Towards Feature-Oriented Specification and Development with Event-B

Michael R. Poppleton

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

Abstract. A proposal is made for the development of a feature-oriented reuse capability for safety-critical software construction using rigorous methods. We précis the Event-B language - the evolution of the B-Method of J.-R. Abrial [1] - a leading formal method for safety-critical software development. Current and new infrastructure for scalable development with Event-B is outlined, and contrasted with support required for feature-oriented development. The proposal is illustrated by a small example of feature-oriented construction and refinement with Event-B.

1 Introduction

1.1 Background and Rationale

We will introduce this paper with a little history of the development of our interest in applying feature-orientation to a formal development method.

Our ongoing work in the current EU project RODIN[1] [22] illustrates a product-line approach to the rigorous engineering of structural generic requirements for a subsystem - failure management and detection - of aircraft engine control. An avionics control system represents - as do its support systems - a *software product line* [19], that is where multiple variants of essentially the same software system are required, to meet a variety of platform, functional, or other requirements. This is moreover a safety-critical product line, motivating the use of the most rigorous methods available, in our case, the B [1] and Event-B [20] methods of J.-R. Abrial.

Event-B is a state-based language for the specification and refinement-based development of a system model, with automated verification built in to the process. It represents the new generation of the classical B language of J.-R. Abrial [1]. Its syntax and semantics are rigorously defined, enabling the automatic production of correctness verification conditions (or *proof obligations*) that can be discharged with theorem prover support. The Event-B language and its comprehensive tooling environment - including *inter alia* project database, syntax analyser, provers, animators, a test case generator - are under production in project RODIN.

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

Our RODIN industrial partner's failure management system (FMS) is a product line of a particular kind: each airframe is characterized by its sensor and actuator fit, their physical and operating range characteristics, and failure detection procedures. Each such system *configuration* can be described as an instance of a single generic configuration model that describes the structural constraints each instance must satisfy. For example, each failure test relates to at least one sensor, each test operates under at least one dynamic condition on system state, each sensor has a defined operating range, etc.

In general the critical system product line will manifest significant *commonalities* and *variabilities* [10] of behaviour and configuration. Considering the FMS application in this light, it became clear that the initial, abstract model was made up of four requirements *features*, or *goals* (in the sense of van Lamsweerde [23]): (i) *detection* of a sensor failure, (ii) failure *confirmation* (reducing sensitivity to noise), (iii) applying detection only under the appropriate *condition*, and (iv) taking appropriate *action* on detection of evidence of failure. These feature models are distinct in requirements terms but interact in terms of shared variables and events. In a product line setting they can be instantiated, combined and refined in various ways. They could be reused in various combinations in different contexts within or even beyond the FMS domain.

One conclusion of our project experience is that a feature-oriented approach would have a clear value in managing reuse and instantiation in the rigorous construction of an FMS, and thus other safety-critical software product lines.

Since the early work on features, e.g. [15], feature-oriented approaches have become prominent contributors to software reuse [11,7], especially for product lines [16].

In a longstanding annual conference series [4,12,21], formal verification techniques have been extensively examined for the feature interaction problem, originally arising in telecommunications. Beyond that there is some evidence of formal verification (as opposed to construction) techniques being applied to feature-oriented development. Feature models defined with differing degrees of genericity, binding into the software construction process at different points, have been validated formally [24]. Formal feature model-checking [op.cit.] and product line architectural model-checking for commonalities of robustness and fault-tolerance have been applied [18]. However, formal refinement-based approaches - in the classical sense of Hoare, He, Back et al [13,6] - largely remain to be applied either to feature-oriented development or to software product lines.

1.2 Formal Feature-Orientation

The mechanism for large-scale structuring in Event-B, similar to that of other model-based formal methods, is decompositional: in Fig. 1(b) a single, "abstract" model M is developed and decomposed into components $\{f_i\}$. The components are refined to more "concrete" form $\{fr_i\}$ and these concrete refinements are then recomposed into model MR in a *particular way* that guarantees that MR refines M. This process is repeated at subsequent refinement steps. Section 2.3 will show that this is a complexity management mechanism for specifications, not concerned with requirements or feature engineering.

In this work we propose a compositional method for feature-oriented working with Event-B, as shown in Fig. 1(a). The atomic unit of modelling, and starting point of

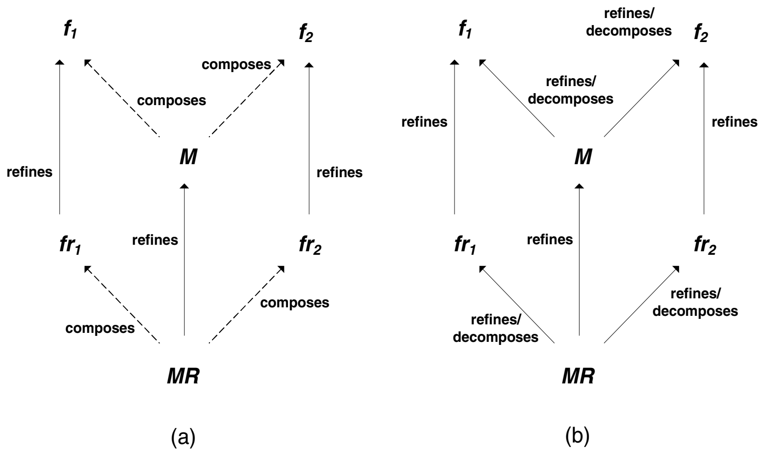

Fig. 1. Composition and decomposition of models through refinement

specification work, will be the feature. To go with existing mechanisms for the specialization (or *instantiation*) and refinement of generic features, we propose a mechanism of feature *composition*. This is a more general process than the inversion of decomposition: we seek a method to compose features $\{f_i\}$ into a composite M, which is monotonic with respect to the composition of their feature refinements $\{fr_i\}$ into composite MR - that is, MR must then refine M (Fig. 1(a) must commute).

Our contribution is thus a statement of requirements for a set of tool-implemented, syntactic transformations for feature instantiation and composition with Event-B. Also, we present a simple vending machine product line development as a case study analysis that generated these requirements. We will refer back to the decomposition mechanism of Event-B because our broad proposal is essentially a generalized inversion of it. Whilst fully enabling the use of verification capabilities of Event-B, the proposal is only the precursor of the semantic work necessary to establish the full benefits of reuse, such as

– the propogation of proven feature correctness properties through composition,
– the discovery of *particular ways* of doing composition of concrete feature refinements to *guarantee* commutatitivity of Fig. 1(a).

Section 2 introduces the Event-B language and briefly describes its two mechanisms for scalable development. Section 3 presents a small example feature-oriented development to demonstrate what can be done in feature terms with the existing CSP and Event-B notations. Section 4 presents the proposal for tool-supported feature composition in Event-B. In conclusion the proposal is restated, re-examined in relation to the existing decomposition of refinement mechanism, and further work is discussed.

2 The Event-B Language and Method

This section is a précis of parts of [20], the Event-B language definition.

2.1 Basics

Event-B is designed for long-running *reactive* hardware/software systems that respond to stimuli from user and/or environment. The set-theoretic language in first-order logic (FOL) takes as semantic model a transition system with guarded transitions between states. The correctness of a model is defined by an invariant property, i.e. a predicate, or constraint, which every state in the system must satisfy. More practically, every event in the system must be shown to preserve this invariant; this verification requirement is expressed in a number of *proof obligations* (POs). In practice this verification is performed either by model checking or theorem proving (or both).

To date, classical B verification tools in use at Southampton have been mainly

- ProB [17], the model-checker for B developed at Southampton and Düsseldorf. ProB syntax checks, animates, and model checks B models and combined B+CSP models. It also provides refinement-checking for B, B+CSP models of two varieties: trace refinement and singleton-failures refinement.
- B4free [9], a prover originally from ClearSy, the authors of the commercial AtelierB [2] toolkit.

A new integrated toolset for Event-B is under construction in project RODIN.

In Event-B the two units of structuring are the *machine* of dynamic variables, events and their invariants, and the *context* of static data of sets, constants and their axioms. Every machine *sees* at least one context.

The unit of behaviour is the *event*. An event E acting on (a list of) state variables v, subject to enabling condition, or *guard* $G(v)$ and *generalized substitution*, or *assignment* $R(v)$, has syntax

$$E \mathrel{\widehat{=}} \text{SELECT } G(v) \text{ THEN } R(v) \text{ END} \tag{1}$$

That is, when the state is such that the guard is true, this enables the state transition defined by $R(v)$, known as a generalized substitution because it denotes a nondeterministic transition. Next we give syntax for a such a substitution, or assignment $R(v)$ and its semantic model in a before-after predicate. Note that t, v are in general variable lists.

$$\text{ANY } t \text{ WHERE } Q(t,v) \text{ THEN } v := F(t,v) \text{ END} \tag{2}$$

$$\exists t \bullet (Q(t,v) \wedge v' = F(t,v)) \tag{3}$$

This defines a t-indexed nondeterministic choice between those transitions $v' = F(t,v)$ for which $Q(t,v)$ is true[2]. t is intrepreted as an input from the environment. Syntactic sugar is available: CHOICE is used for an explicit choice between a small number of assignments, and parallel ($||$) is used to enumerate single-variable assignments. Examples appear in section 3.2.

An event E works in a model (comprising a machine and at least one context) with constants c and sets s subject to *axioms* (properties) $P(s,c)$ and an *invariant* $I(s,c,v)$. Thus the event guard G and assignment with before-after predicate R take s, c as parameters. Two of the consistency proof obligations [3] (POs) for event E defined as (1)

[2] The deterministic assignment is simply written $v := F(v)$.

[3] See [20] for the others.

are FIS (feasibility preservation) and INV (invariant preservation):

$$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \Rightarrow \exists\, v' \bullet R(s,c,v,v') \tag{4}$$

$$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \wedge R(s,c,v,v') \Rightarrow I(s,c,v') \tag{5}$$

2.2 Refinement

In order to progress towards implementation, the process of *refinement* is used. The term *refinement* is used both to refer to the process of transforming models, and to the more concrete model which refines the abstract one. A refinement is a (usually) more elaborate model than its predecessor, in an eventual *chain* of refinements to code; see Fig. 2[4].

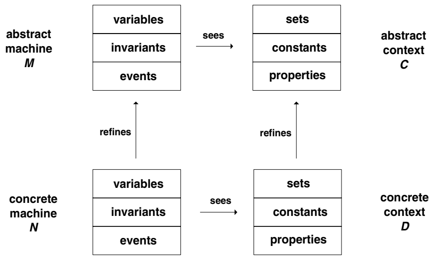

Fig. 2. Machine and context refinements

The refinement of a context is simply the addition of new sets, constants and axioms to it.

To refine a machine, all variables v are replaced by new ones w, some simply by renaming - i.e. of the same type and meaning - and others by variables of different type. For example, a set variable s might be refined to a sequence ss, thus adding the concrete structure of ordering. Existing events are transformed to work on the new variables. New events can be defined; that is, the behaviour of an abstract event E can be refined by some sequence of E and new events. The new behaviour will usually reduce nondeterminism; for example, nondeterministic selection from the set s is refined by the sequence of events first; $\mathit{first}(ss)$ to get the second element from the sequence.

When model $N(w)$ refines $M(v)$, it also has an invariant $J(s,c,v,w)$ which can include M's variables v. This is called a "gluing invariant" and has the function of relating abstract variables v to concrete ones w mathematically. Following the above example, $J(s,ss) \mathrel{\widehat{=}} s = ran(ss)$.

In Fig. 2, M sees C, N refines M and D refines C, then N sees D. It is also possible for C not to be refined, in which case N sees C.

As for simple machines, there are proof obligations for refinement; we just present one here. We assume axioms $P(s,c)$, and abstract, concrete invariants $I(s,c,v)$ and $J(s,c,v,w)$ respectively. An abstract event with guard $G(s,c,v)$ and before-after predicate $R(s,c,v,v')$ is refined by a concrete event with guard $H(s,c,w)$ and before-after

[4] Figure from [20].

predicate $S(s,c,w,w')$. The main refinement obligation INV_REF states that any before-state pair (v,w) related through J where w steps to w' through S, is matched by some J-related (v',w') where v steps to v' through R:

$$\begin{aligned} &P(s,c) \wedge I(s,c,v) \wedge J(s,c,v,w) \wedge H(s,c,w) \wedge S(s,c,w,w') \\ &\Rightarrow \exists\, v' \bullet (R(s,c,v,v') \wedge J(s,c,v',w')) \end{aligned} \tag{6}$$

2.3 Structuring Mechanisms

We complete this account of Event-B by outlining its two structuring mechanisms: generic instantiation and decomposition of refinement.

Generic Instantiation. Here, a prior development $\{(M_i, C_i)\}$ (of machines, refinements, and contexts) is treated as *generic*. This is a mechanism of substitution of identifiers in the generic development with those of the development in hand, say machine N and context D. The substitution must be proved to satisfy the axioms of the generic development.

At its simplest such generic instantiation enables direct substitution of identifiers in generic contexts with specific data from the development in hand. More generally it allows, at a point in a development when a refinement is sought, a library of generic developments to be searched for a candidate. The generic candidates can differ in the identity of static data, provided the development provides at least matching static data structure and axioms; it may in general provide more than that. Generic instantiation should be a valuable supporting mechanism for instantiation in software product lines.

Decomposition of Refinement. The approach of decomposition [3] in Event-B is the inverse of the usual compositional approach in software design and programming. The motivation is an engineering one, to decompose the design of a single model and its refinement into a number of smaller components and component refinements. Correspondingly, each proof task should be smaller, thus more capable of automatic proof.

In section 1.2 we saw that the decompositional approach of Fig. 1(b) is interpreted as a commuting diagram, provided the decomposition and component refinements are done in the right way. This is very much a matter of structural (i.e. model and refinement) engineering, rather than the inverse of some feature-oriented, or compositional requirements structuring method.

3 An Example Development

We choose as a small illustrative example a product family of vending machines - Fig.3 below gives a feature model. Possible variabilities between machines include

Payment mode: traditional coin, credit card, smartcard, or no payment - free items from a generous employer - are four options. The first three may appear in any combination on a machine.

Delivery mechanism: the usual item delivery is *sequential* - an array of horizontal racks facing the user. The user chooses an item (rack) number. An alternative is *carousel*, where items are shelved in a single, vertically mounted circular rack facing the user. Here, the next item for delivery is predetermined by the contents of the rack.

Extending the domain to say, drinks vending, would extend the variability here.

A key technique offered by Event-B for variability specification is refinement. Ideally, a single abstract, generic model describes the essential, common goals of all instances of the product line, and thus incorporates all variabilities. In practice it will not always be possible to abstract to this extent; certain features may be optional in the most abstract model.

For the vending machine, three generic features are composed to form the abstract model: item selection and inventory features are composed either with or without an optional payment feature. For each of the resulting two abstract - we shall call them level 0 - models, a tree of refined models introduces the variabilities in all meaningful combinations. These combinations are defined by the feature model. The above two variabilities, representing implementation technology choices, can be introduced through refinement in this way.

Figure 3 gives the feature model in the style of [5], which work includes a tool *FeaturePlugin* for feature modelling and product line feature-oriented system instantiation. Since the tool is agnostic as to implementation language, it is principle deployable for a future feature-oriented Event-B. Each box in the figure represents a model of a single feature in Event-B. The vending machine comprises features for

– **select/cancel:** user selection of an item/cancellation of selection (mandatory)
– **payment/clear:** accepting payment/clearing payment (which may involve giving change or returning money) (optional)
– **deliver/reload:** item delivery/ machine reload, i.e. stock control (mandatory)

These requirements are packaged as three features in the abstract model, in order to illustrate some of the technicalities of feature-oriented specification in B. Each feature consists of two events named as per the feature name, and supporting data.

deliver/reload is refined to either sequential or carousel (we might call this an *alternative refinement*). **pay/clear** is refined by one or more of coin, smartcard, or credit card payment (called a *multiple refinement*). The constraint links at the bottom of the figure indicate that the clear feature is required in support of all three payment options.

3.1 A Behavioural View in CSP

For this discussion we instantiate feature **pay/clear** in top-level model 0. It is useful at this point to give a behavioural view of the vending machine model. This view is given in CSP [14] in Fig. 4 for illustration only in this paper[5]. The feature composition is shown in colour: **select/cancel** in green, **pay/clear** in blue, and **deliver/reload** in red[6].

[5] An integration of B and CSP exists [8] and is implemented in ProB (sec. 2.1), but is beyond the scope of this paper.

[6] This colour-coded feature marking, inspired by [11], is not part of the CSP language.

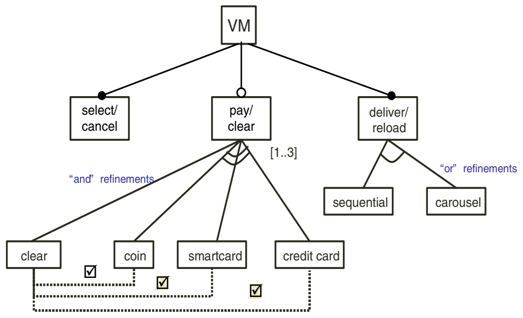

Fig. 3. Vending machine - feature model

Each event name represents an invocation of that event from the B model, which is composed from the 3 B features. The events full, itemAvail, etc. represent boolean tests on data in the B model, i.e. are communications between the two models.

```
                                              deliver ⟶  clear  ⟶ VM
                               pay ──────────⟶ □
                  select ⟶ □                  cancel  ⟶  clear  ⟶ VM
SELECT =   □              cancel ⟶ VM
          RELOAD

RELOAD =  not_full ⟶ reload ⟶ VM
          | full ⟶ VM

VM = (itemAvail ⟶ SELECT | not_itemAvail ⟶ VM)
       □
     RELOAD
```

Fig. 4. Vending machine - behavioural model in CSP

The vending machine process VM starts with a choice (□) between two options: (i) process RELOAD and (ii) a prefixed choice (|) between processes SELECT and VM, depending on whether some suitable item is available for selection. RELOAD will either reload the machine or not, depending on whether it is already full or not, and then proceed to VM. SELECT gives a choice between the selection process and RELOAD. The selection process comprises item selection, followed by payment, delivery of the item, and clearing payment/issuing change, with a cancellation option at each stage. Cancellation is of course followed by clearing payment/issuing change if payment has already been made.

A CSP model describes explicitly the possible event sequences the system might undergo. This is in contrast to the model-based, or state-based nature of B, which is designed to define atomic data transitions. While the syntax of a B event makes clear the data changes during that atomic event, allowed event sequences - or *traces* - are only implicitly and semantically defined in terms of sequences of invocations of enabled events. This behavioural nature of the CSP model gives a more direct picture of how the traces of the composite system are composed of sub-traces from the features.

Notice how in Fig. 4 it is not clear that any of the 3 features offers a desirable behavioural property: we might expect a feature to be *deadlock-free* - preventing the situation where none of its events are enabled. That is, we might expect to see, say, blue events happening without being interspersed by events of other colours (e.g. *deliver* or *cancel*). More formally we might expect to see the colours in the CSP graph restricted to strongly connected subgraphs.

While such a notion of deadlock-freedom may be attractive, it cannot be a requirement of a feature, which can offer other kinds of functional coherence. For example in **pay/clear** pay records the fact of payment being accepted, and clear abstracts over both the issuing of change, and the clearing of payment received. The two events are logically separated by the functions (provided by other features) of item delivery, or payment cancellation. We will return to this point in the following section.

Note that Fig. 4 only describes the behaviour of this particular feature composition. For the behaviour of a standalone feature, or a different composition, different CSP models are required.

3.2 Feature Specification in Event-B

Each of the 3 abstract (level 0) features is specified as a B model. It is a very abstract model, in a sense mimicing the behavioural picture of Fig. 4 by simply recording the changing state of affairs in boolean variables. More structure, data and algorithm - such as collection of payment, identification of selected item - is layered in later by refinement. Figures 5 and 6 give two partial feature definitions as partial B models for features **pay/clear** and **deliver/reload** respectively. Each feature is of course specified for reuse in settings other than the vending machine and must constitute syntactically correct B, and should be verified, in the first instance, in isolation as usual.

Machine payClear0 has two booleans *paid*, *selected* to record that the user has paid for, and selected his chosen item, respectively. The initialisation is as nondeterministic as possible to allow specialization - i.e. reduction of nondeterminism - in composition. Thus initial states appropriate to the feature in isolation may be appropriate in some compositions but not others. Here, the feature invariant allows *selected*, *paid* to be initialised nondeterministically from $\mathbb{B}$, the constant data of this abstract feature model. Since - at the level of the single feature - this is the only meaningful selection of constant data in this example, we do not use a context. In general however, a feature model will require a feature context - here, payClear0ctx, say - as well as a machine.

Provided an item has been selected but payment has not yet been made, event pay records payment in *paid*. If payment has been made, and the item is no longer selected[7], event clear records payment not made. Thus clear abstracts both over giving change where necessary, and recording the payment cleared from the system.

Figure 6 specifies feature **deliver/reload**. This B model has three boolean variables: *selected* as before, *itemAvail* to indicate the required item is available for selection, and *full* to indicate the vending machine is full. There is a little more to this invariant: if an

[7] The item can be deselected by some event outside this feature, such as deliver or cancel.

```
MACHINE           payClear0
VARIABLES         $paid, selected$
INVARIANT         $paid \in \mathbb{B} \wedge selected \in \mathbb{B}$
INITIALISATION    $paid :\in \mathbb{B} \mid\mid selected :\in \mathbb{B}$
OPERATIONS
pay =
    SELECT $paid = \text{false} \wedge selected = \text{true}$
    THEN $paid := \text{true}$
    END;
clear =
    SELECT $paid = \text{true} \wedge selected = \text{false}$
    THEN $paid := \text{false}$
    END
```

Fig. 5. Partial **pay/clear** - level 0

item is selected, it must be available, and if the VM is full then the required item must be available. Event deliver models delivery of an item. Details such as decrementing the item count are left for refinement. Provided the item required is selected and available, deliver will de-select the item, set *full* to false, and assign *itemAvail* nondeterministically. The next required item may or may not be available. Note that there is no concept of payment in this feature.

These three feature models have been model-checked with ProB, although this is of limited value because of the deadlocking that arises in each feature model as discussed in section 3.1.

```
MACHINE           deliverReload0
VARIABLES         $selected, itemAvail, full$
INVARIANT         $selected \in \mathbb{B} \wedge itemAvail \in \mathbb{B} \wedge full \in \mathbb{B}$
                  $\wedge\ (selected = \text{true} \Rightarrow itemAvail = \text{true})$
                  $\wedge\ (full = \text{true} \Rightarrow itemAvail = \text{true})$
INITIALISATION
    CHOICE $selected := \text{false} \mid\mid full := \text{false} \mid\mid itemAvail :\in \mathbb{B}$
    OR $selected := \text{false} \mid\mid full := \text{true} \mid\mid itemAvail := \text{true}$
    OR $selected := \text{true} \mid\mid full :\in \mathbb{B} \mid\mid itemAvail := \text{true}$
    END
OPERATIONS
deliver =
    SELECT $selected = \text{true} \wedge itemAvail = \text{true}$
    THEN $selected := \text{false} \mid\mid itemAvail :\in \mathbb{B} \mid\mid full := \text{false}$
    END;
reload = ...
```

Fig. 6. Partial **deliver/reload** - level 0

4 Composition of Features

We illustrate composition by giving a partial composite B model including event deliver in Fig. 7. We will define composition mechanisms that are automatable as far as possible, while supporting the creative user design input that will usually be necessary.

Note that the text-level composition of n feature models involves the composition of more than n modules: in general (unlike the example) each feature will have at least one generic context defining static data. In composing the features, other objectives may be being addressed: further information may be added (refinement of context), and/or product line specialization may be performed (generic instantiation of context).

```
MACHINE         vending0
VARIABLES       $selected, paid, itemAvail, full$
INVARIANT       $selected \in \mathbb{B} \wedge paid \in \mathbb{B} \wedge itemAvail \in \mathbb{B} \wedge full \in \mathbb{B}$
                $\wedge\ (selected = \text{true} \Rightarrow itemAvail = \text{true})$
                $\wedge\ (full = \text{true} \Rightarrow itemAvail = \text{true})$
INITIALISATION
        $paid := \text{false}\ ||$
        CHOICE $selected := \text{false}\ ||\ full := \text{false}\ ||\ itemAvail :\in \mathbb{B}$
        OR $selected := \text{false}\ ||\ full := \text{true}\ ||\ itemAvail := \text{true}$
        END
OPERATIONS
...
deliver =
        SELECT $paid = \text{true} \wedge selected = \text{true} \wedge itemAvail = \text{true}$
        THEN $selected := \text{false}\ ||\ itemAvail :\in \mathbb{B}\ ||\ full := \text{false}$
        END;
reload = ...
```

Fig. 7. Partial VM - level 0

1. Identifiers: Selection of identifiers in the composed model - machine and context - may require user input. In our example the identifiers in all three features have been chosen to harmonize variables: e.g. *selected* in payClear0 represents the same variable as *selected* in selCancel0. In general the user may need to change identifiers to harmonize on a variable, e.g. if *sel* and *selct* in two composing features represent the same variable, then rename *sel* to *selct*. Alteratively she may need to change identifiers to distinguish between variables: e.g. *paid* in payClear0 may represent a different variable from *paid* in some other feature concerned with payment.
2. Data: sets, constants, variables: All identifiers are concatenated in their respective sections of the composed model (sets, constants in context; variables in machine).
3. Constraints: axioms and invariant: These predicates are conjoined in their respective sections of the composed model (axioms in context; invariant in machine). The user may strengthen these predicates manually. The well-definedness of the composite axioms and invariant are checked by the context PO - "A context of sets and

constants exists subject to the axioms" - and the initialization PO - "The initialization establishes the invariant".

4. Initialization: Feature initialization clauses are composed - in an automatable manner - by (i) placing all variable assignments in parallel (i.e. as a variable list assignment), and within that (ii) composing multiple assignments to a single variable by intersection of transition sets. That is, by $x :: 1..5 \mid\mid x :: \{2, 4, 6\}$ ("assign to x any natural between 1 and 5, and in parallel assign to x one of 2, 4 or 6") we mean $x :: \{2, 4\}$. Suitable nondeterminism in feature initializations - supported by feature contexts - will give scope for this. In any event, the feasibility of such a composed initialization is checked in the initialization PO.

   In the example user constraints are imposed on the composed initialization: *selected*, *paid* are fixed false since a VM must start without a selection and payment.

5. Events: Distinct events are concatenated in the composite machine. Multiple instances of an event *e* from multiple features[8] are composed in the same way as multiple initialisations; these might be thought of as feature *views* of the event *e*. Where event views arise, there are two aspects to event composition:
   - Guards: The view guards are conjoined. User manual guard strengthening is permitted: in the example, deliver is strengthened with $paid = \text{true}$, required in a system with payment. Similarly, select is strengthened with $paid = \text{false}$, since selection always precedes payment in our composite model. A new guard satisfiability PO is required to check the composite guard is not vacuously false.
   - Assignments: These are composed as for initialization. User manual constraint of the composed assignment is permitted. Well-definedness of the composite assignment is verified by the event consistency PO - "This event re-establishes the invariant".

We can think of guard and invariant strengthening as forms of specialization of a simple composition of feature specifications. The feature model Fig. 3 of this composed abstract VM with payment could be annotated with an expression something like the following:

$$\begin{aligned}&(+)([\mathsf{payClear0}, \mathsf{deliverReload0}, \mathsf{selCancel0}],\\ &\quad [(\mathsf{deliver}, \mathrm{gs}, paid = \text{true}),\\ &\qquad (\mathsf{select}, \mathrm{gs}, paid = \text{false})])\end{aligned}$$

This denotes a specialization which is a function of the composition of these three features, named in the first (sequence) argument. The second argument gives the sequence of event specializations mentioned above. In the general case the specialization would include details of identifier substitutions within the composed features.

The composite model has been fully model-checked with ProB.

4.1 Towards Feature Refinement

Figure 8 shows the extent of the practical VM work to date, giving some practical confidence in this enterprise of feature-orientation in Event-B. We have ProB-model-checked the abstract models (level 0), i.e. three features and one composite VM. We

[8] In the vending machine multiple instances of an event do not arise as each event is unique to its feature. In the FMS however this does happen.

have constructed and model-checked a refinement model (level 1) for each of the three feature models and for the composite VM. We have also refinement-checked each of these four refinements. To summarize the verification completed, all models and solid-line refinements in Fig. 8 have been checked.

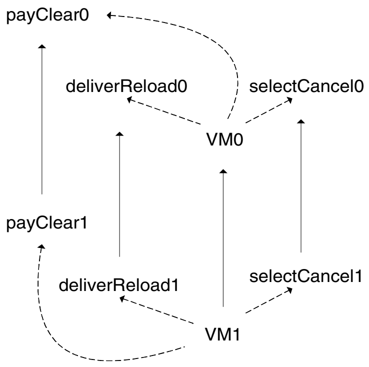

Fig. 8. Vending machine - modelling and verification

Each feature refinement model, and the composed refinement model have been constructed as before, albeit containing more concrete design structure and algorithm - space constraints prevent us elaborating here.

5 Conclusion and Further Work

Via case study experimentation we have proposed a syntactic procedure for composing feature models in Event-B. Our experiment gives some confidence that when using the procedure (i) design and compose abstract features, (ii) design and check (concrete) feature refinements, (iii) compose the concrete feature refinement models, then the composite concrete model should refine the abstract one. This is a flexible mechanism requiring tool support as suggested in sec. 4.

We next consider the extent to which our new feature composition mechanisms break the existing decomposition of refinement mechanism in Event-B, and the implications of this fact. Note that in Fig. 1(b) every line is a refinement: each component is refined by its respective composite. In our feature-compositional approach, only the vertical lines in Fig. 8 for the feature refinements (step (ii) above) are definitely refinements; ongoing theoretical work will seek guarantees that the composition mechanisms we use will produce a refinement of the composite model.

1. User-strengthening of composite axioms and invariant is problematic as it breaks the possibility of the composite model refining each feature.
2. An event guard may be manually strengthened in refinement, as we have done for deliver and select. However, refinement requires that the concrete model does not deadlock more often than the abstract one; thus if one event guard strengthens, other

events must be adapted, or new ones added in the concrete model to compensate. This remains to be investigated.

3. Similar problems arise with manual strengthening of the composition of initialisations and event assignments.
4. Composition of multiple event, or initialisation viewpoints is not defined in the decomposition of refinement mechanism. The implications of this remain to be investigated.

In summary, although there may be certain simple feature composition scenarios that are compatible with - i.e. represent the inverse of - the Event-B decomposition of refinement mechanism Fig. 1(b), in general decomposition will not be directly applicable. That is, work is required to investigate the extensibility of the mechanism to guarantee that composing feature refinements is equivalent to refining composed features. Practical case study work - as in this paper - will provide evidence of specification *patterns* that afford compositionality; this will guide the theoretical work. It is unlikely that such guarantees will emerge for the fully general procedures for feature refinement and composition that we sketch here. Theoretical results defining specification patterns that guarantee composition will serve as methodological guidance to developers, in principle whilst using tool support.

Fig. 1(a) represents the theory of refinement-preserving composition mappings that we seek. That is, given a set of features $\{f0_i\}$, each instantiated with data $\{args_i\}$ we might compose these using some mechanism $\mathsf{Comp}(args)(\{\mathsf{Inst_i}(args_i)(f_i)\})$ to give the abstract composed model *comp*0. The question is, under what conditions can this composition mechanism - or some adaptation of it - be applied to the refined features $\{f1_i\}$ in order to produce a refinement of *comp*0 ?

References

1. Abrial, J.-R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press, Cambridge (1996)
2. Abrial, J.-R., Clear, Sy.: Atelier-B (1998) `http://www.atelierb.societe.com/index_uk.htm`
3. Abrial, J.-R., Hallerstede, S.: Refinement, decomposition and instantiation of discrete models: Application to Event-B. Fundamenta Informaticae, pp. 1001–1026 (2006) (in press)
4. Amyot, D., Logrippo, L. (eds.): Proceedings FIW '03, Seventh International Workshop on Feature Interactions in Telecommunication and Software Systems, Ottawa, Canada. IOS Press, Amsterdam (2003)
5. Antkiewicz, M., Czarnecki, K.: FeaturePlugin: feature modeling plug-in for Eclipse. In: Eclipse '04: Proceedings of the 2004 OOPSLA workshop on Eclipse technology eXchange, pp. 67–72. ACM Press, New York, NY, USA (2004)
6. Back, R.J.R.: A calculus of refinements for program derivations. Acta. Informatica 25, 593–624 (1988)
7. Batory, D.: Feature models, grammars, and propositional formulas. In: Obbink, H., Pohl, K. (eds.) SPLC 2005. LNCS, vol. 3714, Springer, Heidelberg (2005)
8. Butler, M., Leuschel, M.: Combining csp and b for specification and property verification. In: Fitzgerald, J.A., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 221–236. Springer, Heidelberg (2005)

9. Cansell, D., Abrial, J.-R., et al.: B4free. A set of tools for B development (2004), from `http://www.b4free.com`
10. Coplien, J., Hoffman, D., Weiss, D.: Commonality and variability in software engineering. IEEE Software, pp. 37–45 (November/December 1998)
11. Czarnecki, K., Antkiewicz, A.: Mapping features to models: A template approach based on superimposed variants. In: Glück, R., Lowry, M. (eds.) GPCE 2005. LNCS, vol. 3676, pp. 422–437. Springer, Heidelberg (2005)
12. Gilmore, S., Ryan, M.: Language Constructs for Describing Features: Proceedings of the FIREworks Workshop. Proceedings of the FIREworks Workshop. Springer, Heidelberg (2001)
13. He, J., Hoare, C.A.R., Sanders, J.W.: Data refinement refined. In: Robinet, B., Wilhelm, R. (eds.) ESOP86: European Symposium on Programming. LNCS, vol. 213, Springer, Heidelberg (1986)
14. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
15. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie-Mellon University (November 1990)
16. Lee, K., Kang, K.: Feature dependency analysis for product line component design. In: ICSR, pp. 69–85 (2004)
17. Leuschel, M., Butler, M.: ProB: An automated analysis toolset for the B method. Technical report, Electronics and Computer Science, University of Southampton (2006)
18. Lutz, R., Gannod, G.: Analysis of a software product line architecture: an experience report. Journal of Systems and Software 66, 253–267 (2003)
19. Macala, R., Stuckey, Jr. L., Gross, D.: Managing domain-specific, product-line development. IEEE Software, pp. 57–67 (May 1996)
20. Métayer, C., Abrial, J.-R., Voisin, L.: Event-B Language. Technical Report Deliverable 3.2, EU Project IST-511599 - RODIN (May 2005) `http://rodin.cs.ncl.ac.uk`
21. Reiff-Marganiec, S., Ryan, M.D. (eds.): Proceedings ICFI 2005: Feature Interactions in Telecommunications and Software Systems VIII, Leicester. IOS Press, Amsterdam (2005)
22. Snook, C., Poppleton, M., Johnson, I.: The engineering of generic requirements for failure management. In: Kamsties, E., Gervasi, V., Sawyer, P.(eds.) Proc. Eleventh International Workshop on Requirements Engineering: Foundation for Software Quality, pp. 145–160, Oporto, March 2005, Essener Informatik Beitraege (2005)
23. van Lamsweerde, A.: Goal-oriented requirements engineering: A guided tour. In: Proc. RE'01 - International Joint Conference on Requirements Engineering, pp. 249–263, Toronto, August 2001, IEEE (2001)
24. Zhang, W., Zhao, H., Mei, H.: A propositional logic-based method for verification of feature models. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 115–130. Springer, Heidelberg (2004)