Event-B as DSL in Isabelle and HOL

Experiences from a Prototype

Benoît Ballenghien[1] $^{[0009-0000-4941-187X]}$ and Burkhart Wolff[2] $^{[0000-0002-9648-7663]}$

[1] Université Paris-Saclay, LMF, France
`benoit.ballenghien@universite-paris-saclay.fr`
[2] Université Paris-Saclay, LMF, France
`burkhart.wolff@universite-paris-saclay.fr`

Abstract. The proof assistant Isabelle/HOL is made available inside a flexible system framework allowing for logically safe extensions, which comprise both theories as well as implementations for code-generation, documentation, and specific support for a variety of formal methods.
Following the techniques in [9] and the theoretical groundwork in [4], we show the major milestones for the implementation of a B-Tool and the resulting refinement method inside the Isabelle/HOL platform. The prototype HOL-B provides IDE support, documentation support, a theory for the *Z-Mathematical Toolkit* underlying the B-Method, and a generated denotational semantics for a B MACHINE specification implemented as a specification construct in Isabelle/HOL.
Extended by more automated proof machinery geared to refinements, HOL-B can serve as a more portable, flexible and extensible tool for Event-B that may profit from the large Isabelle/HOL libraries providing Algebra and Analysis theories.

Keywords: Event-B, DSLs, Formal Methods, Isabelle/HOL, Refinement

1 Introduction

A recurring question in formal methods research groups is :

What is actually a domain specific language (DSL) ?

The second author of this paper proposed a particular answer to this question, that led to a number of infrastructure developments in the Isabelle/HOL [7] platform. The infrastructure exploits the fact that the Isabelle/HOL proof assistant is made available inside a flexible system framework allowing logically safe extensions, which comprise both theories as well as implementations for an IDE, documentation, specific support for a variety of formal methods and code-generation.

Boiling down [9] and its more theoretical groundwork in [4] to just one sentence, this can be read like this:

A DSL is a function from a DSL syntax to a conservative theory transformation,

so, a bit more formally, a function of type $DSL_{syntax} \Rightarrow theory \Rightarrow theory$, where we still have to clarify what *conservative* means.

The objective of this paper is to make this idea still more precise and to construct along this line a prototype for Event-B [8][1][5][2][3] inside the Isabelle/HOL platform. The result, Isabelle/HOL-B, offers a library for basic Event-B concepts (the *Z-Mathematical Toolkit* underlying the B Method), syntax and IDE support for editing, and the generation of a denotational semantics for Event-B [9] machines, which can be used for refinement proofs. However, sophisticated support for the latter, i. e. Event-B specific automated proof support, is out of the scope of this paper. (The latter is, in our view, the true research question.)

We will proceed as follows: after the introduction of HOL and a strict minimum of the Isabelle platform API in SML, we will describe the *Z-Mathematical Toolkit*, and finally the parser, type-checker, and encoder parts of the DSL function. The resulting Isabelle/HOL-B [3] will be demonstrated with an example.

2 Background

2.1 Isabelle, HOL and the Conservative Method

Isabelle is a platform for proof assistants in various logics, where Higher-Order Logic (HOL) is the one which is most commonly used. It has a fairly small kernel in the tradition of LCF provers, which certifies all derivations on formulas. Formulas and logical rules are represented in typed terms of the polymorphic λ-calculus. Induction and closure rules can be derived within the system. Modeling in HOL has strong similarities with programming in a functional programming language.

The core of the logic HOL can be based on a very small set of seven axioms introducing the equality, the type *bool*, the logical connectives, and a type *ind* which must have an infinite carrier set. HOL comes by construction with a typed set theory; α *set*'s are characteristic functions of type $\alpha \Rightarrow bool$ with $\{\}$ as notation for $\lambda x.\ False$ and $UNIV$ as $\lambda x.\ True$.

Extensions of this core are built exclusively by two syntactically restricted axiom schemes:

– the *constant definitions* which state the equality of a fresh constant symbol to a closed λ-term (not containing this symbol)
– the *type definitions* which state an isomorphism between a fresh type constructor symbol to a closed λ-term denotating a nonempty set.

The first can be seen as the introduction of an abbreviation for a known concept; the latter as an introduction as constraint of a subset of some type

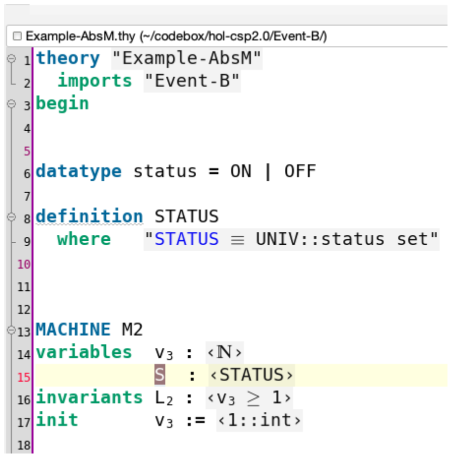
[3] ... available at `https://gitlab.lisn.upsaclay.fr/burkhart.wolff/hol-csp2.0/-/tree/master/Event-B`

constructed over *bool* and *ind*. It is not difficult to see that these definitional axioms will preserve logical consistency of the HOL core.

The HOL library provides a rich number of mathematical concepts like cartesian products, number theories, inductive sets and datatypes. For the latter constructions, Isabelle/HOL provides specific support in form of DSL's for these *specification constructs*, that compile them into definitional axioms and derive the proof rules from them usually automatically. This includes recursive function definitions as well as record notations built over cartesian products.

2.2 A Gentle Introduction into Isabelle Programming

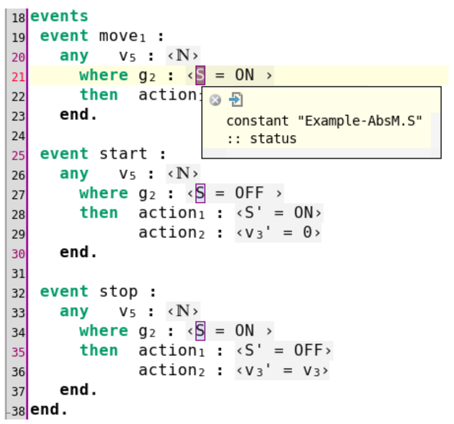
The final result of our programming exercise is shown in Fig. 1: A theory command deeply integrated in the Isabelle IDE allowing continuous parsing and type-checking, navigation into the resulting implicit definitions as well as the underlying background theory adding the set theory operators (that B actually inherited from the "Z-Mathematical Toolkit"). Here, *navigation* means that the IDE treats the MACHINE as a hypertext allowing to link to a corresponding (implicit) declaration by mouse click. In the following, we will describe the techniques to construct the DSL function and its integration into Isabelle which results in this behaviour.



Example-AbsM.thy (~/codebox/hol-csp2.0/Event-B/)

```
theory "Example-AbsM"
  imports "Event-B"
begin


datatype status = ON | OFF

definition STATUS
  where    "STATUS ≡ UNIV::status set"



MACHINE M2
variables  v3 : ‹ℕ›
           S  : ‹STATUS›
invariants L2 : ‹v3 ≥ 1›
init       v3 := ‹1::int›
```

(a) A Machine Spec ...



```
events
 event move1 :
   any   v5 : ‹ℕ›
     where g2 : ‹S = ON ›
     then  action1
   end.

 event start :
   any   v5 : ‹ℕ›
     where g2 : ‹S = OFF ›
     then  action1 : ‹S' = ON›
           action2 : ‹v3' = 0›
   end.

 event stop :
   any   v5 : ‹ℕ›
     where g2 : ‹S = ON ›
     then  action1 : ‹S' = OFF›
           action2 : ‹v3' = v3›
   end.
end.
```

```
constant "Example-AbsM.S"
:: status
```

(b) ... and its Navigation.

Fig. 1: A B-Machine Specification in Isabelle/HOL-B

First, there are commands giving access to the underlying SML toplevel; by :

ML ‹... *some ML code* ... ›

we get a programming IDE that gives access to the interfaces of the Isabelle system. In particular, the concrete datatype `term` provides a syntactic model of λ-terms (consisting of free/bound variables, abstraction, application), with the usual operations (substitution, reduction etc.). Terms are annotated by explicit type information, which we have to compute before terms can be certified and used by more complex operations like "generate a definition" or "generate a record to represent the MACHINE state". The concrete datatype `theory` captures the signature of a theory (with type constructor declarations, constant symbol declaration, and syntax configurations) as well as a set of axioms and derived theorems. Both were captured in the concrete datatype `thm` which captures the traditional triple $\Gamma \vdash_{Th} \varphi$ (from local assumptions Γ within the theory *Th* the formula φ has been derived).

3 The Z Mathematical Toolkit

In this section, we will discuss the background theory of Z, B and Event-B which was developed in the 80s at the University of Oxford and which led, at least in the Z case, to a standardized *Mathematical Toolkit*. HOL and B have closely related, but different foundations: HOL is based on functions, while Z/B are based on sets. The former has the advantage that typed-lambda calculi have higher-order term normal-forms which are decidable, while the latter permits a more traditional mathematical presentation. These foundational choices have consequences in the modeling style: HOL libraries tend to totalize all functions (1 / 0 is usually defined by 0 in many HOL systems) while Z/B emphasize to model partiality explicitly, even at the cost of additional complexity in substitution and deduction. Note that we are not saying here that the HOL approach is unsound - we recall the long mathematical tradition to totalize functions which lies, for example, at the heart of calculus.

Given that most Z/B users develop their models in a typed way anyway, we suggest to rebuild the Mathematical Toolkit inside the typed set theory of HOL rather than axiomatic set theory, which is viable, but hampers the access to HOL libraries.

As a start, we have to redefine the set operators (i. e. *constants* in HOL) $\emptyset$, $A \times B$, and $\mathbb{P}\ A$ of type $'\alpha\ set$, $'\alpha\ set \Rightarrow\ '\beta\ set \Rightarrow ('\alpha \times\ '\beta)\ set$ and $'\alpha\ set \Rightarrow\ '\alpha\ set\ set$, respectively. On top we define the set operators from the book:

```
definition rel:: ‹['a set, 'b set] ⇒ ('a ⇔ 'b) set› (infix ‹↔› 100)
  where ‹A ↔ B ≡ ℙ (A × B)›
definition pfun :: ‹['a set, 'b set] ⇒ ('a ⇔ 'b) set›     (‹- ⇸ -›  [54,53] 53)
  where ‹S ⇸ R ≡ {f ∈ S ↔ R. ∀ x y1 y2. (x, y1)∈f ∧ (x, y2)∈f ⟶ y1 = y2}›
definition pinj::‹['a set,'b set] ⇒ ('a ⇔ 'b) set›       (‹- ⤔ -›  [54,53] 53)
  where ‹S ⤔ R ≡ {s ∈ S ⇸ R. ∀ x1 x2 y. (x1,y)∈s ∧ (x2,y)∈s ⟶ x1 = x2}›
definition dom-restr :: ‹['a set , 'a ⇔ 'b] ⇒ ('a ⇔ 'b)› (‹- ◁ -›  [71,70] 70)
  where ‹S ◁ R ≡ {(x, y). (x, y) ∈ R ∧ x ∈ S}›
definition dom-substr ::‹['a set , 'a ⇔ 'b] ⇒ 'a ⇔ 'b›   (‹- ⩤ -› [71,70] 70)
  where ‹S ⩤ R ≡ {(x, y). (x, y) ∈ R ∧ x ∉ S}› ... etc. etc.
```

From this definitional basis, we derive the laws from the mathematical toolkit, which is usually an easy exercise. Here are a few examples:

$$\begin{array}{ll}
dom\ (S \lhd R) = S \cap dom\ R & S \lhd R \rhd T = S \lhd (R \rhd T) \\
dom\ (Q \mathbin{⨾} R) = (Q^{-1})\llparenthesis dom\ R\rrparenthesis & p \in s \lhd r \Longrightarrow p \in r \\
dom\ (dom\ g \mathbin{⩤} f) \cap dom\ g = \emptyset & f \in A \leftrightarrow B \Longrightarrow s \in \mathbb{P}\ A \Longrightarrow s \lhd f \in A \leftrightarrow B
\end{array}$$

Some definitions are a bit more delicate: since functions in B are relations, the application needs to be distinguished from the built-in application in HOL. These two definitions describe the conversions between functional and function-as-relation applications:

$$\begin{array}{ll} \langle \mathit{Lambda}\ A\ f & \equiv \{(x,\ y).\ x \in A \wedge y = f\,x\}\rangle \\ \langle R \cdot x & \equiv \mathit{SOME}\ y.\ (x,\ y) \in R\rangle \end{array}$$

which results in the beta-reduction rule: $a \in A \Longrightarrow (\mathit{Lambda}\ A\ f) \cdot a = f\,a$.

While the derivation of most rules from the definitions is straightforward, there are some cases which actually required some more serious proof work: the definitions of the various closure operators (reflexive/transitive ...) in the Z Mathematical Toolkit take partiality into account; the resulting induction rules therefore need different justifications than their HOL counterparts.

4 The Event-B Encoder

4.1 Getting Started: Parsing and Toplevel-Integration

Isabelle's API provides a common infrastucture to construct parsers; a type synonym `'a parser` which are functions that map a stream of input tokens to a parsed value and rest-stream, i. e. a function of type $\mathit{token\ list} \to {'a} * \mathit{token\ list}$. Parsing combinators [6] allow this kind of functions to be combined; notably by the sequential composition $P \mathbin{{-}{-}} P'$ of parsers, the alternative $P \mathbin{||} P'$ and the mapping of a function f into the result of a parsing $P >> f$. The toplevel function of the MACHINE parser reads as follows:

```
val parse-machine-spec = (
        Parse.binding
        -- keyword‹variables›     -- (Scan.repeat1 parse-var-decl)
        -- keyword‹invariants›    -- (Scan.repeat1 parse-invariant)
        -- keyword‹init›          -- (Scan.repeat1 parse-init)
        -- keyword‹events›        -- (Scan.repeat1 parse-transition)
        -- keyword‹end.›
    )
```

which allows for the toplevel composition of the DSL function by:

```
command command-keyword‹MACHINE›
      Machine Specification
      (parse-machine-spec >> context-check >> (Toplevel.theory o semantics))
```

and its binding to the keyword *MACHINE* and thus its integration into the Isabelle document model.

4.2 Getting Started: Semantics

We have to clarify some leftovers from the previous section. First, the function *context-check* has to be constructed: it has type $\mathit{absy0} \to \mathit{theory} \to \mathit{absy}$, i. e. it converts a raw abstract syntax into a function that produces a richer syntax where all sub-expressions are type-checked in the given theory context of the MACHINE specification. Isabelle annotates the content in the IDE with coloring and navigation information during this process.

Second, the *semantics* which is a function that takes the $\mathit{theory} \to \mathit{absy}$ function as input, executes it, and produces a theory extension $\mathit{theory} \to \mathit{theory}$ that is lifted via the Isabelle combinator *Toplevel.theory* into a global transition of the Isabelle system state. This completes the construction of the aforementioned DSL function as well as its integration into the Isabelle system level. Note that the pure functional API enables parallel execution inside the Isabelle kernel.

The *semantics* function generates type and constant definitions, notably:

– a record definition modeling the entire machine *state*; this constructs also the definition of the "variables" of the spec as selectors in that state,
– a constant definition that comprises all variable constraints,
– a constant definition is generated for the initial state,
– a predicate of $\mathit{state} \Rightarrow \mathit{bool}$ for the invariant and a constant STATES comprising the set of states satisfying the invariant,
– for each event declaration L_i, there is a constant definition of the form $L_i\text{-}\mathit{trans} \equiv \lambda\sigma\ e\ \sigma'.\ \exists\ a_1...a_n.\ \mathit{guard}\ \sigma\ e\ a_1...a_n \wedge \mathit{action}\ \sigma\ e\ \sigma'\ a_1...a_n$
– there is a constant definition for TRANS that composes the global transition relation as union of the L_i-*trans*itions.

For example, the *start* event in Fig. 1 is converted into the definition: $\mathit{start\text{-}trans} \equiv \lambda\sigma\ \mathit{event}\ \sigma'.\ \exists v_5.\ v_5 \in \mathbb{N} \wedge S\ \sigma = \mathit{OFF} \wedge S\ \sigma' = \mathit{ON} \wedge v_3\ \sigma' = 0.$

5 Conclusion

In this paper, we demonstrated how a perhaps little known technique to construct DSL's can be used to build tool support for the B method inside the Isabelle platform. The resulting prototype offers a typed version of the mathematical toolkit underlying B and Event-B, based on definitional principles and derived rules, and continuous parsing and type-checking. Its integration into the Isabelle document model also permits seamless navigation into models and libraries as well as document generation. The specification constructs were translated into a family of definitions capturing the denotational semantics representing the states, events and transitions of a Machine specification. While quite powerful, our prototype is fairly small: about 450 lines for the encoder and about 3000 lines of definitions and proofs for the mathematical toolkit.

The potential benefit of our approach for the ABZ community is to profit from the developments in the generic Isabelle platform wrt. libraries, powerful prover technologies, and user interface technologies rather than investing effort into own tools.

Future work will have to address specialized proof automation for the refinement of B machines. Another line of extension is to adapt the powerful generic code-generators of Isabelle to the idiom of the mathematical toolkit and to generate code and animations of B specifications.

References

1. Abrial, J.: The B-Book - Assigning Programs to Meanings. Cambridge University Press (1996). https://doi.org/10.1017/CBO9780511624162, https://doi.org/10.1017/CBO9780511624162
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J., Butler, M.J., Hallerstede, S., Hoang, T.S., Mehta, F., Voisin, L.: Rodin: An Open Toolset for Modelling and Reasoning in Event-B. Int. J. Softw. Tools Technol. Transf. **12**(6), 447–466 (2010). https://doi.org/10.1007/S10009-010-0145-Y, https://doi.org/10.1007/s10009-010-0145-y
4. Brucker, A.D., Tuong, F., Wolff, B.: Model Transformation as Conservative Theory-Transformation. J. Object Technol. **19**(3), 3:1–16 (2020). https://doi.org/10.5381/JOT.2020.19.3.A3, https://doi.org/10.5381/jot.2020.19.3.a3
5. Cansell, D., Méry, D.: Foundations of the B Method. Comput. Artif. Intell. **22**(3-4), 221–256 (2003), http://www.cai.sk/ojs/index.php/cai/article/view/456
6. Hutton, G.: Higher-Order Functions for Parsing. Journal of Functional Programming **2**(3), 323–343 (1992). https://doi.org/10.1017/S0956796800000411
7. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002). https://doi.org/10.1007/3-540-45949-9
8. Robinson, K.A.: Introduction to the B Method, pp. 3–37. Springer London, London (1999). https://doi.org/10.1007/978-1-4471-0585-5_1, https://doi.org/10.1007/978-1-4471-0585-5_1
9. Wenzel, M., Wolff, B.: Building Formal Method Tools in the Isabelle/Isar Framework. In: Schneider, K., Brandt, J. (eds.) Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings. Lecture Notes in Computer Science, vol. 4732, pp. 352–367. Springer (2007). https://doi.org/10.1007/978-3-540-74591-4_26, https://doi.org/10.1007/978-3-540-74591-4_26