

# A Basis for Feature-oriented Modelling in Event-B

Jennifer Sorge, Michael Poppleton, Michael Butler

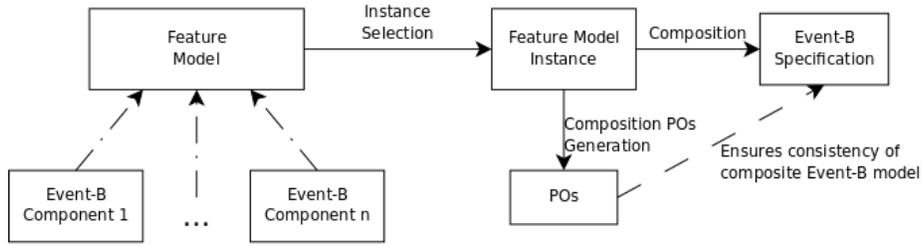
{jhs06r, mrp, mjb}@ecs.soton.ac.uk  
DSSE, ECS, University of Southampton, UK

**Abstract.** Feature-oriented modelling is a well-known approach for Software Product Line (SPL) development. It is a widely used method when developing groups of related software. Due to reuse methods, the development of the software product is quicker, less expensive and of higher quality. However, this approach is not common in formal methods development, which is generally high cost and time consuming, yet crucial in the development of critical systems. We present a method to integrate feature-oriented development with the formal specification language Event-B. Our approach allows the user to map a feature from the feature model to an Event-B component, which contains a formal specification of that feature. We also present some patterns, which assist the user in the modelling of Event-B components. We describe a composition process which consists of the user selecting an instance in the feature model and then constructing this instance in Event-B. While composing, the user may also discharge new composition proof obligations in order to ensure the model is consistent. The model is then constructed using a number of composition rules.

## 1 Introduction

Software families are very often produced through reuse methods and other software product line approaches, as it is known to produce better software, is less costly and less time-consuming. Software product line (SPL) approaches can be applied to a set of related software that contains some commonalities, however differs in certain aspects, which we refer to as variability of a product line [1]. To our knowledge, in the critical software domain, where formal methods are used to verify and reason about systems, SPL approaches are not applied.

In this paper, we present a method which integrates an SPL approach with formal methods, or more specifically, which integrates feature modelling [2] and Event-B [3]. We show some modelling patterns and guidelines that will help the user to model features in Event-B. A feature can be regarded as a piece of program functionality or a requirement [4]. We then represent these Event-B features (referred to as components) in a feature model. The feature model should represent a complete product line and through selection of several features an instance of a product line can be chosen. The leaf nodes in the feature model can be mapped to an Event-B component, and after instance selection,



**Fig. 1.** Overview of Composition Process

these components can be retrieved. The process of composition is then applied to compose these components. Composition of components means that all components are merged into one composite Event-B model. In order to do this, we develop a set of composition proof obligations (PO) which can be discharged before constructing the composite machine. The composition of components is an n-wise composition, however is carried out as sequential pair-wise composition, where composition order is not important because it is essentially conjunction. We present a number of composition rules which can be applied to pair-wise composition to get a valid composite model.

Figure 1 illustrates the composition process. The feature model is formed by features which might be associated with Event-B components. A subset of features from the feature model can be selected to form a feature model instance, thereby selecting several of these Event-B components. These components are composed pair-wise, and composition POs can be discharged to prove properties and to ensure consistency of the composition. The final Event-B machine represents the formal specification which is associated with the feature model instance and is obtained by composing these components.

The aim of this work is to facilitate the reuse of formal models, as it is currently done in non-formal SPLs. Due to the increase of complex systems, formal methods are on the rise. Thus, an integration of formal methods and product line approaches appears to be practical, since the need for formal methods development has increased and is used for systems that would benefit from SPL development. Several critical software systems, in the area of data storage, aviation, automotive technology and others, are becoming more and more important in today's life, which results in the need for higher productivity and decrease in cost during the development. Developing a formal system as a product line might result in higher reliability, since common parts of the system should be verified and tested several times throughout instance creation.

Our contribution is the integration of feature models and Event-B, and thus the provision of a method that allows SPL development in the critical domain. Composition POs are presented, which follow from existing Event-B POs. We show how composition POs can be simplified or do not need to be re-proved, based on what is already known through discharging of existing Event-B POs. In this way, we avoid re-proof of several properties. Finally, we present a set

of composition rules which aid in the construction of a syntactically correct composite model. This is demonstrated by a case study.

The remainder of this paper will proceed as follows. In section 2 we give an overview of standard feature modelling notation and describe the formal method Event-B. In section 3 we show the integration of feature models and Event-B components. We then describe a modelling pattern for the definition of user-defined types. Section 4 describes the composition process, which starts by selecting an instance and then discharging composition POs. We demonstrate how these composition POs were derived from existing POs. Composition rules are defined and we demonstrate by using a case study example. In section 5 we discuss related work. Section 6 contains concluding remarks and future work.

## 2 Background

In this paper we discuss an approach that integrates feature modelling notation and Event-B. The following sections provide some background information on the concepts of feature modelling and Event-B.

### 2.1 Feature Modelling

We use standard feature modelling notation [2,5] to specify product lines. Feature modelling is a widely used technique to represent a set of related software products that share some commonalities and vary in different points.

In Figure 2 we introduce the basic constructs of a feature model. This is a summary of the most common constructs of different notations.

	feature
	mandatory
	optional
	optional XOR
	mandatory XOR
	OR with cardinality
	includes

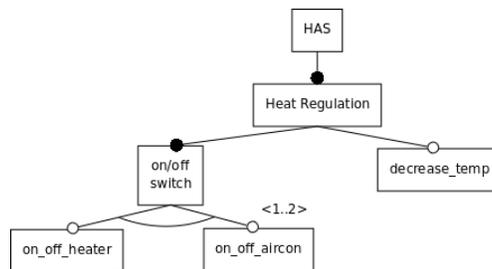
**Fig. 2.** Most Common Feature Modelling Constructs

A rectangular box is used to indicate a feature in a feature diagram. A feature can either be mandatory or optional. If a feature is mandatory this means that it has to be selected if its parent is selected, whereas an optional feature may or may not be selected if its parent is selected. The XOR group means that

one and only one feature within this group may be selected. If the XOR group is optional, this means that zero or one feature can be selected if the parent is selected, whereas a mandatory XOR group means that exactly one feature must be selected if its parent is selected. The OR group with cardinality allows to group a specific number of features. The cardinality indicates the minimum amount of features that may be selected, and the maximum amount of features that can be selected. In Figure 2 the minimum is two and the maximum is three features within this group. An includes relationship between features can be indicated by an arrow with a dotted line.

An example of a feature model is presented in Figure 3. This is a simplified extract of the home automation system (HAS) feature model we will present later.

The feature model consists of the root feature *HAS*. In this example, we only show the *Heat Regulation* tree of the feature model, which is represented as a mandatory feature. The *on/off switch* feature is mandatory and has an OR group with cardinality  $\langle 1..2 \rangle$ . This means that either *on\_off\_heater* or *on\_off\_aircon* or both feature can be selected. The feature diagram also contains an optional feature *decrease\_temp*.



**Fig. 3.** Simplified Home Automation System Feature Model

The example in Figure 3 shows that a feature model represents all possible valid configurations of a product line. For example, the configuration containing *on\_off\_aircon* and all of its parents is a valid configuration, however, *on\_off\_aircon* and *decrease\_temp* and all of their parents is also a valid configuration. This feature model has 6 valid configurations in total.

## 2.2 Event-B

In this paper we use the formal specification language Event-B [3], which is developed by Abrial and is based on the B method [6]. Event-B is based on first-order logic and set theory, and is structured into a dynamic part (describing system behaviour) and a static part (describing constant data and types). The dynamic part is referred to as a machine, and the static part is called a context. Proof obligations are associated with Event-B models. They give semantics to

Format	Assignment Type
$v := E(p, v)$	Deterministic Assignment
$v \in E(p, v)$	Non-deterministic Set Assignment
$v :  R(c, s, p, v, v')$	Non-deterministic Assignment with before-after Predicate

**Table 1.** Event-B Assignments

an Event-B model and are used to reason and prove certain properties about the model. Event-B can be developed using the Rodin platform[7], which is an extensible development platform and is currently supported by the DEPLOY project<sup>1</sup>. The Rodin platform contains a number of plugins that help with the development of Event-B models; examples are an animator and theorem provers.

The structure of an Event-B model is shown in Figure 4. A machine may see zero or more contexts. It contains a set of typed variables  $v$ . The machine contains a set of invariants  $I$ , which are predicates. They describe properties of the machine over the variables  $v$ , constants  $c$  and sets  $s$ . A machine contains an initialisation which is a special event whose guard is always true and whose before-after predicate only refers to variables in the after-state, and it is used to assign an initial value to each variable  $v$  of the machine. In general, it takes the following form:

**Event** *Initialisation*  $\hat{=}$  **when**  $\top$  **then**  $v :| RI(c, s, v')$  **end**

The set of events  $e$  are the behavioural entities of a machine. An event may contain parameters  $p$ , guards  $G$  and actions  $R$  and it takes the general form:

**Event**  $e_1$   $\hat{=}$  **any**  $p$  **where**  $G(c, s, p, v)$  **then**  $R(c, s, p, v, v')$  **end**

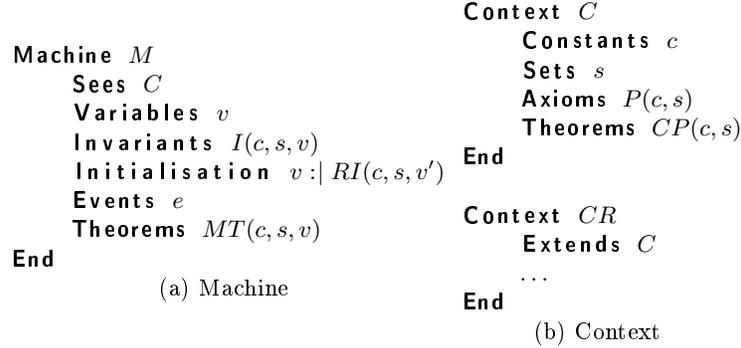
The assignment of actions can be any of the three forms described in Table 1. The non-deterministic assignment with before-after predicate is the most general form and will be used for our models and proof obligations. A machine may contain a list of theorems  $MT$ , which are predicates that follow from the invariants and any previously declared theorems.

Event-B also supports machine refinement, however in this paper we are only concerned with single machine components that do not have refinement.

An Event-B context consists of a list of typed constants  $c$ , sets  $s$  and axioms  $P$ , which are used to describe certain properties. A list of theorems  $CT$  are properties that can be proved and must follow from the axioms and any previously declared theorems. Contexts can be extended using the keyword *Extends*.

An Event-B model has proof obligations associated with it. A comprehensive list of all proof obligations is given in [3]. We present the feasibility *FIS* and invariant preservation *INV PO*, as they are the main POs discussed in this

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



**Fig. 4.** Event-B Structure

paper. The *FIS* PO denotes that under the properties  $P(c, s)$ , the list of invariants  $I(c, s, v)$  of a machine and the guard  $G(c, s, p, v)$  of an event, there exists at least one after-state satisfying  $R(c, s, p, v, v')$ . This PO has to be discharged for each event, consisting of a guard  $G(c, s, p, v)$  and an action  $R(c, s, p, v, v')$ , with a non-deterministic assignment. The *INV* PO, which is discharged for each invariant  $i(c, s, v)$  of the list of invariants  $I(c, s, v)$  and for each event  $e$  with guard  $G(c, s, p, v)$  and action  $R(c, s, p, v, v')$ , denotes that if the properties  $P(c, s)$ , the list of invariants  $I(c, s, v)$  and the guard  $G(c, s, p, v)$  of an event are true, and an after-state  $R(c, s, p, v, v')$  exists, then the invariant  $i(c, s, p, v')$  is also true in the after-state (note that to prove that the invariant is still true in the after-state we can assume all invariants in the pre-state).

FIS	$P(c, s) \wedge I(c, s, v) \wedge G(c, s, p, v) \Rightarrow \exists v' \cdot R(c, s, p, v, v')$
INV	$P(c, s) \wedge I(c, s, v) \wedge G(c, s, p, v) \wedge R(c, s, p, v, v') \Rightarrow i(c, s, v')$

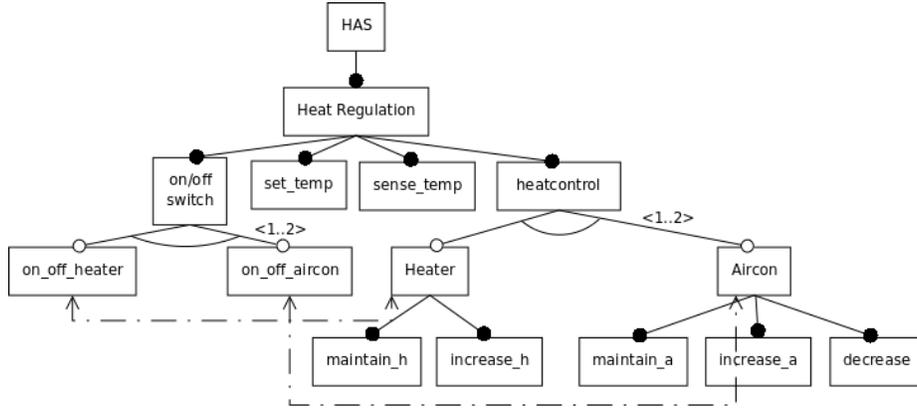
### 3 Feature Models and Event-B Modelling Patterns

In this section we show how we can model an SPL using standard feature modelling notation and then map it to Event-B models. An Event-B model that represents a feature in the feature model is referred to as *component*. A component should be independent of other components, it may see zero or more contexts, and represents some part of the program functionality.

#### 3.1 Mapping Features to Components

We present a feature model of the heat regulation subsystem (HRS) of a home automation system and how we map features to Event-B components.

The feature model in Figure 5 represents the HRS. The HRS may consist of either a heater, aircon or both. The feature *on/off switch* has an OR feature group, which allows an on/off switch to be selected for either heater, aircon



**Fig. 5.** Heat Regulation Subsystem

or both. The selection of *on\_off\_heater* includes the selection of the *Heater* features, and *on\_off\_aircon* requires the selection of *Aircon* (as indicated by the *includes* dashed line). The HRS allows the user to set a certain temperature, represented by *set\_temp*, and the system can sense the room temperature, represented by *sense\_temp*. The *heatcontrol* is broken up into two groups, the *Heater* and *Aircon* heat controls, of which either one or both must be selected. The heater can maintain or increase the temperature, represented by *maintain\_h* and *increase\_h* respectively. The aircon also has this functionality plus the ability to decrease the temperature, represented by *decrease*.

These features must now be mapped to Event-B components. When mapping features to components, only leaf nodes are mapped, other nodes are only intermediary and used for structure. The mapping occurs by name. This means that the feature *increase\_h* will be mapped to an Event-B component called *increase\_h*. This component can be viewed in Figure 6.

### 3.2 Modelling of User-defined Types

User-defined types are specified in the context of an Event-B model. Each component may have a context, this means that several contexts might contain the same user-defined type. In our example, all devices are of type *DEVTYPE*. This means, that contexts *AIRCON*, *HEATER* and *TEMPSENSOR* use this type. We define an abstract context *abstract\_context*, which contains all common user-defined types. In this abstract context, all user-defined types must be declared as deferred sets. This context can be extended by the other contexts of the product line. These may then use all user-defined types defined in the abstract context.

In Figure 6 we show both increase components for the heater and aircon devices. Some of the detail of the components is removed and replaced by  $\langle \dots \rangle$  for

illustration purposes. Both components see a different context, *HEATER* and *AIRCON*. These contexts both extend the abstract context *abstract\_context*, which contains the common type *DEVTYPE*.

<i>abstract_context</i>	<i>increase_h</i>	<i>increase_a</i>
	<pre> Context HEATER   Extends abstract_context   Constants HH   Axioms HH ∈ DEVTYPE End  Machine increase_h   Sees HEATER   (...) Event increase_h ≐   where     sensed = TRUE     desired &gt; sensed_temp     enabled(HH) = TRUE   then     sensed, heat :        sensed' = FALSE ∧       heat' = heat + 1   end End </pre>	<pre> Context AIRCON   Extends abstract_context   Constants AC   Axioms AC ∈ DEVTYPE End  Machine increase_a   Sees AIRCON   (...) Event increase_a ≐   where     sensed = TRUE     desired &gt; sensed_temp     enabled(AC) = TRUE   then     sensed, heat :        sensed' = FALSE ∧       heat' = heat + 1   end End </pre>
<pre> Context abstract_context   Sets DEVTYPE End </pre>		

Fig. 6. Components and Abstract Types

## 4 Composition Process

The composition process is the process of selecting an instance from the feature model, discharging some POs about this instance and then constructing a complete Event-B model by following composition rules. In this section we discuss how an instance is created and which composition POs can be discharged. This will be demonstrated by the HRS case study.

### 4.1 Creation of an Instance

An instance of a feature model is a specific configuration of features conformed to the constraints. Once the feature model of a product line has been developed, it can be used to create instances, i.e. product line family members. The instance can be validated by the constraints of a feature model; this validation method of feature model instances has been shown in [8].

The construction of a composite model entails the extension of contexts and the merging of machines. Machines are merged according to the composition rules presented in this section.

The composition rules for the merging of two machines are summarised in Figure 7. A type check of common variables must be performed and only if the type of common variables is the same, the construction of a composite model can proceed.

**Variables**  $v_1 \cup v_2$   
**Invariants**  $I_1(c_1, s_1, v_1) \cup I_2(c_2, s_2, v_2)$   
**Initialisation**  $v_3 := | RI_1(c_1, s_1, v'_1) \wedge RI_2(c_2, s_2, v'_2)$   
**Event**  $e_1 \cup e_2$   
**Theorems**  $MT_1(c_1, s_1, v_1) \cup MT_2(c_2, s_2, v_2)$

**Fig. 7.** Composition Rules

The variables clauses of both machines are merged into  $v_3$ , where  $v_3 = v_1 \cup v_2$ . The invariant lists of both machines are also merged into one list. The initialisation assignment predicates  $RI_1$  and  $RI_2$  are conjoined. In order to do so, the initialisation clauses must be transformed into a non-deterministic assignment. The composite events clause is the union of the events of machine  $M_1$  and  $M_2$ . Finally, the theorems are also the union of the theorems of machine  $M_1$  and  $M_2$ .

## 4.2 Proof Obligations

Before performing the construction of a complete Event-B model which reflects the selected instance, it is useful to prove certain properties about the composite model. In general, an Event-B machine has POs associated with it, and a composition also has POs associated with it. In order to avoid later confusion, we will refer to the POs associated with single machines as *component proof obligations* and those that are associated with a composition are referred to as *composition proof obligations*.

Composition POs are an extension of component POs. We are able to simplify the composition POs or show that no reproof for the composition is required based on what is already known from discharging the component POs. The composition POs are important because they eliminate a lot of reproof of component POs after two components have been merged.

For the following POs, we assume two Event-B components, where all parts of the component  $M_1$  are subscripted with 1, and all parts of component  $M_2$  are subscripted by 2. The components can contain common variables.

The first two composition POs presented have to be discharged when copying an event  $e$  from component  $M_1$  or  $M_2$  into the composition of  $M_1$  and  $M_2$ . Event  $e$  consists of guard  $G^e(c, s, v)$  and non-deterministic assignment  $R^e(c, s, v, v')$ . We assume that all component POs have been discharged for  $M_1$  and  $M_2$ .

The **FIS** PO asserts that if the context properties, the invariants and the guards for an event are true, it follows that there exists an after-state to the event. In the case of composition, this means that if the context properties and invariants for both components and the guard of the event being composed are true, it follows that there still is an after-state to the event.

The following theorem shows that there is no composition PO that has to be proved. In other words, to show the feasibility of an event in a composition does not require discharging further POs.

**Theorem 1 (Feasibility in Composition).** *Feasibility of each component event is maintained by composition.*

*Proof.* We assume *FIS*.

$$P_1(c_1, s_1) \wedge I_1(c_1, s_1, v_1) \wedge G_1^e(c_1, s_1, v_1) \Rightarrow \exists v_1 \cdot R_1^e(c_1, s_1, v_1, v_1') \quad (1)$$

We can derive a PO that represents *FIS* during the composition of two components.

$$P_1(c_1, s_1) \wedge P_2(c_2, s_2) \wedge I_1(c_1, s_1, v_1) \wedge I_2(c_2, s_2, v_2) \wedge G_1^e(c_1, s_1, v_1) \Rightarrow \exists v_1 \cdot R_1^e(c_1, s_1, v_1, v_1') \quad (2)$$

Equation 2 follows from Equation 1 by strengthening of the antecedent of the implication and thus does not have to be reproved.

The invariant preservation PO **INV** asserts that if the properties of the context are true, and the invariants are true and if there was a transition from one state to another triggered by an event, then it follows that the invariants remain true in the after-state. This component PO can be extended for composition of two components. In this case, we have to prove that after an event has occurred, the invariants of both components have to be true in the after-state.

**Theorem 2 (Invariant Preservation in Composition).** *Events of  $M_1$  (respectively  $M_2$ ) maintain the invariants of  $M_1$  (respectively  $M_2$ ) in the composition.*

*Proof.* We will prove whether the events of  $M_1$  maintain the invariants of  $M_1$  then it will also maintain them in the composition. The proof for  $M_2$  is analogous and therefore will be omitted. We can assume for  $M_1$ :

$$P_1(c_1, s_1) \wedge I_1(c_1, s_1, v_1) \wedge G_1^e(c_1, s_1, v_1) \wedge R_1^e(c_1, s_1, v_1, v_1') \Rightarrow i_1(c_1, s_1, v_1') \quad (3)$$

In the composite machine:

$$P_1(c_1, s_1) \wedge P_2(c_2, s_2) \wedge I_1(c_1, s_1, v_1) \wedge I_2(c_2, s_2, v_2) \wedge G_1^e(c_1, s_1, v_1) \wedge R_1^e(c_1, s_1, v_1, v_1') \Rightarrow i_1(c_1, s_1, v_1'), \quad (4)$$

where  $i_1$  is an invariant from the list of invariants of  $M_1$ . This is a logical consequence from our assumption given it is a strengthening of the antecedent of the implication in the assumption (in Equation 3).

However, in the composite machine it remains to be proved that the events of  $M_1$  maintain the invariants of  $M_2$  and vice versa. Therefore, the following PO needs to be discharged for each event  $e$  of  $M_1$  and all invariants  $i_2(c_2, s_2, v_2')$  of  $M_2$ . There is an analogous PO for events of  $M_2$  and invariants of  $M_1$ .

<b>INV</b> <b>COMP</b>	$P_1(c_1, s_1) \wedge P_2(c_2, s_2) \wedge I_1(c_1, s_1, v_1) \wedge I_2(c_2, s_2, v_2) \wedge G_1^e(c_1, s_1, v_1) \wedge R_1^e(c_1, s_1, v_1', v_1) \Rightarrow i_2(c_2, s_2, v_2')$
------------------------	--

The next two POs are concerned with the feasibility of the initialisation. It is clear, that during composition it is possible that common variables might be initialised to disjoint sets of values. This means that the feasibility of the initialisation during composition is not maintained, and results in a composition PO *INI\_FIS\_COMP*.

$$\boxed{\text{INI\_FIS\_COMP} \mid \begin{array}{l} P_1(c_1, s_1) \wedge P_2(c_2, s_2) \\ \Rightarrow \exists v'_1, v'_2. RI_1(c_1, s_1, v'_1) \wedge RI_2(c_2, s_2, v'_2) \end{array}}$$

**Theorem 3 (Invariant Preservation for Initialisation in Composition).**  
*The invariant is preserved during the initialisation in composition.*

*Proof.* We assume that *INI\_INV* was discharged for each invariant  $i_1$  of machine  $M_1$  for the initialisation and similarly in  $M_2$ .

$$P_1(c_1, s_1) \wedge RI_1(c_1, s_1, v'_1) \Rightarrow i_1(c_1, s_1, v'_1) \quad (5)$$

From these component POs, we can derive the following PO:

$$P_1(c_1, s_1) \wedge P_2(c_2, s_2) \wedge RI_1(c_1, s_1, v'_1) \wedge RI_2(c_2, s_2, v'_2) \Rightarrow i_1(c_1, s_1, v'_1) \quad (6)$$

This composition PO is trivially discharged because it is a strengthening of its counterpart for a single machine. Therefore the PO does not need to be re-proved for the composition. There is a similar proof for machine  $M_2$ .

### 4.3 Demonstration by Case Study

We demonstrate the composition process by extracting a small example from the HRS case study. We show the composition process of *sense\_temp*, and *decrease*. The *sense\_temp* component is shown in Figure 8(a) and *decrease* is shown in Figure 8(b). The contexts of both components extend the *abstract\_context* shown in Figure 6.

We now compose the two components *sense\_temp* and *decrease*. For this composition, we can discharge certain composition POs. We have to discharge *INV\_COMP* for events *sense\_temp* and *decrease*. We are able to discharge it for *sense\_temp* as we can prove that the invariants of component *decrease* are true in the after-state of event *sense\_temp*. We can also discharge this PO for *decrease*, by showing that the invariants of component *sense\_temp* are true in the after-state of event *decrease*. Similarly, we can discharge, the composition PO *INI\_FIS\_COMP* for the initialisation clauses of both machines. The composite model can now be constructed, which is shown in Figure 9.

## 5 Related Work

Even though a lot of work has been done in the area of composition, only A-style composition [9] and B-style composition [10] are directly related to our work.

```

Context TEMP
  Extends abstract_context
  Constants TSR
  Axioms TSR ∈ DEVTYPE
End

Machine sense_temp
  Sees TEMP
  Variables enabled, sensed_temp, sensed
  Invariants
    enabled ∈ DEVTYPE → BOOL
    sensed_temp ∈ ℤ
    sensed ∈ BOOL
    sensed_temp ≥ 0
  Initialisation ≐
    begin
      enabled, sensed_temp, sensed :|
      enabled' = DEVTYPE × {TRUE} ∧
      sensed_temp' = 10 ∧
      sensed' = FALSE
    end
  Event sense_temp ≐
    any tt
    where
      tt ∈ 10..30
      sensed = FALSE
      enabled(TSR) = TRUE
    then
      sensed_temp, sensed :|
      sensed_temp' = tt ∧
      sensed' = TRUE
    end
End
(a) Event-B sense_temp Component

Context AIRCON
  Extends abstract_context
  Constants AC
  Axioms AC ∈ DEVTYPE
End

Machine decrease
  Sees AIRCON
  Variables
    enabled, sensed_temp,
    desired, heat, sensed, enabled
  Invariants
    enabled ∈ DEVTYPE → BOOL
    sensed_temp ∈ ℤ
    desired ∈ ℤ
    heat ∈ ℤ
    sensed ∈ BOOL
  Initialisation ≐
    begin
      enabled, sensed_temp,
      desired, heat, sensed, enabled :|
      enabled' = DEVTYPE × {TRUE} ∧
      sensed_temp' = 10 ∧
      desired' = 10 ∧
      heat' = 0 ∧
      sensed' = FALSE
    end
  Event decrease ≐
    where
      sensed = TRUE
      desired < sensed_temp
      enabled(AC) = TRUE
    then
      sensed, heat :|
      sensed' = FALSE ∧
      heat' = heat - 1
    end
End
(b) Event-B decrease Component

```

Fig. 8. Event-B *sense\_temp* and *decrease* Components

Both composition styles were presented as ways to compose machines which were previously parts of the same machine and through decomposition (a framework for divide-and-conquer in formal methods) became independent of each other. Composition, in this case, provides a way to inverse the process. A-style [9] decomposition splits a machine into two, thereby introducing the notion of external variables for common variables, and external events which “mimic” the usage pattern of external variables in decomposed machines. This decomposition method requires a simple composition procedure that joins the decomposed machines by merging the variables sets (and removing duplicates) and removing the external events. In this way, not only the machine can be recomposed but it is also proved that this recomposed machine refines the machine before it was decomposed. B-style decomposition [10], similar to A-style decomposition, splits a machine into two. In the same way the author defined a procedure to recompose the machines. The composition procedure joins two machines where these may not have common state variables and may synchronise on common events. Our work differs from this work in that the defined composition procedure is not a way to recompose a machine that was previously decomposed. Even though the composition procedure was presented in a framework to integrate SPLs and Event-B, it can be used on its own. We provide not only the composition rules

```

Machine instance
  Sees AIRCON, TEMP
  Variables enabled, sensed_temp,
            desired, heat, sensed, enabled
  Invariants enabled ∈ DEVTYPE → BOOL
            sensed_temp ∈ ℤ
            desired ∈ ℤ
            heat ∈ ℤ
            sensed ∈ BOOL

  Initialisation ≐
  begin
    enabled, sensed_temp,
    desired, heat, sensed :|
    enabled' = DEVTYPE × TRUE ∧
    sensed_temp' = 10 ∧
    desired' = 10 ∧
    heat' = 0 ∧
    sensed' = FALSE
  end
Event decrease ≐
  where
    sensed = TRUE
    desired < sensed_temp
    enabled(AC) = TRUE
  then
    sensed, heat :|
    sensed' = FALSE ∧
    heat' = heat - 1
  end
Event sense_temp ≐
  any tt
  where tt ∈ 10..30
    sensed = FALSE
    enabled(TSR) = TRUE
  then sensed_temp, sensed :|
    sensed_temp' = tt ∧
    sensed' = TRUE
  end
End

```

(a) Composite Event-B Machine

```

Context abstract_context
  Sets DEVTYPE
End

Context AIRCON
  Extends abstract_context
  Constants AC
  Axioms AC ∈ DEVTYPE
End

Context TEMP
  Extends abstract_context
  Constants TSR
  Axioms TSR ∈ DEVTYPE
End

```

(b) Composite Event-B Context

Fig. 9. Composition of *sense\_temp* and *decrease* Components

but also show which POs need to be discharged to ensure that the composed machine is still consistent.

## 6 Conclusions

In this paper we have presented a novel method for integrating feature models with Event-B. In this way we enable SPL development for formal methods and provide a way to prove certain properties about a composition. We introduced some Event-B modelling patterns that assist the composition of contexts. Several composition POs were derived from existing component POs and allow the user to prove certain properties about a composition before constructing the composite model. The composite model is then constructed according to the instance selected in the feature model. This is done by following a number of composition rules.

Currently, our work is fundamentally theoretical, however we have been collaborating in the development of a Rodin plugin to integrate this theoretical approach with the Rodin platform. This plugin contains a composition tool, and in future will support feature model editing and an instance generator [11,12].

This composition tool detects naming conflicts (e.g. duplicate variable names) in Event-B models which are composed. It allows the user to manually edit these. The tool currently does not have the option of discharging composition proof obligations, however the goal is to integrate these into the tool.

We have proposed the composition of Event-B components without refinement. Future work will see the composition of refinement trees, which will also require the extension of feature modelling notation to represent refinement graphically.

## References

1. van Ommering, R., Bosch, J.: Widening the Scope of Software Product Lines - From Variation to Composition. (2002) 31–52
2. Czarnecki, K., Eisenecker, U.W.: Generative programming: methods, tools, and applications. ACM Press/Addison-Wesley Publishing Co. New York, NY, USA (2000)
3. Abrial, J.: Modeling in Event-B: Systems and Software Engineering. To be published by Cambridge University Press (2009)
4. Kang, K., Lee, J., Donohoe, P.: Feature-Oriented Project Line Engineering. *IEEE Softw.* **19**(4) (July 2002) 58–65
5. Sipka, M.: Exploring the commonality in feature modelling notations. *Proceedings of IIT* (2005)
6. Abrial, J.: The B-book: assigning programs to meanings. Cambridge University Press (1996)
7. Butler, M., Hallerstede, S.: The Rodin Formal Modelling Tool. BCS-FACS Christmas 2007 Meeting - Formal Methods In Industry, London. (December 2007)
8. Sun, J., Zhang, H., Wang, H.: Formal Semantics and Verification for Feature Modeling. In: *ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS'05)*, Washington, DC, USA, IEEE Computer Society (2005) 303–312
9. Abrial, J., Hallerstede, S.: Refinement, decomposition, and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae* **77**(1) (2007) 1–28
10. Butler, M.: Decomposition Structures for Event-B. In: *Proceedings of the 7th International Conference on Integrated Formal Methods*, Springer (2009) 20–38
11. Poppleton, M., Fischer, B., Franklin, C., Gondal, A., Snook, C., Sorge, J.: Towards Reuse with Feature-Oriented Event-B?. (2008)
12. Gondal, A., Poppleton, M., Snook, C.: Feature composition-towards product lines of event-B models. (2009)