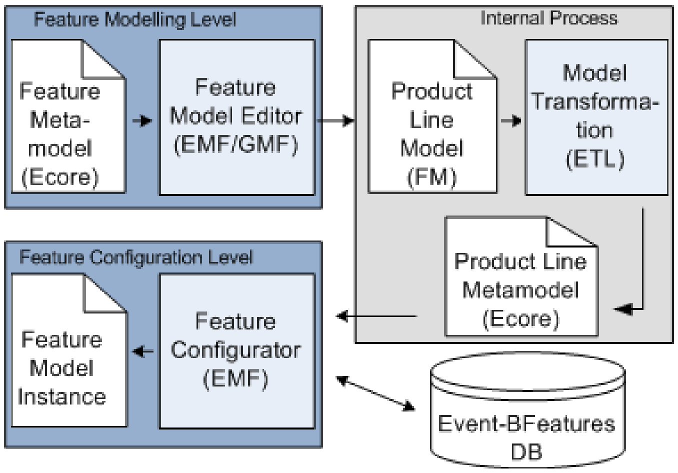

Feature Modelling Level
Feature Meta-model (Ecore)
Feature Model Editor (EMF/GMF)
Internal Process
Product Line Model (FM)
Model Transforma-tion (ETL)
Product Line Metamodel (Ecore)
Feature Configuration Level
Feature Model Instance
Feature Configurator (EMF)
Event-BFeatures DB

Fig. 4. Feature Modelling Tool Architecture

A. Feature Model Editor

Our feature modelling tool includes a graphical feature model editor (FME) [5] developed using GMF [14] to build the feature models. The GMF (Graphical Modelling Framework) is an Eclipse modelling project which provides infrastructure for building graphical editors based on models. The FME allows feature models to be built in a free form and uses common graphical notations of feature modelling which are mostly used when drawing feature models by hand. The FME also implements validation mechanism based on the validation properties/constraints mentioned above and does not allow users to build inconsistent models. It also warns the user if any of the validation properties are being violated upon saving the model.

B. Model Transformation

The feature models built using the feature model editor are then transformed into EMF metamodels at run-time and for each product line. This model to metamodel transformation is needed in order to instantiate the feature models and this in a way forces the instances to conform to their metamodel. This transformation is done using Epsilon Transformation Language (ETL)[15] which is part of the Epsilon framework. It provides execution engine for the transformation rules. By using a transformation language, we can achieve the benefits of Model-Driven Development. It is also easier to use rule-based language compared to a programming language. We might need to use the Epsilon validation framework later on to validate that the generated instances are valid instances of the feature models. We have looked at other transformation

[4]http://wiki.event-b.org/index.php/Feature_Modelling_Tool

[5]Contributed by Nikola Milikic (a University of Southampton Intern)

languages such as ATL [16] and RDL [17], but ETL is more suitable for Eclipse development and better in terms of documentation and support compared to others. After transforming feature models into metamodels, for different product lines, these are then used as an input to the feature configurator discussed below.

C. Feature Configurator

The feature configurator (FECON) allows the modeller to select a configuration of features. The tool then composes these selected Event-B features into a single feature. The FECON is a collapsible tree-structure editor (see Fig. 6) which extends our feature composition tool [8]. It provides a configuration mechanism where the user selects features that he wants to include in a particular product or a feature model instance. It enforces 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, it automatically selects/deselects features specified using *includes/excludes* constraints. The FECON also shows the associated Event-B machines and contexts for the features and highlights any conflicts that need to be resolved. At the moment, it detects naming conflicts (e.g. *variables* or *events* with same name) and provides ways to automatically resolve these name clashes either by making them disjoint through renaming 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 i.e. all the machines are merged into a machine and all the contexts are merged into a context. It 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. Figure 6 shows a configuration of the example feature model on the left and event fusion is shown on the right.

The composition of refined features here is not straight forward. When a refined machine is composed with other models, its variables may be typed in abstract models along with other invariants. These variables and invariants should also be composed along with the refined model. The user needs to guide the tool in selecting such elements. Now that we have a tool which can be used to build example case-studies, it would help us to figure out common patterns for composition and will be extended in the future to include such patterns to automate the composition process.

V. Production Cell Example

We have used the Production Cell (PC) case-study, an example of reactive system, which has been specified in a number of formal modelling languages. The PC is metal

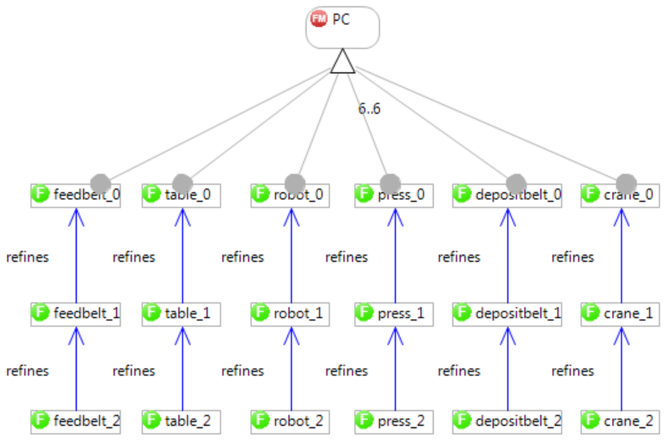
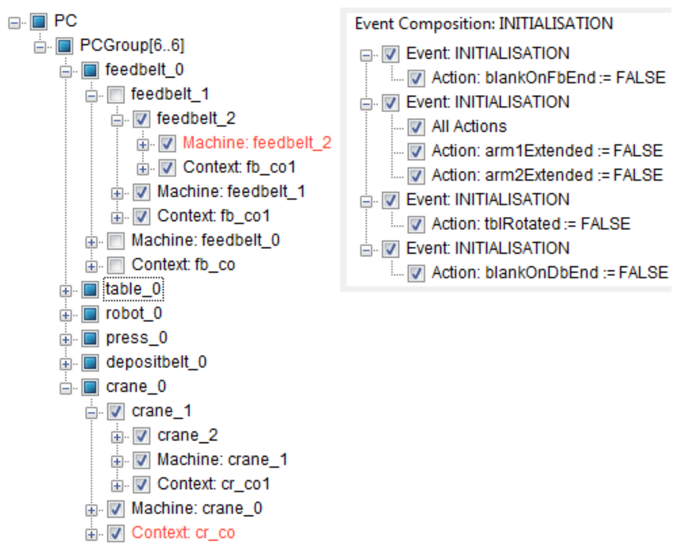


Fig. 5. PC Feature Model



```
PC
  PCGroup[6..6]
    feedbelt_0
      feedbelt_1
        feedbelt_2
          Machine: feedbelt_2
          Context: fb_co1
        Machine: feedbelt_1
        Context: fb_co1
      Machine: feedbelt_0
      Context: fb_co
    table_0
    robot_0
    press_0
    depositbelt_0
    crane_0
      crane_1
        crane_2
        Machine: crane_1
        Context: cr_co1
      Machine: crane_0
      Context: cr_co
```

```
Event Composition: INITIALISATION
  Event: INITIALISATION
    Action: blankOnFbEnd := FALSE
  Event: INITIALISATION
    All Actions
    Action: arm1Extended := FALSE
    Action: arm2Extended := FALSE
  Event: INITIALISATION
    Action: tblRotated := FALSE
  Event: INITIALISATION
    Action: blankOnDbEnd := FALSE
```

Fig. 6. Feature Configurator Screenshot

processing plant where metal blanks enter into the system through the feed belt and dropped on to the elevating-rotary table. The table elevates and rotates to a position where the first robot arm can pick up the blanks. The robot rotates anti-clockwise to drop the blanks in the press. The press forges the blanks which are then picked up by the second robot arm dropping on to the deposit belt. A moving crane then picks the blanks from the deposit belt and brings them back to the feed belt and completes a processing cycle. We have modelled this using Even-B, starting with the abstract model and refined by gradually adding further details in each refinement step. All of this is horizontal refinement. The proof obligations at each refinement step were discharged automatically and interactively to prove each step as a valid refinement.

The decomposition plays important role when modelling complex systems. It is easier to model and refine subsystems or modules than that of the entire system at once. It also enables several independent modules to be dealt with in parallel by different teams. Hence, we have decomposed the Event-B model of the PC in two ways i.e. one based on physical components and the other based on the various controllers. These two types of decomposition were used as example cases for the feature composition tool [8]. This experience showed some interesting results such as the control based decomposition/composition was more reusable than the one based on the physical components and also discovered the need to model the system fairly generic which can then be specialized and composed at instantiation stage.

We have used the same case-study for the feature modelling tool to build product line of a production cell. Figure 5 shows a feature model of the Production Cell. Here we have six features which are basically physical components in the processing unit. We have refined each one of these further using horizontal refinement, i.e., adding more functionality in each step. The variability here is in terms of refinement selection e.g. the feature 'table' is mandatory which means one of the features in the table's refinement chain must be selected. In this example, we had the Event-B features already specified, so feature modelling is useful in understanding the ways these features and/or their refinements can be composed to build instances of the production cell.

After building the PC feature model, we configured it to generate an instance, i.e., a product of the PC product line as shown in the Figure 6. As we selected the features that we wanted to include, the tool highlighted any conflicts which were then resolved using the automatic deseletion of redundant elements, as most of the features were sharing the same state variables. The box on the right shows the merging of all the INITIALIZATION events from selected machines into one, as we can only have one such event in the composite machine. After all the conflicts were resolved, an instance was generated by composing all the features i.e. all the machines into one machine and all the contexts into one context. We had to reprove the composite model because proofs were not carried through to the composite feature. This is what we will be exploring next as discussed in the future work section.

VI. Related Work

To our knowledge, there is no tool support for feature-oriented modelling or product line reuse within the formal methods domain. But various tools exist which support feature modelling for product lines such as CaptainFeature, XFeature, Pure::Variants, FeaturePlugin etc. [18]. The underlying concepts discussed in [19] are quite similar to our feature-oriented modelling in the domain of formal product line development. A brief comparison and discussion on these tools is given by Antkiewicz et al. [18]. Most of these tools do not mention how actual product-derivation works or how a feature model can be used to instantiate working application instance for a product line. They do not have clear mapping of features to actual components or program units and are mostly generic as compared to our domain specific feature modelling. The real problem arises when actual components are composed or integrated and then how to resolve any conflicts that may occur due to the components sharing the environment or how to deal with components having cross-cutting concerns. Our combination of feature modelling and Event-B can cover the complete development cycle i.e. feature models are instantiated as Event-B specification which can then be used to produce executable code for a particular product (e.g. using Event-B to C code generator [20]).

Another area that is closely related to our work is the definition of composition rules. Work is in progress where composition rules are used while composing Event-B features [21] and we will be extending our approach to include this in the future.

VII. Conclusion & Future Work

We have given an overview of our approach to introduce product line reuse i.e. the concept of feature modelling within formal methods using Event-B. We have developed a feature modelling tool for building feature models, configuring Event-B features and composing them to instantiate software product line systems by reusing and extending existing features. This was required because existing feature modelling tools do not provide enough facilities to give semantics to features and the resulting formal verification capabilities such as offered by Event-B. Our prototype tool will enable us to experiment with different case studies to figure out common practices and at the same time will help us to define design patterns for such type of formal product line development. This contribution should substantially increase productivity and improve user confidence in using feature-oriented modelling and formal methods for systems development.

In future, we will explore how to reuse proofs associated with the features. When we compose Event-B features into a composite feature to generate a feature model instance, the tool generates proof obligations (POs) for verification. Most of the POs for the composed features still exist for the composite feature, and may have already been discharged interactively. Hence, it would be useful if the tool could reuse interactively discharged POs to save user time and effort. We will extend and implement the ideas of composition POs as discussed in [21]. There is also a need to automate the composition process so as to avoid user interaction as much as possible. This might require adding composition rules as annotations to the feature models which can then be used while instance generation.

References

[1] J.-R. Abrial, "Formal methods in industry: achievements, problems, future," in *ICSE '06: Proceedings of the 28th international conference on Software engineering*. New York, NY, USA: ACM, 2006, pp. 761–768.

[2] J. P. Bowen and M. G. Hinchey, "The use of industrial-strength formal methods," in *COMPSAC '97: Proceedings of the 21st International Computer Software and Applications Conference*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 332–337.

[3] C. Metayer, J.-R. Abrial, and L. Voisin, "Event-B Language," EU Project IST-511599 -RODIN, RODIN Deliverable 3.2, May 2005.

[4] J. R. Abrial, *The B-book: assigning programs to meanings*. New York, NY, USA: Cambridge University Press, 1996. [Online]. Available: http://portal.acm.org/citation.cfm?id=236705

[5] P. Clements and L. Northrop, *Software Product Lines : Practices and Patterns*. Addison-Wesley Professional, August 2001.

[6] K. Lee, K. C. Kang, and J. Lee, "Concepts and guidelines of feature modeling for product line software engineering," in *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*. London, UK: Springer-Verlag, 2002, pp. 62–77.

[7] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson, "Feature-oriented domain analysis (foda) feasibility study." Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, Tech. Rep., 1990.

[8] A. Gondal, M. Poppleton, and C. Snook, "Feature composition - towards product lines of Event-B models," in *1st International Workshop on Model-Driven Product Line Engineering (MDPLE'09)*. CTIT Workshop Proceedings, June 2009. [Online]. Available: http://eprints.ecs.soton.ac.uk/17547/

[9] M. Poppleton, B. Fischer, C. Franklin, A. Gondal, C. Snook, and J. Sorge, "Towards Reuse with "Feature-Oriented Event-B"." Nashville, TN: McGPLE: Workshop on Modularization, Composition, and Generative Techniques for Product Line Engineering, October 2008.

[10] J. McGregor, "Preparing for automated derivation of products in a software product line," Carnegie Mellon Software Engineering Institute (SEI), Technical Report CMU/SEI-2005-TR-017, September 2005.

[11] J. Bosch, *Design and use of software architectures: adopting and evolving a product-line approach*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000.

[12] K. Czarnecki, S. Helsen, and U. Eisenecker, "Staged configuration through specialization and multilevel configuration of feature models," *Software Process: Improvement and Practice*, vol. 10, no. 2, pp. 143–169, 2005. [Online]. Available: http://dx.doi.org/10.1002/spip.225

[13] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks, *Eclipse Modeling Framework*, 2nd ed., ser. The Eclipse Series. Addison-Wesley Professional, December 2008.

[14] Online, "Graphical modeling framework (GMF)," Project Website: http://www.eclipse.org/modeling/gmf/. [Online]. Available: http://www.eclipse.org/modeling/gmf/

[15] D. Kolovos, R. Paige, and F. Polack, "The epsilon transformation language," 2008, pp. 46–60. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-69927-9_4

[16] F. Jouault and I. Kurtev, "Transforming models with atl," *LECTURE NOTES IN COMPUTER SCIENCE*, pp. 128–138, 2006.

[17] "Model transformation framework," http://www.alphaworks.ibm.com/tech/mtf, accessed Ocober 2009.

[18] M. Antkiewicz and K. Czarnecki, "Featureplugin: feature modeling plug-in for eclipse," in *eclipse '04: Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange*. New York, NY, USA: ACM, 2004, pp. 67–72.

[19] V. Cechticky, A. Pasetti, O. Rohlik, and W. Schaufelberger, "Xml-based feature modelling," 2004, pp. 101–114. [Online]. Available: http://www.springerlink.com/content/yvw9e22nanqmv27f

[20] S. Wright, "Automatic Generation of C from Event-B," in *Workshop on Integration of Model-based Formal Methods and Tools*, February 2009. [Online]. Available: http://www.cs.bris.ac.uk/Publications/Papers/2000990.pdf

[21] J. Sorge, M. Poppleton, and M. Butler, "A Basis for Feature-oriented Modelling in Event-B," in: ABZ2010, 23-25 Feb 2010, Orford, Canada.