B♯: Enabling Reusable Theories in Event-B

James Snook, Thai Son Hoang, and Michael Butler

Electronics and Computer Science, University of Southampton, United Kingdom
{jhs1m15,t.s.hoang,mjb}@ecs.soton.ac.uk

1 Background

The Event-B formal method [1] is used for system modelling and verification, it is supported by the Rodin IDE (Integrated Development Environment) [2]. Event-B was designed for modelling discrete systems using a rich set theoretic modelling language; it was not designed as a general theorem prover. This design focus resulted in some useful structures being difficult to model in a reusable way. This was partially addressed by the introduction of the Theory Plug-in [5] which extended the Event-B language with polymorphic recursive datatypes (allowing types such as lists and the naturals to be easily created) and user-defined operators. User defined operators came in two flavours, axiomatically defined operators and constructively defined operators (working more like functions in other languages, substituting arguments into expressions).

The extended Event-B syntax has been used to construct axiomatic definitions of types, e.g., the real numbers have been modelled as a commutative field, which has been used within Event-B system modelling [4]. This approach results in a lot of repetition in definition and theorem proving. For example, when constructing a field it is not possible to reuse a group construct, which results in it being necessary to repeat the group axioms for addition and multiplication, and repeat any group theorems and proofs. This was noted in [9], which proceeded to show that abstract mathematical structures such as monoids could be created using the Event-B set syntax in such a way that they could be related to concrete types (such as the naturals). Further, the abstract types could be used to construct other abstract types (e.g., using the group definition to facilitate the construction of a field). There were several problems noted with this approach, such as proving a concrete type was an instance of an abstract type (e.g., that zero and addition form a monoid) did not automatically allow theorems to be used on the concrete type (the user needed to manually move the theorems). It was noted that the work-arounds to these problems were repetative enough to be done automatically, and could be applied during a translation from another language. The B♯ language was proposed [10] for this purpose, with syntactic elements designed for the construction of mathematical types in such a way that they could be translated to the Event-B syntax, allowing them to be used by an Event-B modeller.

2 The B♯ Tools

An initial implementation of the B♯ language has been made. This is constructed in a way to make it compatible with the Rodin tool set. Several mathematical structures have been implemented using the B♯ tool to test its effectiveness e.g.,

$$\begin{aligned}&\textbf{Class } Monoid[M] : SemiGroup\ (ident : M)\\ &\quad \textbf{where } \forall x : M \cdot equ(op(x, ident), x) \wedge equ(op(ident, x), x)\{\dots\}\end{aligned} \tag{1}$$

This declares that a *Monoid* is a *SemiGroup* (*SemiGroup* is a previously defined type with an equivalence relation *equ* and a binary operator *op*, and is referred to as the supertype of

Monoid) with an identity (*ident*). The **where** statement introduces a series of predicates to constrain the class, in this case they define the identity to work in the expected way. Within the type body ($\{\ldots\}$) theorems and functions are declared. For example the following theorem about the identity:

$$\forall x : M \cdot equ(op(x, ident), ident) \Leftrightarrow equ(x, ident) \tag{2}$$

M was declared above (1), and can be used as an instance of the Monoid class anywhere within the type body. As can the *op*, *equ* and *ident* members of the *Monoid*, they do not need to be explicitly quantified over.

Concrete types are also constructed such as the natural numbers (*Nat*). The following statement is used to show that zero and addition form a commutative monoid:

$$\textbf{Instance}\ CommMonoid\langle pNat\rangle\ (add, zero)\ addMon \tag{3}$$

addMon is a name allowing the *CommMonoid* to be explicitly referenced in other B♯ statements.

The B♯ tool uses an instance statement like (3) to produce an Event-B theorem stating that *add* and *zero* form a *CommMonoid*. It also instantiates all of the theorems and functions declared in the class bodies of the abstract types (*CommMonoid* and its supertypes such as *Monoid*). For example (2) instantiates to:

$$\forall x : Nat \cdot add(x, zero) = zero \Leftrightarrow x = zero \tag{4}$$

This in principal needs no proof, instead the proof is done on (1) so it is true for all *Monoid*s and the **Instance** statement requires it to be proved that *add* and *zero* form a *CommMonoid* (which requires them to be a *Monoid*).

Instantiating the theorems in this manner makes future proofs considerably easier as the person doing the proofs does not have to manually instantiate the theorems whilst proving. When developing theorems in B♯ it was found that the theorems were considerably more concise than the equivalent theorems developed in the Event-B case study [9]. A large part of this was due to the instantiation mechanism, however, other features of the B♯ language helped, including the B♯ type system, and predicates not being a separate syntactic category in B♯ (unlike Event-B). It was also notable that due to the IDE features built into the B♯ tool such as syntax aware autocompletion and on the fly error highlighting, it was easier and faster in our opinion to develop theories than when using the Theory Plug-in tools.

3 Discussion and Future Work

B♯ bears similarities to HOL [7] style languages, with a type class style feature [12]. Languages such as Coq [3] and Isabelle/HOL [8] with similar features have been used to construct large libraries of mathematical types. These features also have a lot in common with Algebraic specification language [6], where parameterised programming [11] allows generic types to be constricted to type specifications. The unique feature of this work is the translation to the set theoretic syntax used by Event-B, and the interaction with its tools.

Developing mathematical theorems in B♯ highlighted problems and improvements that could be made to the B♯ language. For example **Instance** statements infer their supertype from their parametric context, which is the wrong way round; supertypes, if required, should be explicit. In **Class** statements what constitutes a supertype needs to be extended. An attempt was made

to create a class of all commutative monoids on the naturals with equality as the equivalence operator, whilst this was possible it would have been easier if **Instance**s could be supertypes. These issues will be addressed with updates to the B♯ syntax.

The B♯ tool does not interact directly with the Rodin interactive prover (this is done through the translation to existing Event-B and the current tools). The B♯ type system is not the same as the Event-B type system, as it allows subtyping (and requires functions to explicitly state their return types). Prover rules and tactics could be generated for the interactive prover to allow this additional information to be used, in some cases this would allow proofs to be discharged automatically, further reducing the proof burden on the user.

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. *Int. J. Softw. Tools Technol. Transfer*, 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 *Proc. of 28th Int. Symp. on Software Reliability Engineering, ISSRE 2017*, pages 194–204. IEEE, 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] J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Inf.*, 10(1):27–52, 1978.

[7] M. J. C. Gordon and T. F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[8] 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.

[9] James Snook. Technical note: building abstract mathematical types in Event-B. Working paper, University of Southampton, April 2019.

[10] James Snook, Michael J. Butler, and Thai Son Hoang. Developing a new language to construct algebraic hierarchies for Event-B. In Xinyu Feng, Markus Müller-Olm, and Zijiang Yang, editors, *Proc. of 4th Int. Symp. on Dependable Software Engineering: Theories, Tools, and Applications, SETTA 2018*, volume 10998 of *Lecture Notes in Computer Science*, pages 135–141. Springer, 2018.

[11] James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Data type specification: Parameterization and the power of specification techniques. In *Proc. of 10th Ann. ACM Symp. on Theory of Computing, STOC '78*, pages 119–132. ACM, 1978.

[12] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proc. of 16th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, POPL '89*, pages 60–76. ACM, 1989.