The original publication is available at
http://doi.org/10.1007/978-3-319-99933-3_9
In SETTA 2018: The 4th Internation Symposium on Dependable Software Engineering. Theories, Tools, and Applications © Springer 2018

Developing A New Language to Construct Algebraic Hierarchies for Event-B

James Snook, Michael Butler, and Thai Son Hoang

ECS, University of Southampton, Southampton, U.K.
{jhs1m15,mjb,t.s.hoang}@ecs.soton.ac.uk

Abstract. This paper proposes a new extension to the Event-B modelling method to facilitate the building of hierarchical mathematical libraries to ease the formal modelling of many systems. The challenges are to facilitate building mathematical theories, be compatible with the current method and tools, and to be extensible by users within the Rodin Platform supporting Event-B.
Our contribution is a new language, called $B^\sharp$, which includes the additional features of type classes and sub-typing. The $B^\sharp$ language compiles to the current language used by the Rodin's Theory Plug-in, which ensures consistency, and also gives compatibility with the current Rodin tools. We demonstrate the advantages of the new language by comparative examples with the existing Theory Plug-in language.

Keywords: Formal methods; Event-B; Theorem Prover; Mathematical Extensions

1 Introduction

The Event-B method [1] and its supporting Rodin Platform (Rodin) [2] are designed specifically for system modelling. Event-B incorporates mechanisms such as refinement and decomposition to cope with the system complexity. Rodin includes facilities such as animation, model checking, and theorem proving for validating and verifying the Event-B models. Often, during system development, in order to ensure system dependability, developers need to model the system's operating environment. Having extensive mathematical libraries makes this modelling task faster and easier. Building these libraries of mathematical definitions and theorems is made considerably easier with the right tools and language features.

The challenges addressed in this paper are designing a language with the required features to enable the development of consistent mathematical theories, in such a way that minimises repeated proofs. Moreover, the mathematical theories can be used within the Event-B models.

Our contribution therefore is the design of a new language, called $B^\sharp$, with features to aid the construction of mathematical libraries. The new features are designed to reason about abstract and concrete mathematical types. The $B^\sharp$

language is mapped to the current Event-B syntax as supported by Rodin and its Theory Plug-in. The mapping phase will generate necessary theorems that required to be discharged by the developers. Other proof rules will be generated which would (normally) have required a manual proof in Event-B, however, the new language will generate the proof itself.

Structure. The rest of this paper is structured as follows. Section 2 examines the mathematics that we want to model, and the relationships between mathematical types. We also summarise the modelling task in Event-B and its Theory extension. Section 3 shows the problems with the current tools by describing a case study, and presents elements of the $B^\sharp$ language to facilitate the construction of mathematical definitions and theorems. Section 4 discusses the related work and gives some conclusion of our work.

2 Background

2.1 Mathematical Data Structures

Within mathematics, the study of abstract algebra deals with abstract structures. Rather than dealing with specific functions and sets, they generalise to deal with all sets and functions that have given properties. For instance, the abstract structure of a monoid is a set S and a binary function f and an identity element e, such that:

$$\forall x, y, z \in S \cdot f(x, f(y, z)) = f(f(x, y), z) \text{ , and} \tag{1}$$

$$\forall x \in S \cdot f(x, e) = x \land f(e, x) = x \ . \tag{2}$$

Property (1) declares that the function f is associative. Property (2) defines e as an identity element.

Many concrete structures are examples of this abstract type such as, addition and zero on the real numbers, or matrix multiplication with the identity matrix. Theorems and functions on the abstract type apply to all of the concrete examples, so the proof only has to be done on the abstract type.

Axioms can be added to abstract types to form new types e.g., a group is a monoid where all elements have an inverse. A group can utilise all the results from a monoid (as the group has all of the monoid's axioms), and can have new theorems provable with the new axioms.

Reasoning like this reduces the proof burden as proofs done on the abstract type do not need to be repeated by either concrete instances, or abstract types that extend the current type. The proofs are inherited by the new types.

2.2 Event-B and the Theory Plug-in

The Event-B modelling method [1] allows the modelling of discrete event systems. The Event-B modelling language is supported by the Rodin Platform [2], an open and extensible toolset for constructing formal models. To increase the ability of Event-B to model systems the Theory Plug-in [5] was added to Rodin allowing the extension of the Event-B mathematical language with user-defined operators and proof rules. The Theory Plug-in can be used as a theorem prover to create domain-specific mathematical theories. In [4] a 3-D Euclidean space was modelled, and used to formally verify the safety of a set of paths of Unmanned Aerial Vehicles (UAVs). The Euclidean space model would be useful to any other system requiring a safe distance is maintained. Other mathematical structures that model environment are also widely reusable e.g., two's complement arithmetic would be useful to many software system models. The aim of creating the new language B$^\sharp$ is to improve the tools for building these mathematical models for Event-B.

3 A Language for Mathematical Libraries in Event-B

3.1 A Case Study Using Theory Plug-in

This section summarises the result of a case study evaluating the ability of the current Theory Plug-in to build and use algebraic hierarchies. The case study used the Theory Plug-in to construct abstract and concrete mathematical classes, and then see how they can be related.

On the one hand, our case study shows that abstract mathematical types were representable within in the Event-B syntax. It also found that abstract types could be extended to make new abstract types, and that concrete types could be related to the abstract types. On the other hand, the following issues with the representation were identified:

1. Event-B operators are not first class members of the language, resulting in the need to encapsulate them within total functions to relate concrete types to abstract types, e.g., showing addition and zero form a monoid required a theorem such as: $zero \mapsto (\lambda x, y | x\ add\ y) \in Monoid(pNat)$ (the operator is encapsulated within a lambda construct).
2. Demonstrating that a concrete object forms an algebraic type does not make it inherit the theorems/proof rules of the algebraic type. These have to be re-written and proved (although the proof can be constructed by instantiating the theorems/proof rules of the algebraic type).
3. When making theorems about an abstract type the type required construction from its constituent parts, resulting in verbose theorem definitions. Alternatively the abstract types can be passed in and deconstructed with the Event-B projection operators making the theorem difficult to read/understand (this can be helped by the user making operators to deconstruct the abstract types).
4. The Event-B language is not able to reason about subsets as types, this resulted in the user having to manually do many well-definedness proofs.
5. Predicates in Event-B are not expressions. This makes it difficult to reason about relations (instead the $BOOL$ type was used and turned back into a predicate where necessary using equality).

6. Abstract types definitions and declarations rapidly increased in complexity.

3.2 The B$^\sharp$ Language

This section gives a brief introduction to the B$^\sharp$ language, in particular focusing on the **Class** declaration, allowing the user to create new type classes, and new subtypes, fully supported by the language. A type class allows the definition of a subtype of some existing type structure such that the subtype has additional properties and operators. A type class also allows us to constrain polymorphism. For example, the following declaration defines the *ReflexRel* class:

$$\begin{aligned}&\textbf{Class}\ \mathit{ReflexRel}\langle T\rangle : T \times T \rightarrow \mathit{Pred}\\ &\qquad \textbf{where}\ \forall x : T, \mathit{rel} : \mathit{ReflexRel}\langle T\rangle \cdot \mathit{rel}(x,x)\{\}\end{aligned} \tag{3}$$

This class declaration creates a type class $\mathit{ReflexRel}$ a subtype of $T\times T \rightarrow \mathit{Pred}$, any relation in $\mathit{ReflexRel}$ must have the additional property that all elements are related to themselves.

Some differences to the Event-B syntax can be seen immediately. The polymorphic type T can be a subtype i.e., a type created using the subtyping mechanism above. In Event-B this is the equivalent of allowing entities created with the subset syntax to be treated as types. The B$^\sharp$ language does extra work to reason about subtypes and reduce well-definedness proofs. In Event-B predicates are a different syntactic category to boolean expressions and are not first class. In B$^\sharp$, predicates are first class of type Pred.. It allows the language to create functions that return predicates without having to use the BOOL type as an intermediate.

This class declaration maps to the following underlying Event-B statement:

$$\begin{aligned}\mathit{ReflexRel}(t : \mathbb{P}(T)) \hat{=} \{&\mathit{rel} | \mathit{rel} \in \mathbb{P}(t \times t)\\ &\wedge \forall x \cdot x \in t \Rightarrow x \mapsto x \in \mathit{rel}\}\end{aligned} \tag{4}$$

To allow the $\mathit{ReflexRel}$ operator to work on subtypes the Event-B power set operator $\mathbb{P}$ is used to give the type of t. Pred is replaced by contricting the set of rel. When using rel within an expression the mapping to Event-B will become $x \mapsto x \in \mathit{rel}$ when the Event-B language requires a predicate value (e.g., as in the quantifier in (4)).

The class declaration can also be used to create type classes where the type class is required to have certain elements, e.g., a monoid:

$$\begin{aligned}&\textbf{Class}\ \mathit{Monoid} : \mathit{Setoid}(\mathit{ident} : \mathit{Monoid}, \mathit{op} : \mathit{AssocOp}\langle \mathit{Monoid}\rangle)\\ &\qquad \textbf{where}\ \forall x : \mathit{Monoid}\ \cdot\ \mathit{op}(x, \mathit{ident})\ \mathit{Monoid.equ}\ x\\ &\qquad\quad \wedge\ \ \mathit{op}(\mathit{ident}, x)\ \mathit{Monoid.equ}\ x\{\}\end{aligned} \tag{5}$$

Type classes create templates for new classes. For a class to become part of the monoid type class it needs to have an identity and an associative operator that follows the rules in the **where** clauses. $Monoid : Setoid$ means that $Monoid$ is a subtype of the $Setoid$ type class, which is a type which has an equivalence relation (this is created using a class declaration similar to the one above). Definition (5) maps to the following Event-B:

$$\begin{aligned} &Monoid(t : \mathbb{P}(T)) \mathrel{\hat=} \{setoid \mapsto ident \mapsto op \mid \\ &\quad setoid \in Setoid(t) \land ident \in t \land \\ &\quad op \in AssocOp(t, setoid) \land \\ &\quad \forall x \cdot x \in t \Rightarrow \\ &\qquad op(x \mapsto ident) \mapsto x \in Setoid\_equ(setoid) \land \\ &\qquad op(ident \mapsto x) \mapsto x \in Setoid\_equ(setoid)\} \end{aligned} \tag{6}$$

$$Monoid\_Setoid(m : Monoid(\mathbb{P}(T))) \mathrel{\hat=} prj1(m) \tag{7}$$

$$Monoid\_ident(m : Monoid(\mathbb{P}(T))) \mathrel{\hat=} prj1(prj2(m)) \tag{8}$$

$$Monoid\_op(m : Monoid(\mathbb{P}(T))) \mathrel{\hat=} prj2(prj2(m)) \tag{9}$$

Given a instance $a = b_1 \mapsto b_2 \cdots \mapsto b_n$ $prj1(a)$ will give b_1 and $prj2(a)$ will give $b_2 \mapsto \ldots b_n$. From (6) it can be seen that the B$^\sharp$ syntax is much more concise. It is useful to see how a type becomes a member of the $Monoid$ type class:

$$\textbf{Instance } Monoid(zero, add); \tag{10}$$

This will make the $pNat$ type (inferred from the $zero$ and add arguments) an instance of a $Monoid$. A proof obligation to demonstrate that addition and zero form a monoid will be generated. Proof rules, theorems and functions from the $Monoid$ are re-written to rules about zero and addition and added to the $pNat$ type. As these have been proved in the $Monoid$ type class they do not need to be reproved.

Polymorphic types can be restricted to a given type class, e.g.:

$$\begin{aligned} &\textbf{Class } AssocOp < T : Setoid > T \times T \to T \\ &\textbf{where } \forall x, y, z : T \cdot AssocOp(AssocOp(x, y), z)\ T.eq\ AssocOp(x, AssocOp(y, z)) \end{aligned} \tag{11}$$

The polymorphic type T has to be a member of the $Setoid$ type class (it has to have an equivalence relation). This will map to the following Event-B:

$$\begin{aligned} &AssocOp(t : \mathbb{P}(\mathbb{T}), setoid : Setoid(t)) \mathrel{\hat=} \{op \mid op \in t \times t \to t \\ &\qquad \land \forall x, y, z \cdot x \in t \land y \in t \land z \in t \\ &\qquad \Rightarrow op(op(x \mapsto y) \mapsto z) \mapsto op(x \mapsto op(y \mapsto z)) \in Setoid\_Eq(setoid) \end{aligned} \tag{12}$$

The polymorphic context in (11) ($< T : Setoid >$) becomes the the arguments to the Event-B operator in (12). $Setoid\_Eq$ is an Event-B deconstructor created from the *Setoid* type class mapping, in the Event-B mapping this is mapped from the B$^\sharp$ $T.eq$ statement.

The syntax for a **Class** statement is:

$$\textbf{Class}\ \gamma\langle\tau_1 : \gamma_1, \ldots, \tau_n : \gamma_n\rangle : S_1 \ldots S_m\ (s_1 : T_1, \ldots, s_p : T_p)\ \textbf{where}\ e_1; \ldots; e_l; \{\} \tag{13}$$

This declaration has the following meanings:

1. γ is the name chosen for the new class, e.g., $ReflexRel$ in Example (3), this maps to the Event-B operator name.
2. $\langle\tau_1 : \gamma_1, \ldots, \tau_n : \gamma_n\rangle$ is the polymorphic context. γ_i are optional, and restrict τ_i to a given type class. These map to the arguments of the Event-B operator.
3. $S_1 \ldots S_m$ are super types and $(s_1 : T_1, \ldots, s_p : T_p)$ define addition structure. In the Event-B syntax they are mapped to a statement such as $\{\gamma \mapsto s_1 \mapsto \cdots \mapsto s_p | \gamma \in S_1 \wedge dots \wedge \gamma \in S_m \wedge s_1 \in T_1' \cdots \wedge s_p \in T_p' | \ldots\}$. With multiple inheritance any shared supertypes remain shared, the mapping for these is more complex than shown above (it requires the supertypes to be deconstructed to their last shared ancestor). This is omitted for brevity.
4. Properties $e_1; \ldots; e_l$ are predicate expressions. These create the subtype from the supertypes. Each e_i is translated to the Event-B syntax and they constrain the set returned.

Due to the space limit, we omit other features of B$^\sharp$, such as B$^\sharp$ functions, and their mapping to Event-B, or the generation of proof rules from the B$^\sharp$ class statements (to make proving easier).

4 Related Work and Conclusion

There are many examples of similar constructs in other languages. Of particular interest is Coq [3] for which there has been an extensive library of abstract algebraic structures developed [6]. The language feature which makes building this library possible is called type classes [11] originally created for Haskell. Type classes set out a structure, which types can adopt, and inherit functions from the type class. Isabelle [10] also has a similar feature allowing abstract specifications called locales [9]. Algebraic Specification [8] languages give a formal specification to datatypes rather than describing the structure of the datatype. This abstraction means that many concrete datatypes could comply with the specification, giving a similar concept to the ones described above. For example, OBJ3 [7] has a similar concepts with parameterised modules and theories. Theories define a structure for an module, parameterised types can then be restricted to models with this structure.

The novelty in the B$^\sharp$ language is not the invention of new language features but of tailoring and applying these to the Rodin toolset, mapping the extended B$^\sharp$ features to the Event-B syntax for consistency and proof purposes. The work above demonstrates a method by which this can be achieved. These new features will allow for the development of hierarchies of generic theories and the ability to develop domain-specific specialisations of these, while avoiding the need to redo proofs over similar structures for each specialisation.

References

1. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: An open toolset for modelling and reasoning in event-b. *STTT*, 12(6):447–466, 2010.
3. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.
4. Chris Bogdiukiewicz, Michael J. Butler, Thai Son Hoang, Martin Paxton, James Snook, Xanthippe Waldron, and Toby Wilkinson. Formal development of policing functions for intelligent systems. In *28th IEEE International Symposium on Software Reliability Engineering, ISSRE 2017, Toulouse, France, October 23-26, 2017*, pages 194–204. IEEE Computer Society, 2017.
5. Michael J. Butler and Issam Maamria. Practical theory extension in Event-B. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 67–81. Springer, 2013.
6. Luís Cruz-Filipe, Herman Geuvers, and Freek Wiedijk. C-corn, the constructive Coq repository at nijmegen. In *MKM*, volume 3119 of *Lecture Notes in Computer Science*, pages 88–103. Springer, 2004.
7. Joseph A Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In *Software Engineering with OBJ*, pages 3–167. Springer, 2000.
8. J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10(1):27–52, Mar 1978.
9. Florian Kammüller, Markus Wenzel, and Lawrence C. Paulson. Locales a sectioning concept for Isabelle. In Yves Bertot, Gilles Dowek, Laurent Théry, André Hirschowitz, and Christine Paulin, editors, *Theorem Proving in Higher Order Logics*, pages 149–165, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.
10. Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science.* Springer, 2002.
11. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.