Composing Event-B Specifications - Case-Study Experience

Ali Gondal, Michael Poppleton, Michael Butler

School of Electronics and Computer Science
University of Southampton, Southampton, SO17 1BJ, UK
{aag07r, mrp, mjb}@ecs.soton.ac.uk

Abstract. Event-B is a formal method, based on set theory and first-order logic, for specification and verification of reactive systems supported by the Rodin tool kit. Feature modelling is a well-known technique for managing variability and configuring products within software product lines (SPLs). Our objective is to explore whether we can use existing Event-B composition techniques and tooling for feature-based product line development. If case-study experiments reveal these mechanisms to be inadequate, then they also should suggest further research directions. The main objective is to maximise the amount of reuse. This includes avoiding as far as possible having to reprove a composed specification when the models being composed have already been proven. We have modelled two case-studies in Event-B using both horizontal and vertical refinements. This work contributes by analysing existing tools and techniques in Event-B for feature-based development, exploring composition related issues by modelling example case-studies and suggesting further tooling requirements.

1 Introduction

Event-B [1] is a formal modelling language, a successor of Abrial's classical B [2]. It was developed as part of the RODIN[1] and earlier EU projects. The DEPLOY[2] project, along with industrial partners, is currently focused on deploying this work into industry. Event-B, a state-based language, is based on set theory and first-order logic and allows the specification and verification of reactive systems. The correctness of a model is defined by invariant properties on its state which must be preserved by all transitions in a system, called *events*. An event is enabled when certain pre-conditions on the event, called *guards*, become true. Verification conditions (known as proof obligations or *PO*s), concerned with model consistency, i.e., invariant preservation, are generated and discharged by proof support tools. Event-B is further supported by the integrated Rodin toolkit comprising editors, theorem provers, animator and model checker.

A Software Product Line (*SPL*) refers to a set of related products built from a shared set of resources with a common base while having significant variability

[1] RODIN: EU Project IST-511599. http://rodin.cs.ncl.ac.uk
[2] DEPLOY: EU Project IST-214158. http://www.deploy-project.eu

to meet the user requirements [3]. SPLs provide the benefits of reusability in reducing the time to market, lower cost and reduce effort involved in product development. Feature modelling [4] is a well-known technique for building SPLs. The *feature* has been defined as "a logical unit of behaviour specified by a set of functional and non-functional requirements" [5] and usually referred as a property of the system that is of some value to the stakeholders. A feature model is drawn using tree structured feature diagrams to describe variability among the product line members, and the valid ways in which these features can be instantiated to generate various products.

Our objective is to explore how we can devise a feature-oriented approach for reuse, and ultimately for modelling product lines in Event-B. This will allow the reuse of existing specifications for a product line to build further products of the family without redoing all the modelling and proof effort. In the feature-oriented software development (*FOSD*) community, a feature is in general made up of different modules or classes. In Event-B, a feature is a well-formed model, the basic modular unit in the Rodin tool. Eventually, we will consider a feature to be a partial Event-B model; this will require further research and tooling development. In a state-machine semantics, a feature can add or remove states and transformations. Also, we are specifying features in the problem space (abstract requirements) compared to the features in solution space (concrete implementation, e.g., java classes) as considered by the FOSD community. In order to build a product, we must compose various features and that brings us to the core issue of composing specifications. The real benefit of the product line modelling can not be achieved without automatically proving composed specifications. At present, there are three types of composition for Event-B which guarantee refinement preservation.

We present two case-studies to investigate whether the existing composition techniques are adequate for feature-oriented modelling in Event-B and suggest further tooling requirements for such development. We call this formal modelling of a system to produce reusable specifications "Domain Modelling" which corresponds to domain engineering activity within SPL engineering. Also, we seek to find some composition patterns that can be applied to automate the composition process to save time and effort. This work can be encouraging for SPL community to use formal methods as we have adapted and extended existing feature modelling notations. This paper focuses on the application of our feature modelling and composition methodologies to the case-study work. This work builds on the methodology of Snook et al. [6] for the formal verification of product lines. We want to explore how their methodology can be developed by experimenting with the case-studies using Event-B and suggesting further research.

Section 2 gives a brief introduction of the Event-B language and the (de)composition techniques for Event-B are presented in Section 3. Section 4 summarises our feature-oriented modelling support for Event-B. Section 5 describes the two case-studies we have modelled using Event-B. Related work is given in Section 6. Section 7 concludes the paper and suggests directions for the future work.

2 Event-B Introduction

An Event-B model consists of a *machine* and multiple *contexts*. The machine specifies the behavioural 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 give 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 the typing for the variables 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 known as local variables). The variables are initialized using a special event called `Initialization` which is unguarded. An event has the following syntax:

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

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. 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` and initialized 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. This example is taken from our ATM case-study which is explained in Section 5.2.

Refinement is a top-down development method and is at the core of Event-B modelling. We start by specifying the 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 the correct refinement of the abstract model by discharging suitable refinement POs. Typically, we classify the refinement into horizontal and vertical refinements [1]. 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 (also known as data refinement), the focus is on design decisions, i.e., transforming and enriching data types and the elaboration of algorithms. 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.

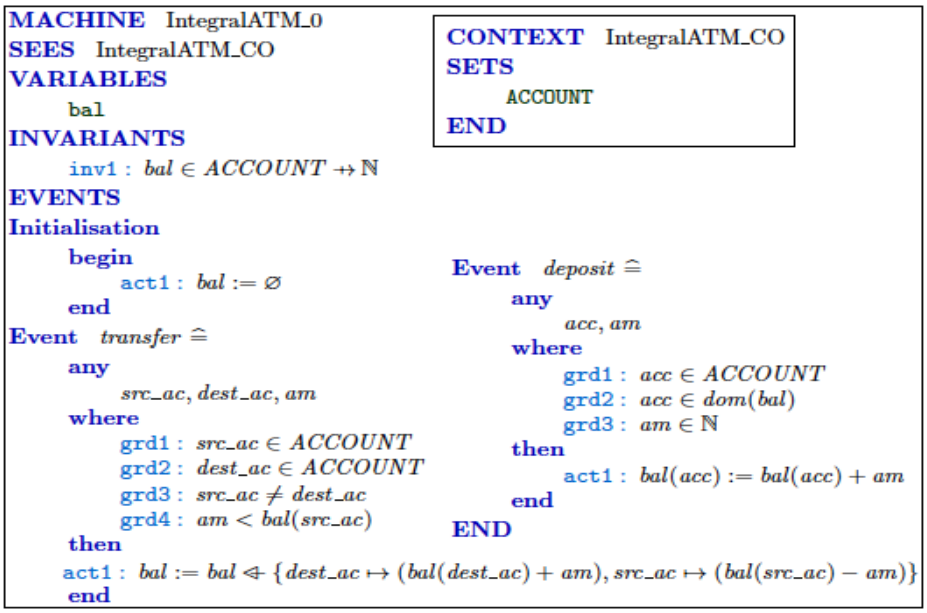

Fig. 1. Integral ATM abstract model

3 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*) [7] and *shared-event* [8] decomposition (*SED*). Like Event-B language, these techniques are influenced by earlier formalizms such as CSP [9] and Action Systems [10]. The refinement preserving nature of these decomposition techniques differentiates these from the feature-based decomposition with in the FOSD community.

In *shared-variable* 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 2 (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.

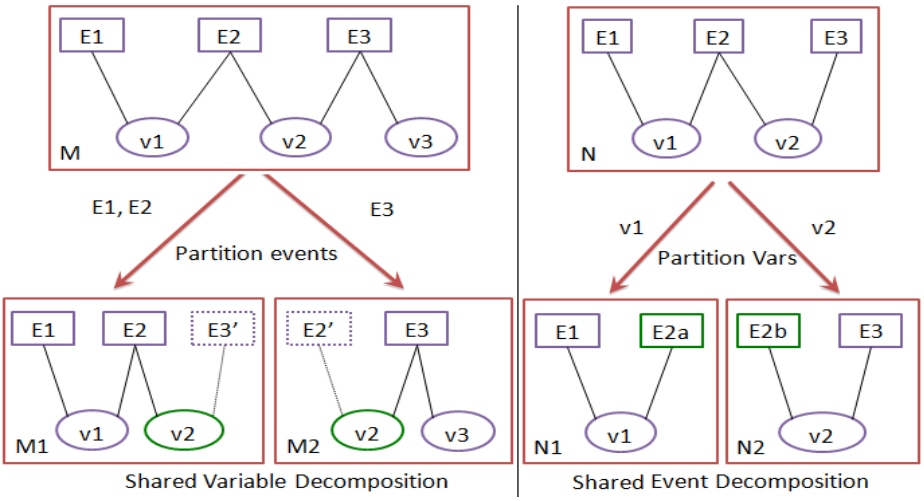

Fig. 2. Decomposition types in Event-B

The *shared-event* style is based on shared events rather than shared variables. During the decomposition, the independent events are kept in each decomposed component and the shared events are split. Figure 2 (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 [11] 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 [12] 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. A prototype feature composition tool [13] has been developed which allows the composition of models using this style. Since the tool is not restrictive, it does not enforce the correct fusion style composition. This means that the user needs to make sure that the composition is performed correctly. Also, the tool does not automatically discharge proof obligations for the composite model at the moment but our case-study results will provide some directions to deal with proof reuse as mentioned in future work (Section 7).

4 Feature-orientation for Reuse with Event-B

We have adapted and extended the cardinality-based feature modelling notations [14]. Apart from our prototype tool (discussed below), there are no feature modelling tools specifically for Event-B. Tooling specific to Event-B is required because standard feature modelling tools are not enough to provide Event-B semantics and all the modelling, proving, animation and model checking facilities provided by the Rodin toolkit.

We define an Event-B *feature* as an Event-B model which consists of a machine and one or more contexts. An Event-B feature model - except for its leaf nodes - supports all the usual feature modelling constructs and constraints [14], subject to some syntactic customization [15]. The leaf level nodes in the tree denote Event-B features. So, a leaf feature could be a whole Event-B development modelling various refinement levels[3]. We have also developed a prototype feature modelling tool [15] that can be used to build feature models of product lines. These feature models can be configured by selecting a set of features, resolving any conflicts and composing these to model an instance of the product line.

5 Approach to Experimental Case-Study Work

We have modelled two well-known case-studies in Event-B using different modelling styles to explore composition related issues that may arise when we try to model product lines in Event-B. Our focus has been the feature-oriented development approach while modelling these systems and to maximise the amount of reuse that can be achieved. We have used the different types of decompositions

[3] This notion of feature has evolved over the time and is slightly different to what we have published previously.

available in Event-B (see Section 3) to decompose each system into a number of features which can be independently refined. These features can then be composed later to build a particular product of the product line. So, we have used the top-down methodology of Event-B in order to build an asset of reusable features to experiment with. This case-study work should also suggest further guidelines for feature modelling, should one needs to do so using the existing techniques and tool. Following is a brief overview of each of the case-studies.

5.1 Production Cell Case-Study

The Production Cell (PC) [16] is an example of a reactive system, which has been specified in a number of formal modelling languages. The PC is a metal processing plant where metal blanks enter into the system through the feed belt and are dropped on to the elevating-rotary table. The table elevates and rotates to a position so that 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, that have not been forged properly, and brings them back to the feed belt for reprocessing. Figure 3 shows the top view of the production cell plant.

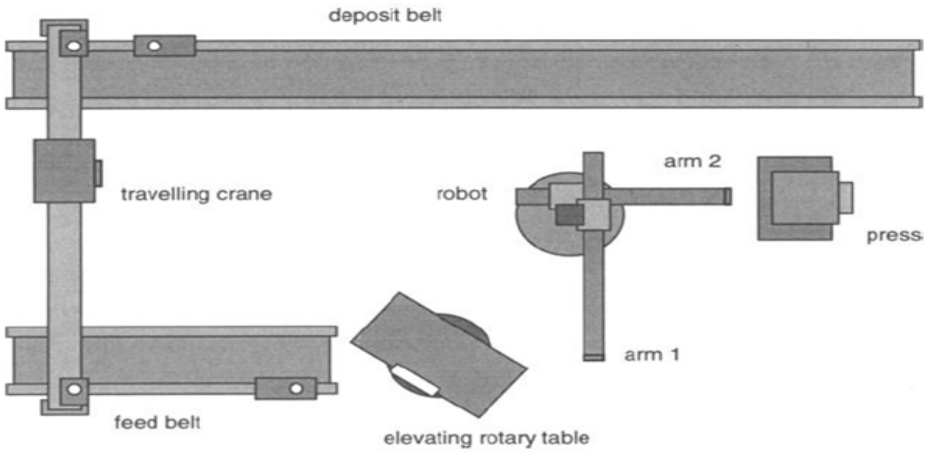

Fig. 3. Production Cell Plant [16]

The production cell was modelled in two ways, (i) based on the physical components and (ii) the controllers of the system. We will refer to these as component or control based modelling of PC. This allows us to use two different methods of modelling the same system in Event-B and analysing our methodologies for the feature-based modelling framework using existing tools and techniques in Event-B.

PC Component-based: We started with an abstract model of the production cell where we only model the processing of metal blanks from unforged to forged state. We then introduced further requirements in the next three levels of refinements. These were all horizontal refinements. At this stage we found it reasonable to decompose this integral model into separate models for each of the physical components of the PC. Each of these components can then be refined vertically to include sensors and actuators bringing it closer to implementation. We tried both the shared-event (*SED*) and shared-variable decomposition (*SVD*). Since there were shared-variables in the model (e.g. *blanks* shared by all components), it was not possible to use the SED technique without further refinement. The decomposition of the integral model using SVD resulted in six sub-models (components), i.e., feed belt, table, robot, press, deposit belt and crane. We could then refine each of these sub-models of PC independently while maintaining the restrictions of the SVD style, i.e., not to refine shared state and the external events. For example, we vertically refined the 'press' component up to three refinement levels by introducing actuators and sensors for handling the press using the refinement patterns for control systems [17].

We can build more variants of PC by selecting a different configuration of these reusable physical components. For example, if we need to increase productivity, we can model a production cell with two press components for forging the blanks and two robots. We can build another product of PC by modelling a different kind of press and reusing existing models where a second robot picks a blank processed by press1 to be processed differently by press2.

PC Controller-based: The control-based modelling of Production Cell was done by grouping the requirements for various controllers of the system. Each group of requirements was modelled as a controller. This was not just decomposing the PC system into controllers but also generalized these controllers so that these can be specialized and reused for modelling various PC components. Hence, the control-based modelling of PC was a result of decomposition plus generalization. The control-based PC models consisted of loader, movement, rotation and magnet controllers. These models were generic so that these can be used later on for modelling a particular physical component of PC. A complete PC model could also be modelled by instantiating and composing these control-based reusable features. We only discuss the magnet and movement controllers here. We refined these using the sensing and actuation patterns for refining control systems as suggested in [17]. Following is the detail of modelling and refining the magnet and movement features respectively.

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

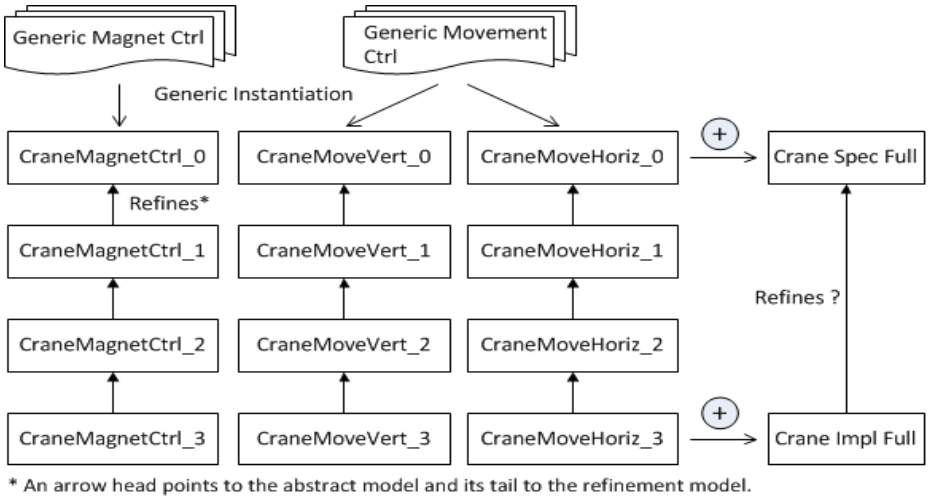

Fig. 4. Crane Instantiation

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

Instantation & 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. 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 movement controllers for moving the crane horizontally and vertically, as shown in Figure 4. Figure 5 shows a simple example where event *PickBlank* of Magnet controller is specialized for the crane component.

Here the generic model parameter *X_comp_X* 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 current Rodin tool does not support generics.

```
event PickBlank                    (before)
  any b
  where
    @grd1 X_comp_X ∉ ran(position)
    @grd2 b ∈ dom(position)
  then
    @act1 position(b) ≔ X_comp_X
end
```

```
event CranePickBlank               (after)
  any b
  where
    @grd1 crane ∉ ran(position)
    @grd2 b ∈ dom(position)
  then
    @act1 position(b) ≔ crane
end
```

Fig. 5. Event Specialization for Crane

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 6 shows two events from crane and movement controllers for picking up blanks by crane and the movement of crane to feed belt (before composition). Figure 7 (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 7 specifies that the crane can only move to 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 model the entire PC model, we will need to strengthen this guard to say that the crane can only pick a blank from the deposit belt.

We call this style of composition 'feature composition' which extends the fusion composition where additional predicates can be added during the composition. As of yet, this style of composition does not guarantee refinement preservation between the composed abstract and implementation models. 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. 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

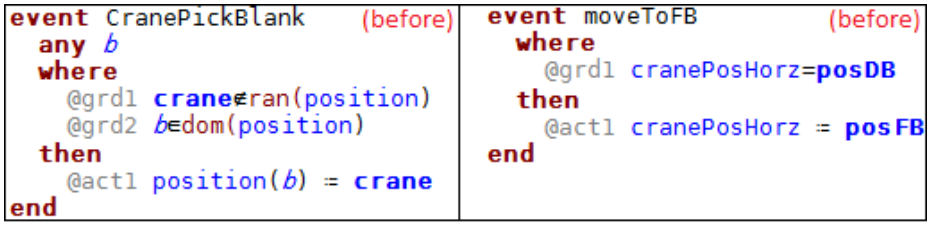

Fig. 6. Events of Magnet and Movement Controllers

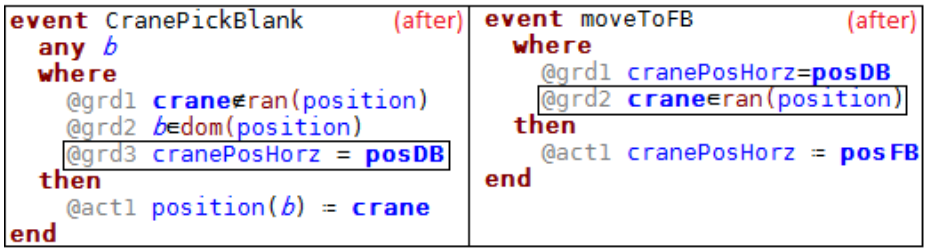

Fig. 7. Guard strengthening of events during composition

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 5.2 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.

Evaluation: By modelling the PC in two different ways, we can argue that the amount of reusable assets can be increased by modelling the system from a reuse perspective. If we model features by generalizing, as in control-based modelling, then this increases the amount of reuse. For example, a PC model can be built by composing six physical components. If we want to model the same PC model using control-based features, we need to specialize and compose many generic features, e.g., a crane component would require the instantiation and composition of three features. In fact, we are modelling coarse-grained features in component-based PC where as the features in control-based PC are fine-grained. The top-down development of component-based PC resulting in coarse-grained features provides the communication between various physical components. Where as the bottom-up development of control-based PC resulting in fine-grained features provides more reuse opportunities. We could only utilise the potential benefits of reuse if we can automate this process of specialization and composition.

We could further explore reusability of component-based features by modelling them completely independent of each other, unlike we did for the component-based PC by decomposing horizontally refined integral model to get these. Then again there will be a question of how these components will communicate for

passing blanks to each other when composing these to build a PC? Another option is to decompose horizontally refined integral model using SED where the components would communicate by synchronising their shared-events. This means that the individual components are not completely independent of each other and their reuse must be constrained by the topology of the PC. We could also have generic predicates in each component which must be specialized during the composition. For example, the table receives a blank from the feed belt (fb). So, a predicate $position(b) = XinputCompX$ could be used which must be specialized to $position(b) = fb$. In this way we can model the static variability represented by the connection topology of the physical components of PC. For instance, the input component for table is feed belt and the output component is robot which picks blanks from the table. It would be useful to have a tool that reads product configuration from a file to instantiate a product line, i.e., the $XinputCompX$ is instantiated to fb in the above example. So, our objective is to prototype some mechanism (e.g., syntactic definitions or patterns) that could be used for generic instantiation and composition of Event-B specifications.

We also found that the process of vertically refining the features to include sensors and actuators is quite similar in both the modelling styles. This is because we used the patterns for refining control systems [17]. This refinement process can also be automated where features can be refined vertically to model actuators and sensors. This requires further work.

5.2 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 pin related services. Other services can also be provided by ATMs which vary for different banks and ATM locations, e.g., mobile top up and cash deposit etc. In this Section, we will only discuss balance transfer, deposit and withdrawal features of the ATM.

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. 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 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 compliment the feature-based development in Event-B.

We started with an integral model of the ATM that allows cash deposit and balance transfer between two accounts. This abstract model is shown in Figure 1. We then decomposed this model into deposit and balance transfer features using shared-variable decomposition (SVD). The event $deposit$ goes into the deposit feature and the $transfer$ event goes to the transfer feature - see Figure 8. Both features will have shared variable bal and external events, e.g., deposit feature will have $transfer$ event as external event which must not be refined

along with the shared variable. These features were then refined horizontally and vertically.

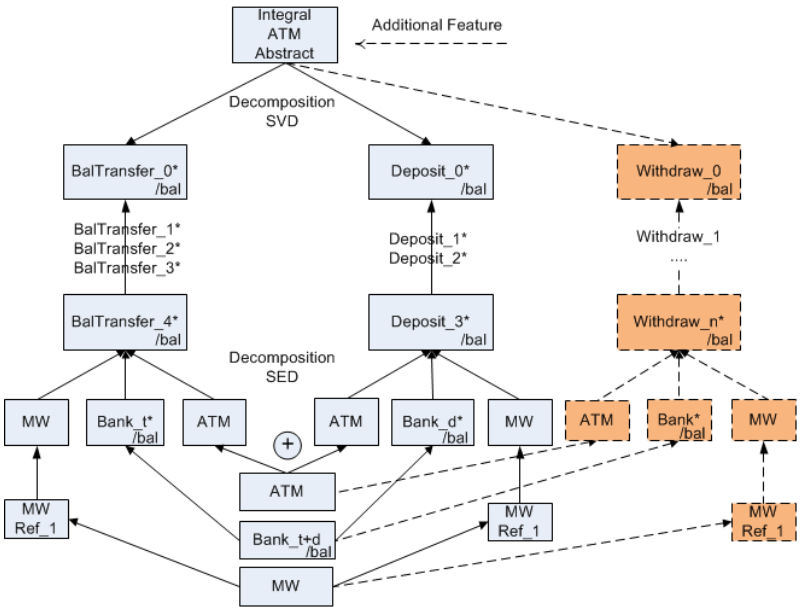

Fig. 8. Refinement & (de)composition architecture for ATM features

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 introduce 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 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).

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 8 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. 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 $\text{Bank}_{t+d} = \text{Bank}_t + \text{Bank}_d$, 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. There is to date no shared-variable composition (SVC) tool support.

Evaluation: Consider generalizing the above approach. For example, 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 8). We elaborate the top-level integral model to include withdrawal feature and decompose it into three components (i.e., deposit, balance transfer and withdrawal). Provided the new feature is *separable* - in the sense that in the SVD refinement the other two feature models remain unchanged - then all we have to do is refine the withdrawal component. Since the deposit and balance transfer components have already been proven, new POs will only be generated for the newly added external events corresponding to events of the withdrawal component acting on `bal`. 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.

Recall that in SVD the shared-variable (SV) may not be refined - this is very restrictive, and further investigation is required to establish whether after architectural recomposition, it is possible to then refine the SV for implementation purposes.

We have examined a specific pattern of mixed decomposition-recomposition - SVD followed by SED and then SVC 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. 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.

6 Related Work

Formal modelling of SPLs is not a new area and work has been done to formally specify requirements and reusing these to produce variants of a product line [18]. HATS [19] provides a methodology for applying formal methods at differ-

ent stages of SPL development cycle. This seems to be a promising work to bring together the domains of formal methods and SPL. All the existing techniques for composing Event-B specifications have been discussed in this paper and we have not seen any work being done for modelling SPLs in Event-B. Lau et al. [20] 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. Some work has been done by Sorge et al. [21] which deals with invariant proof obligations for composing features. This does not support feature refinement and event fusion which is required to complement our feature modelling framework. FEATUREHOUSE [22] is another tool that allows the composition of artefacts and supports various languages with option to include more. It would be very useful to see if Event-B can be included in this framework and whether the composition of Event-B refinement chains can be achieved with proof reuse.

7 Conclusion & Future Work

We have given an overview of the two case-studies that we have modelled to explore existing capability for feature-based development in Event-B. We sought to identify patterns of refinement, generic instantiation, decomposition and composition that could be exploited for reuse. Further, we sought to identify further requirements for tooling, and further research - both experimental and theoretical - to develop a feature-based reuse capability for Event-B/Rodin.

In the component-based PC model we explored SVD. Then considering the option of modelling physical components as features, it appeared that generic elements defining the connection of these features could be instantiated from contextual data about the topology/connectivity of the cell. This idea is close to the shared-event composition of such features and needs further examination.

In the controller-based modelling approach to the PC we found a finer-grained set of generic template models which could be instantiated and composed into physical features, which could then be further composed as above. However, this freer form of composition (which we call feature composition) carries no "correct-by-construction" guarantee, and thus a full reproof burden. It may be that this is an opportunity for fusion composition, which remains to be investigated. Investigation is also needed to find modelling and composition patterns that will give partial PO reuse, thus partial "reproof-for-free". For example, guard-strengthening of events preserves simple refinement, but can introduce deadlock problems.

In the ATM case, we explored one specific pattern of mixed decomposition-recomposition - SVD followed by SED and then SVC. This looks like a promising pattern for composition of abstract features followed by architectural design, subject to structural constraints about shared variables. Future work will reveal exactly what those constraints are, and will explore other such patterns of

refinement and (de)composition for feature-based modelling. It would also be interesting to explore how to deal with feature interactions using our case-study work.

References

1. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. First edn. Cambridge University Press (June 2010)
2. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press, New York, NY, USA (1996)
3. Clements, P., Northrop, L.: Software Product Lines : Practices and Patterns. Addison-Wesley Professional (August 2001)
4. Lee, K., Kang, K.C., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering. In: ICSR-7, UK, Springer-Verlag (2002) 62–77
5. Bosch, J.: Design and use of software architectures: adopting and evolving a product-line approach. ACM Press/Addison-Wesley, USA (2000)
6. Snook, C., Poppleton, M., Johnson, I.: Rigorous engineering of product-line requirements: a case study in failure management. IST **50**(1-2) (Jan 2008) 112–129
7. Abrial, J.R., Hallerstede, S.: Refinement, Decomposition, and Instantiation of Discrete Models: Application to Event-B. Fundam. Inf. **77**(1-2) (2007) 1–28
8. Butler, M.: Synchronisation-based Decomposition for Event-B. In: RODIN Deliverable D19 Intermediate report on methodology. (2006)
9. Hoare, C.A.R.: Communicating sequential processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
10. Back, R., von Wright, J.: Trace refinement of action systems. In Jonsson, B., Parrow, J., eds.: Concurrency Theory. Volume 836 of LNCS. (1994) 367–384
11. Silva, R., Pascal, C., Hoang, T.S., Butler, M.: Decomposition tool for event-b. In: Workshop on Tool Building in Formal Methods - ABZ Conference. (January 2010)
12. Poppleton, M.: The composition of event-b models. In: ABZ2008: Int. Conference on ASM, B and Z. Volume 5238., Springer LNCS (September 2008) 209–222
13. Gondal, A., Poppleton, M., Snook, C.: Feature composition - towards product lines of Event-B models. In: MDPLE'09, CTIT Workshop Proceedings (June 2009)
14. Czarnecki, K., Helsen, S., Eisenecker, U.: Staged configuration through specialization and multilevel configuration of feature models. Software Process: Improvement and Practice **10**(2) (2005) 143–169
15. Gondal, A., Poppleton, M., Butler, M., Snook, C.: Feature-Oriented Modelling Using Event-B. In: SETP-10, Orlando, FL, USA (2010)
16. Lindner, T.: Task description. In Lewerentz, C., Lindner, T., eds.: Formal Development of Reactive Systems. Volume 891 of LNCS., Springer (1995)
17. Butler, M.: Towards a cookbook for modelling and refinement of control problems. http://deploy-eprints.ecs.soton.ac.uk/108/ (2009)
18. Kishi, T., Noda, N.: Formal verification and software product lines. Commun. ACM **49**(0001-0782) (December 2006) 73–77
19. Clarke, D., Diakov, N., Hähnle, R., Johnsen, E.B., Puebla, G., Weitzel, B., Wong, P.Y.H.: HATS-a formal software product line engineering methodology. In: Proc. Intl. Workshop on Formal Methods in SPL Engineering, South Corea. (2010)
20. Lau, K.K., Wang, Z., Wang, A., Gu, M.: A component-based approach to verified software: What, why, how and what next? In Chen, X., Liu, Z., Reed, M., eds.: Proc. 1st Asian Working Conference on Verified Software. (2006) 225–229
21. Sorge, J., Poppleton, M., Butler, M.: A Basis for Feature-oriented Modelling in Event-B. In: ABZ2010. (February 2010)
22. Apel, S., Kastner, C., Lengauer, C.: FEATUREHOUSE: Language-independent, automated software composition. ICSE '09, USA (2009) 221–231