Towards Reuse with "Feature-Oriented Event-B"

Michael Poppleton, Bernd Fischer, Chris Franklin, Ali Gondal, Colin Snook, Jennifer Sorge
Dependable Systems and Software Engineering
University of Southampton
Southampton, SO17 1BJ, UK
{mrp, b.fischer, aag07r, cf105, cfs, jhs06r}@ecs.soton.ac.uk

Abstract

Event-B [19] is a language for the formal specification and verification of reactive systems. The language and its RODIN toolkit represent a leading model-based technology for formal software construction. However, scalability is a major current concern, especially the dimension of reusability. We outline a proposed infrastructure for scalable development with reuse for Event-B. We focus specifically on our agenda for reuse in Software Product Lines, and explain how a form of feature modelling will be central to this programme.

1 Introduction

Event-B [19] is a formal modelling language that evolved naturally from the classical B language [1] of J.-R. Abrial. The recent project RODIN[1] saw the definition of Event-B and the creation of the rich Eclipse-based [14] RODIN toolkit [2] for formal modelling, animation, verification, and proof with Event-B. This includes a project repository, syntax- and type-checkers, proof obligation generator, animators, theorem provers, and various front-end plug-ins.

In software development with Event-B, *refinement* is the central method by which initially small, abstract models of requirements are elaborated through architectural and detailed design to code. Refinement[2] M1 of a model M0 will usually remove some nondeterminism (implementation-freedom) and add data and algorithmic structure. M1 is mathematically *proved* to be a "black-box" simulation of M0, i.e. to offer only behaviour specified by M0. Event-B allows us to formally state and prove both consistency properties for models and refinement properties between them; we call these properties *proof obligations* (POs) in Event-B. These capabilities are part of the extra "bang for the buck" that Formal Methods offer to critical systems developers.

While there is now growing evidence of successful industrial critical systems development using B technology, e.g. [13, 18], only limited (and commercially protected) tool support exists to scale up to large applications. Project DEPLOY[3] aims to address this by scaling methodology in requirements validation, requirements evolution, reuse, and resilience, and scaling tooling in simulation, analysis and verification of formal models. This paper adds "feature-oriented Event-B" to that agenda.

Modularization and structuring are key issues in scaling Event-B models: a number of model decomposition mechanisms [3, 16, 21] have been proposed, and tool support for them is under development. The *event fusion* of [21] is designed specifically for feature-oriented structuring with Event-B.

The authors are working towards defining a method for feature-based modelling with Event-B, specifically aimed at reuse in software product lines (SPLs). Feature modelling [12] is a well understood approach for variability modelling for SPLs. To date it has mostly been applied to code or detailed design documents; we apply it to an abstract, nondeterministic language with formal semantics and verification conditions (POs). This paper outlines definitions of features as generic Event-B model elements, and defines feature composition and specialization. Using a simple example we present a scheme for precise definition of product line instances as particular feature compositions. This suggests a graphical feature modelling notation in the usual style, but with rigorous semantic foundations.

We present an agenda for methodological and tool development to support feature-oriented software development with Event-B. While the detailed feature structuring ideas

[1]RODIN - Rigorous Open Development Environment for Open Systems: EU IST Project IST-511599

[2]The term *refinement* is overloaded, meaning (i) the process of transforming one model into another, and (ii) the concrete model which refines the abstract one.

[3]DEPLOY - Industrial deployment of system engineering methods providing high dependability and productivity (2008 - 2011): FP VII Project 214158 under Strategic Objective IST-2007.1.2. Further information and downloadable tools are available at http://www.deploy-project.eu/

of this paper are syntactic, we emphasise that this is scaffolding for the real, semantic value that we anticipate from this work: the scaling of verification through a product line. The starting point is designing an identified feature and its chain of refinements, and then proving each of these refinements consistent, and a correct refinement transformation of its predecessor. This is part of the legwork of feature construction for the application domain.

The theoretical job is then as far as possible to prove compositionality, i.e. that consistency and refinement are preserved when we compose features. For any consistent features f and g we must prove $f \oplus g$ consistent (for a defined composition operator "$\oplus$"). Given their refinements f_1 and g_1, we must further prove $f_1 \oplus g_1$ both consistent and a correct refinement of $f \oplus g$. While such compositionality has been proved for the operators of [3, 16, 21], much work remains for the intricate needs of feature composition.

2 The Event-B language

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 its 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 reachable 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).

In Event-B the top level unit of modularization is the *model* consisting of a *machine* and zero or more *contexts*. The dynamic machine contains state variables, the state invariant, and the events that update the state. The static *context* contains sets, constants and their axioms.

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

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

That is, the action defined by $R(v)$ may occur only when the state enables the guard. An event E works in a model 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[4] $\hat{R}$ take s,c as parameters. Two of the consistency proof obligations (POs) for event E defined as (1) are FIS (feasibility) and INV (invariant preservation):

$$P(s,c) \wedge I(s,c,v) \wedge G(s,c,v) \Rightarrow \exists\, v' \bullet \hat{R}(s,c,v,v') \tag{2}$$

$$P \wedge I \wedge G(s,c,v) \wedge \hat{R}(s,c,v,v') \Rightarrow I(s,c,v') \tag{3}$$

Intuitively speaking, the static typing and axioms P and state consistency property I give the known properties of the system at any time. For event E (1), FIS states that when its guard G is true (enabled) at state v, then E - via its before-after predicate $\hat{R}$ - is able to make the state transition from v to v'. INV states that E will maintain the invariant, i.e. consistency: when G is enabled, any after-state v' reachable by E will satisfy invariant I.

In order to progress towards implementation, the process of *refinement* is used. A refinement is a (usually) more elaborate model than its predecessor, in an eventual *chain* of refinements to code.

The refinement of a context is simply its elaboration, by the addition of new sets, constants and axioms. When refining a machine, new variables may be added, and some or all abstract (refined) variables v may be replaced by new concrete (refining) ones w. New invariant clauses and events will usually be added, elaborating data and algorithmic structure. There are proof obligations for refinement, both for correctness of the simulation of an abstract model by its more concrete refinement, and for preservation of certain liveness properties. We do not discuss these further.

3 Feature-oriented Event-B ?

A small case study from project DEPLOY will be used to demonstrate the prototype scheme for product-line development with Event-B, based on formal feature modelling. The example consists of specifications and Event-B developments for two simple, related products: a switch and a pushbutton.[5] Switch and pushbutton each have a single two-valued output, off (false) or on (true). Each has one continuous input in the interval $[0,1]$. Rising and falling thresholds are used on the sampled input to determine switching conditions.

Neither specifications nor models have been developed through any product-line process: commonalities in the Event-B models were cut-and-pasted, and variabilities were modelled in situ for each model instance. Again, requirements features have simply been identified by intuition, rather than by any defined process.

A precise syntactic definition of an Event-B feature remains to be established after case study experience; for the present we regard the feature as a well-formed machine, context or model, and a subfeature as a well-formed element of such, e.g. a variable + typing invariant, a constant + typing axiom, an event. Expressiveness is required in the

[4] Here $R(v)$ is a syntax of actions, corresponding to a before-after predicate $\hat{R}(v,v')$.

[5] A 3-way and an n-way switch are also part of the product line, but have not been included for the sake of brevity.

Event-B feature definition to allow easy correspondence between requirements and model features.

The switch is specified as follows, paraphrasing [22], and adding named features. Note that all features here map to Event-B machine (i.e. behavioural) features, except for *bounce, threshr, threshf*, which are context features. The switch has four parameters (Debounce time BT, Rising threshold RTH, Falling threshold FTH, Cycle time CT). The input will be read cyclically with Cycle time CT.

1-1 Initially, the output is "off" (*initop*).

1-2i If the output is "off" and the switch on condition is true, the output is set to "on" (*switchopi*).

1-2ii If the output is "on" and the switch off condition is true, the output is set to "off" (*switchopii*).

1-3 A rising edge is detected if at time t the input is higher than RTH and at time t-CT it was lower than RTH (*edge*).

1-4 A falling edge is detected if at time t the input is lower than FTH and at time t-CT it was higher than FTH (*edge*).

1-5i The switch on condition is true if a rising edge was detected and the input exceeds RTH for BT after the rising edge (*swcond*).

1-5ii The switch off condition is true if a falling edge was detected and the input is lower than FTH for BT after the falling edge (*swcond*).

1-6 BT > CT (*bounce*)

1-7i 0 < RTH <= 1 (*threshr*)

1-7ii 0 <= FTH < 1 (*threshf*)

The requirements are now expressed in terms of the composition of features, e.g. including two variants of *swcond*. Such variation is achieved by making features specializable by parameter. The pushbutton differs from the switch in that it uses a single switch condition based on a rising threshold.

The specification suggests a graphical notation for the Event-B feature model - shown in Fig. 1 - comprising a machine graph and a context graph. This notation will build on some version of standard feature modelling notation [12]. We add two kinds of edge: "c" for "consists of", as per standard notation, and "r" for "refines" (we make the distinction because of the verification POs denoted by a "refines" edge). As usual, black or white dots denote mandatory or optional features.

Thus a switch device comprises machine features *initop, switchopi, switchopii* and context features *bounce, threshr, threshf*, in an abstract (level 0) model. The symbol $\oplus$ denotes various feature compositions outlined below. *switchopi*, the switch-on feature, is enabled when the switch is off. It is a machine feature comprising variable *output*, its typing invariant, initialisation, and event

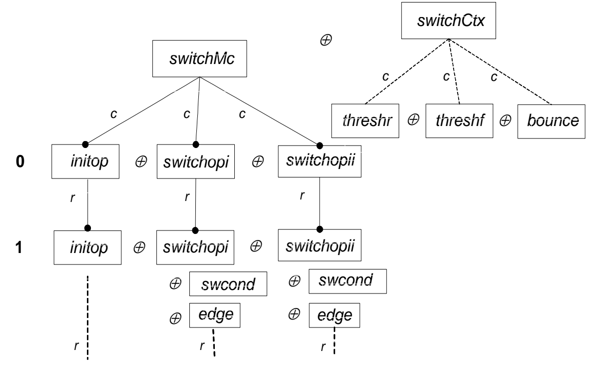

Figure 1. Switch feature diagram

```
out_F_T =
    WHEN grd1: output = false
    THEN act1: output := true
    END
```

switchopii is the reverse, switch off feature. Since at level 0 the switch condition is abstracted away, the switch and pushbutton models are identical at level 0.

At refinement level 1 abstractions of the switch condition *swcond* and the rising/falling edge test *edge* are introduced. A counter variable c is introduced and initialised to zero. When a rising edge is detected on the input by *edge*, c is set to $n = BT/CT$. c is decremented (by *swcond*) in each cycle that the input remains high, and the switch condition is satisfied when n reaches 1.

For each product line instance, its composition from constituent features, and these features' specialization (parameterization) must be explicitly defined at each refinement level. We show refinement level 1 for the switch:

$$Switch \mathrel{\widehat{=}} SwitchCtx \oplus_{scmc} SwitchMc$$

$$SwitchCtx \mathrel{\widehat{=}} (threshr \oplus_{sccc} threshf) \oplus_{scac} Axiom(\text{“}fth<rth\text{”})$$

$$\begin{aligned} SwitchMc \mathrel{\widehat{=}}\ & initop \oplus_{smmc} \\ & (switchopi \oplus_{smmc} \\ & \quad swcond(lbl = \text{“}re\text{”}, grd = \text{“}output = \text{false”}) \oplus_{smmc} \\ & \quad edge(lbl = \text{“}re\text{”}, grd = \text{“}output = \text{false”}) \\ & \oplus_{smmc} \\ & (switchopii \oplus_{smmc} \\ & \quad swcond(lbl = \text{“}fe\text{”}, grd = \text{“}output = \text{true”}) \oplus_{smmc} \\ & \quad edge(lbl = \text{“}fe\text{”}, grd = \text{“}output = \text{true”})) \end{aligned}$$

Next we elaborate the various composition operators $\oplus_{slrp}$ by describing these four modifiers:

- *s*: Strength of composition: *s*(default) denotes "strong", i.e. the two composing elements must be syntax- and type-consistent and must not require any user specialization. "w" denotes "weak", i.e. allowing

user specialization and resolution of inconsistencies at composition/instantiation time.

- *lr*: Syntactic kind of left l and right r elements being composed: these may be feature elements, i.e. m(default) for machine, c for context, and b for model, in any combination. Further, a feature may be composed with a subfeature of appropriate kind, i.e. machine m with variable(s) v, invariant i, event(s) e, or context c with constant(s) o, carrier set(s) r, axiom(s) a. A model b may compose with a consistent subfeature of any kind.
- *p*: Composition of predicates: whether to conjoin c(default) or disjoin d predicates, i.e. when combining invariant clauses, or adding guard clauses to an event, or fusing events.

For the switch instance *Switch*, context feature *SwitchCtx* is composed from context features for each of the rising (*threshr*) and falling (*threshf*) thresholds, as well as an extra axiom relating the two. All machine compositions are simply $\oplus_{smmc}$. For an example of specialization consider the single event in *edge* denoting threshold detection:

%lbl%_F_T =
WHEN %grd%:
grd1: $c = 0$
THEN act1: $c := n$
END

This skeleton feature requires a label and a guard to be completed. In the above definition of *Switch*, *edge* is instantiated twice, once for the rising edge (lbl="re", grd="$output$ = false"), resulting in event re_F_T and once for the falling edge, resulting in event fe_F_T. Thus, a rising edge can only be detected when output is off, and conversely for the falling edge instantiation.

Considering the second instance in our little SPL, the pushbutton differs from the switch precisely in that it uses a single switch condition *swcond* based on a rising threshold. Thus refinement 1 for *Pushbutton* differs from *Switch* in that (i) in *swcond, edge* for *switchopii*, the label parameters become "re", (ii) the outer composition between the *switchopi* and *switchopii* is $\oplus_{smmd}$, and (iii) *PushbuttonCtx* is simply *threshr*. In this case the two instantiations of *edge* in the same instance produce two versions of the same event re_F_T. Hence the two versions of event re_F_T must be *fused* [21]. That is, duplicated guards and actions are ignored, and the d-modifier on $\oplus_{smmd}$ specifies that the extra guard clause be disjoined. This gives a guard of $output$ = false $\vee$ $output$ = true, which should be preprocessed to true during instantiation, giving an always-enabled switch condition.

4 Tooling for feature modelling

Our tool development takes place in support of some future feature modelling process for Event-B. A feature modelling phase, during domain analysis, will develop a feature model based on any existing feature database, at the same time developing new features. This will include feature consistency proof, refinement and verification, as far as possible: the question of exactly how much verification can be done on an unspecialized feature remains open. Athough an event like %lbl%_F_T can be interpreted as well-formed Event-B, and be consistency-verified, in general this will not be true. Further, [20] described how in general a feature, not containing all behaviour affecting its variables, will fail to verify liveness POs.

An instance modelling phase will follow where system instance specifications will be developed in the style of the *Switch*. Most probably there will be iterative feedback to the feature modelling phase. Finally, an instance production phase will follow.

The starting point in tooling was the construction of an Eclipse Modeling Framework [11] (EMF) editor for Event-B, based on a language metamodel produced in DEPLOY. EMF, based on metamodelling in the UML sense, enables quick construction of a simple editor with a tree-structured user interface reflecting the metamodel structure. A composition metamodel was developed by inheritance from the Event-B metamodel, to define a small number of prototype feature compositions. A prototype EMF feature composition editor (comp-editor) was then produced based on the composition metamodel. This enables recording of the composition and specialization parameters in a particular composition instance.

The comp-editor shown in Fig. 2 allows the user to specify all composition and specialization parameters interactively. Its interactive style will be useful during the early feature and instance modelling activities. In the figure, on

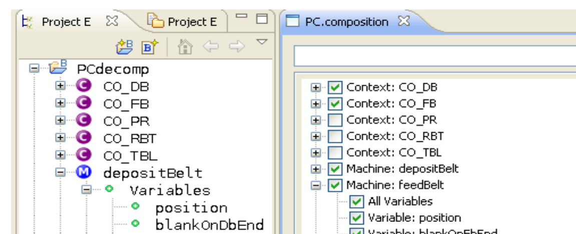

Figure 2. Composition tool

the left is a RODIN project explorer panel showing various project elements. When a project is clicked on, the comp-editor panel on the right is opened and blank. A drop-down menu allows selection of elements from the selected project, which produce corresponding tickboxes in the the comp-editor panel; in the figure we see identified context, machine and variable features. The user then instantiates the composition by ticking required features. On clicking "Compose", a third panel opens, allowing user specialization of selected

features and resolution of any conflicts. Dependency analysis is provided; e.g. given a variable the tool will identify all (sub-)features requiring that variable.

The next step for comp-editor is support for the automated composition variants "$\oplus_{slrp}$", which is a matter of suitably packaging existing functionality. Such automated compositions will be required for recording, managing, and generating predefined instance models. Next, an EMF feature metamodel must be constructed, against which the comp-editor should easily be adaptable for feature instance modelling. This will require full definition of the feature modelling language indicated by Fig. 1 and elaborated by the *Switch* definition. A further, more costly development, would be a graphical version of the EMF feature instance modeller.

Methodological work - beyond that in this paper - has started with an approach based on the "refinement by restriction" of [27]. A "maximal" Event-B model is constructed, containing all features. The feature model is annotated with mapping information to the Event-B model, so that instance modelling is done by slicing required features into the output instance model according to these mappings.

5 Related work

Recent proposals [8, 6] identifying generic algebraic models for feature-oriented software construction schemes are relevant to our work. These models can support instance construction; e.g. (i) associative composition operators give freedom in how they can be ordered, (ii) the occurrence of non-commuting compositions can indicate feature interactions. These ideas will inform the development of an algebra of Event-B features.

Turning to verification, another recent development [7] presents a product-line development where verification - theorem statements and their proofs - is modularized and assembled by features. For certain Event-B composition operators, certain properties (POs) are guaranteed by construction, as indicated in section 1. For most operators this will not be true, and patterns of construction will be sought that propagate POs, either partially or completely. [7] is encouraging, but we note that its case study exploits the fact that the feature increments are logical *conservative extensions*, i.e. each increment to the feature model does not interfere with prior features in the construction order. While Event-B *superposition* refinements, which simply add structure, should work similarly, in general refinements will not be modularizable in this way.

The notion of a feature as a reusable requirement [12] or an increment in functionality [10] emerged in the context of domain modeling and software product line engineering. However, features are often considered as concepts only, i.e., as names without any predefined semantics [12]. Feature diagrams can be given an (internal) semantics by translating them into propositional logic [10, 26], which can be used for checking the consistency of entire diagrams as well as individual configurations. Feature diagrams can also be "lifted" from a pure domain modeling method to a programming method by defining mappings into class diagrams [12], or by defining features as programming language constructs, e.g., in the language FeatureC++ [5]. Such feature-oriented programming languages [9] are usually implemented using generative techniques, e.g., mixins [23]. We anticipate that our approach will lift in the same way to UML-B [25], a grahical UML-like front-end for Event-B.

In formal methods, a variety of formally well-founded structuring methods have been developed, such as the ladder construction [24]. However, these typically focus on module composition and parameterization [15] and do not allow the combination of incomplete specification elements that could represent features.

6 Conclusion and future work

We have outlined a usable (if intricate) syntactic scheme and graphical notation for the automatable composition of each product line instance from a set of specializable features. The fact that this could be done based on a set of simple models with no prior generic structuring through some domain analysis process, gives us some confidence in this enterprise. It is of course a very modest start which must now be built on.

The future work required is extensive but clearly contributes to an existing agenda in both Generative Programming and Formal Methods communities, as identified at GPCE'06 [17]. This work consists of methodological (see section 4), theoretical, and tooling strands.

Theory:

1. From case study work, full definition of the feature composition operators outlined in section 4.
2. Establish the extent to which unspecialized features can be proved consistent, and can be proved refinements. Can this extend to the liveness POs ?
3. For all possible feature composition operators of section 4, proof of compositionality. For noncompositional operators, an investigation of what properties can be established.

Technology:

1. From case study work, definition of a feature modelling language for Event-B. To include graphical as well as composition/instantiation syntax as per section 3.
2. Implementation of a prototype working subset of such operators in RODIN, based on our current interactive composition editor prototype.

3. Development of a full feature and instance modelling toolset inspired by e.g. FeaturePlugin [4]. GUI design will be appropriate both to RODIN and to the visualisation demands of the user building or instantiating feature models.
4. Validation by case study application.
5. The RODIN provers build and manage proof trees for every proved PO, and take a reuse-oriented approach to the management of these trees, when models, then POs, then finally proof trees change. We need to investigate, for the many cases where full compositionality does not apply, whether unspecialized feature proof trees can be transformed for reuse in proving POs about their compositions. For example, if we have proof trees for features f, g and their refinements $\{f_i\}, \{g_j\}$, to what extent can we transform any of these proof trees to be be applicable for reuse in proof about $f \oplus_{slrp} g$ and its refinements ?

References

[1] J.-R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[2] J. R. Abrial, M. Butler, S. Hallerstede, and L. Voisin. An open extensible tool environment for Event-B. In Z. Liu and J. He, editors, *Proc. ICFEM 2006*, volume 4260 of *LNCS*, Macau, 2006.

[3] J.-R. Abrial and S. Hallerstede. Refinement, decomposition and instantiation of discrete models: Application to Event-B. *Fundamenta Informaticae*, 77(1-2):1–28, 2007.

[4] M. Antkiewicz and K. Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *Eclipse '04: Proceedings of the 2004 OOPSLA workshop on Eclipse technology eXchange*, pages 67–72, New York, NY, USA, 2004. ACM Press.

[5] S. Apel, T. Leich, M. Rosenmüller, and G. Saake. FeatureC++: On the symbiosis of feature-oriented and aspect-oriented programming. In R. Glück and M. R. Lowry, editors, *GPCE*, volume 3676 of *Lecture Notes in Computer Science*, pages 125–140. Springer, 2005.

[6] S. Apel, C. Lengauer, B. Möller, and C. Kästner. An algebra for features and feature composition. In J. Meseguer and G. Rosu, editors, *AMAST*, volume 5140 of *Lecture Notes in Computer Science*, pages 36–50. Springer, 2008.

[7] D. Batory and E. Börger. Modularizing theorems for software product lines: The jbook case study. *JUCS*, to appear.

[8] D. Batory and D. Smith. Finite map spaces and quarks: Algebras of program structure. Technical Report TR-07-66, Department of Computer Sciences, University of Texas at Austin, December 2007.

[9] D. S. Batory. Feature-oriented programming and the ahead tool suite. In *ICSE*, pages 702–703. IEEE Computer Society, 2004.

[10] D. S. Batory. Feature models, grammars, and propositional formulas. In J. H. Obbink and K. Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005.

[11] F. Budinsky, D. Steinberg, E. Merks, R. Ellersick, and T. J. Grose. *Eclipse Modeling Framework*. Addison-Wesley, 2003.

[12] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[13] B. Dehbonei and F. Mejia. Formal development of safety-critical software systems in railway signalling. In M. Hinchey and J. Bowen, editors, *Applications of Formal Methods*, chapter 10, pages 227–252. Prentice-Hall, 1995.

[14] E. Gamma and K. Beck. *Contributing to Eclipse*. Addison-Wesley, 2003.

[15] J. A. Goguen. Parameterized programming. *IEEE Trans. Software Eng.*, 10(5):528–544, 1984.

[16] C. Jones. Intermediate report on methodology. Technical Report Deliverable 19, EU Project IST-511599 - RODIN, August 2006. http://rodin.cs.ncl.ac.uk.

[17] G. T. Leavens, J. R. Abrial, D. Batory, M. Butler, A. Coglio, K. Fisler, E. Hehner, C. B. Jones, D. Miller, S. Peyton-Jones, M. Sitaraman, D. R. Smith, and A. Stump. Roadmap for enhanced languages and methods to aid verification. In *Proc. 5th Int. Conf. Generative Programming and Component Engineering*, Portland, Oregon, 2006.

[18] T. Lecomte, T. Servat, and G. Pouzance. Formal methods in safety-critical railway systems. In *Proc. 10th Brasilian Symposium on Formal Methods*, 2007.

[19] C. Métayer, J.-R. Abrial, and L. Voisin. Event-B Language. Technical Report Deliverable 3.2, EU Project IST-511599 - RODIN, May 2005. http://rodin.cs.ncl.ac.uk.

[20] M. Poppleton. Towards feature-oriented specification and development with Event-B. In P. Sawyer, B. Paech, and P. Heymans, editors, *Proc. REFSQ 2007: Requirements Engineering: Foundation for Software Quality*, volume 4542 of *LNCS*, pages 367–381, Trondheim, Norway, June 2007. Springer.

[21] M. Poppleton. The composition of Event-B models. In E. Boerger, editor, *Proc. ABZ 2008*, volume 5238 of *LNCS*, page 209222, London, September 2008. Springer.

[22] Robert Bosch GMBH. Specification on/off switch. 2008.

[23] Y. Smaragdakis and D. S. Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[24] D. R. Smith. Toward a classification approach to design. In M. Wirsing and M. Nivat, editors, *AMAST*, volume 1101 of *Lecture Notes in Computer Science*, pages 62–84. Springer, 1996.

[25] C. Snook and M. Butler. U2B - a tool for translating UML-B models into B. In J. Mermet, editor, *UML-B Specification for Proven Embedded Systems Design*, chapter 5. Springer, 2004.

[26] J. Sun, H. Zhang, Y.-F. Li, and H. H. Wang. Formal semantics and verification for feature modeling. In *ICECCS*, pages 303–312. IEEE Computer Society, 2005.

[27] A. Wasowski. Automatic generation of program families by model restrictions. In *SPLC 2004*, volume 3154 of *LNCS*, pages 73–89. Springer, 2004.