Meta-Programming Event-B — Advancing Tool Support and Language Extensions

Julius Armbrüster, Philipp Körner

Meta-Programming Event-B

Advancing Tool Support and Language Extensions

Julius Armbrüster and Philipp Körner

Heinrich Heine University Düsseldorf — Faculty of Mathematics and Natural Science — Department of Computer Science — 40225 Düsseldorf, Germany
{julius.armbruester,p.koerner}@hhu.de

Abstract. Transforming models based on their textual representation is a cumbersome task. This is particularly the case for Event-B, where the predominant representation is a set of XML files. As a consequence, tool support is lacking, even for minor refactoring operations.
The contribution of this paper extends the *lisb* library with a front and backend based on Event-B. The aim is to bring benefits, that have been demonstrated for classical B, such as an easily transformable data representation of formal specifications as well as creation of custom DSLs and tooling, to Event-B.
We see great benefits of such a meta-programming approach for formal specifications and advocate that similar mechanisms will be sensible extensions to the expressiveness of formal methods. Ultimately, our work facilitates language extensions (e.g., re-introducing if-then-else constructs to Event-B which generate multiple events or a proper macro system to avoid code duplication) and tool support (e.g., refactoring tools or automatic refinement).

Keywords: B-Method · Event-B · Meta-Programming

1 Introduction

The benefits of DSLs in formal methods is well-known and many tools have been brought forth in particular for B and Event-B [10,11,24]. The *lisb* library[1] [12] embeds the classical B language in a general-purpose programming language. In contrast to similar embeddings [14,17], this means that not only expressions and predicates can be solved; but instead an intermediate representation (IR) of the underlying B machine is generated that is still susceptible to programmatic transformations (e.g., to implement custom DSLs or data refinement tools [12]).

In this article, we present our work to bring this meta-programming approach to Event-B by extending *lisb* to support Event-B. This approach caters to programmers who will (i) prefer a data representation in a programming language to generated specifications from data; (ii) obtain easier way to modify their specification (e.g., DSLs, shared sub-expressions or refactoring); (iii) prefer

[1] https://www.github.com/pkoerner/lisb

a text-based format over the default structural editors Rodin offers; (iv) gain the ability to quickly prototype tools for Event-B.

The rest of the paper is structured as follows: We give an overview of B and Event-B as well as the *lisb* library in Section 2. Next, we describe extensions made for Event-B to *lisb* in Section 3. Related work will be presented in Section 4. We discuss our contribution in context of possible future developments in Section 5.

2 Background

In this section, we first give an overview of B and Event-B. Details can be found in a survey on their industrial usage [5] as well as a comparison of the formalisms [15]. We also outline the architecture of and our additions to *lisb*.

2.1 B and Event-B

Both B [1] (or "classical B") and Event-B [2] are state-based formal methods. They share a *correct-by-construction* workflow. Starting from an abstract, mathematical description based on set theory, the model is iteratively refined by adding implementation details. Each refinement step is linked with the previous via *proof obligations*. Assisted by automatic provers, one can prove that the overall behaviour of the system is preserved. Once a low-level subset of B is reached, one can apply code generators (e.g., [18]) to obtain executable code.

Main Differences Between B and Event-B The mathematical sub-language used for predicates and expressions is almost identical in both formalisms. However, there are some discrepancies:

- *Operators*: Some operators are present in only one formalism (e.g., `FIN` (the set of finite subsets) in classical B, or `partition(S, x, y, ...)` (S is a disjoint union of x, y, ...) and `finite(S)` (a set S is finite) in Event-B). Others differ slightly (e.g., minus `-` and multiplication `*` are polymorphic w.r.t. numbers and sets in B, while two operators each are used in Event-B).
- *Guarded Substitutions*: Classical B offers different substitutions (`SELECT`, `PRE`, `IF`, `CASE`, ...) that can be nested arbitrarily. Event-B only offers flat events that allow only guards and assignments.
- *Machine Structure*: B specifications can be structured as a set of machines with a variety of clauses for inclusion and reference (`USES`, `INCLUDES`, `SEES`, `EXTENDS`, ...). Event-B separates the static components (as *contexts*) from the dynamic parts (as *machines*). Then, machines may refine a single other machine at most, but can reference multiple contexts.

Rodin [3] is the de facto IDE for Event-B that we target with *lisb* (though AtelierB [8] supports Event-B as well). Its two built-in editors allow manipulation of the structure (e.g., adding nodes for guards, operations or invariants) due to the underlying file format (details will be discussed in Section 3.3). As the workflow of these structural editors can be inconvenient, proper text-based editor were developed as plug-ins [4,9].

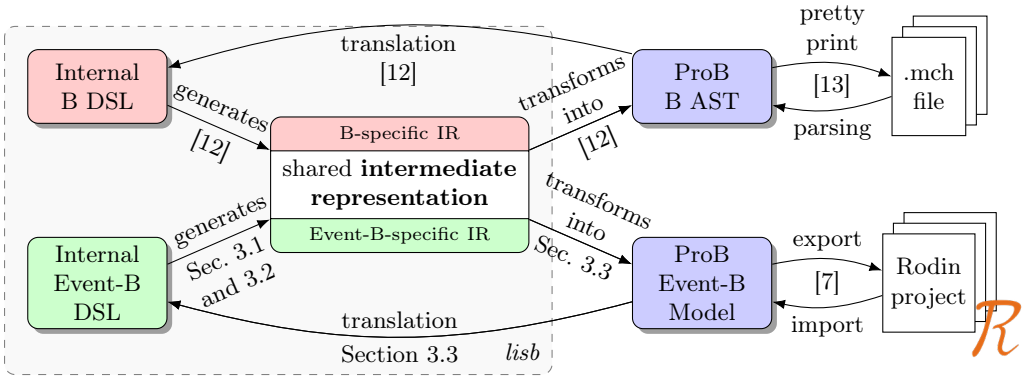

Fig. 1. Simplified Overview of the *lisb* Architecture

2.2 *lisb*

The *lisb* library [12] embeds the B language in Clojure, a functional Lisp running on the JVM. This allows to make use of (i) Lisp's powerful macro systems (for DSLs) and (ii) ProB's Java API [13] and its powerful toolchain (including ProB's B parser, pretty printer, type and model checker, etc.).

The *lisb* library itself consists of three components: (i) An internal DSL which is intended to be read and written by humans and which can be freely extended by user DSLs. (ii) An intermediate representation (IR) which is readable but appropriate for programmatic transformations (e.g., data refinement, refactoring or translation). (iii) A tool backend that is the translation target for the IR. For classical B, ProB [16] was used to output the corresponding `.mch` file(s).

Figure 1 depicts this architecture. We extend the upper half, i.e., an internal DSL for B, an intermediate representation and the integration of ProB's AST as a backend. The article's contributions can be found in the lower half, by adding an internal DSL for Event-B, extending the IR for Event-B and integrating a generation of Rodin projects. These are discussed in Section 3 below.

3 Embedding Event-B in *lisb*

In this section, we outline three categories of extensions made to the *lisb* library. First, language components that are part of both classical B and Event-B (with the same semantics); Second, additions made to the IR which are specific to Event-B; Third, how an Event-B (Rodin) project is obtained from the IR, and how an Event-B project is transformed back to *lisb*'s internal Event-B DSL.

3.1 Shared Sub-Language

Large parts of B and Event-B are very similar, e.g., logical predicates or number and set operators. In these instances, the exact same IR can be generated.

```
(defmacro eventb [body]
  `(b (let [... bindings of Event-B-specific operators ...]
        ~body)))
```

Listing 1.1. The `eventb`-Macro

Note that the IR is not concerned with syntactic differences: E.g., the Cartesian product of S and T can be written as `S * T` (classical B), as `S ** T` (Event-B ASCII) or $S \times T$ (Event-B Unicode). These expressions are all represented by the same data literal `{:tag :cartesian-product, :sets (:S :T)}` in the IR.

We show the concept of the internal Event-B DSL in Listing 1.1. Note that an `eventb` expression in *lisb* introduces Event-B-specific operators in a local binding. Everything else is evaluated in the *context* of classical B expressions. The internal DSL inherits almost the entire sub-language of predicates and expressions from B, generates exactly the same data (IR) for the shared language, and is also arbitrarily to extendable. This also allows shadowing B-specific operators, e.g., one could disallow B's `FIN` operator and throw an exception instead.

3.2 Event-B-Specific Extensions

Below, we give examples for language elements that are specific to Event-B.

Operators The `partition(S, T1, ... Tn)` predicate, which holds true iff the set `S` is equal to the disjoint union of an arbitrary number of other sets $T_1, \ldots, T_n$, is particular to Event-B. There are two possibilities how to add this to *lisb*:

A *DSL-based* approach maps an operator to a B expression, i.e., `partition` is re-written to $S = \bigcup_{i=1}^{n} T_i \wedge T_1 \cap T_2 = \varnothing \wedge \ldots \wedge T_{n-1} \cap T_n = \varnothing$. This means that an export to or import from Rodin will always generate this bloated formula.

Instead, we opted for an *IR-based* approach that adds a `partition` node to the IR. This node is handled by the Event-B backend (emitting the idiomatic predicate) but is rejected by the classical B backend. If needed in classical B, one can add the re-writing rule above as an IR-to-IR transformation.

Machine Structure Specifications in Event-B are split into contexts and machines. While we can re-use the IR of some existing structures (e.g., machines as sets of clauses), most structuring elements specific to Event-B require additions to the IR. Examples are contexts or the additional variant clause in machines. Even the structure of events differ too much, as they may include witnesses, may be marked as convergent, etc.

3.3 Interacting with Rodin Projects

A more accurate image than Fig. 1 for our Event-B extension is given in Fig. 2. In the following, we will discuss the details of how a Rodin project is obtained from the intermediate representation in *lisb* and vice versa.

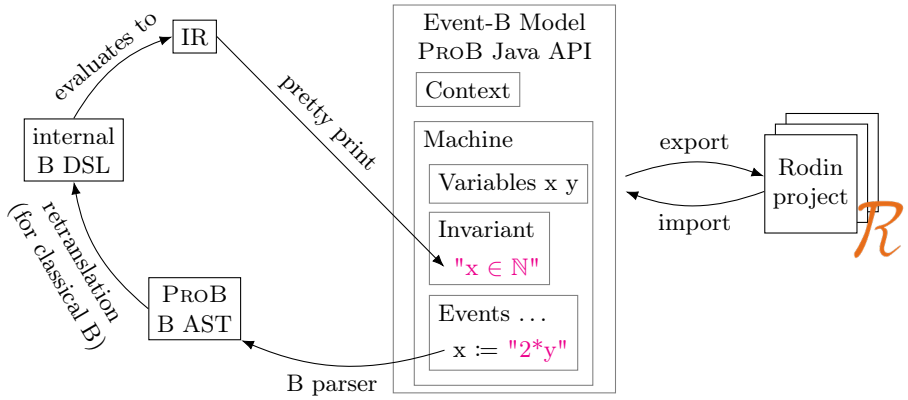

Fig. 2. Transformations Between IR and Rodin Projects

Generation of a Rodin Project In the Rodin XML files, predicates (for guards, invariants, theorems, etc.) and expressions (for substitutions) are stored as pretty-printed strings. Thus, one step in the translation from the IR to Rodin projects is a pretty printer for such predicates and expressions. For the rest of the structure of the file, we make use of the ProB Java API [13]:

As part of a DSL for Algorithm Descriptions [7], a Java representation of Event-B models was created. E.g., in this Java representation one can construct guards, actions or parameters from pretty-printed strings and re-combine them into an Event object. Step by step, the model is then made up of multiple machine and context objects. This representation can then be exported to `.bum` and `.buc` files as well as the `.project` file containing the Rodin project description.

Reading a Rodin Project To obtain the IR from a given project, we use the same Java representation for Event-B models. We generate the code for the internal Event-B DSL that evaluates to the IR. The structure of this code aligns with the structure of the model; e.g., machines consisting of invariants, variants, variables and events, and events in turns consist of guards, actions, etc.

ProB's parser suite offers an extension of its classical B parser that is also able to parse Event-B-specific operators. We make use of this parser to obtain an AST. The existing re-translation from B yields the internal DSL for classical B for all operators that are present in B and Event-B. As no B-specific terms can be read from an Event-B project, the obtained DSL must be valid Event-B. The re-translation is extended to also support all Event-B-specific operators.

4 Related Work

Embedding specification languages in general-purpose programming languages is not a novel approach. A first prototype of *lisb* only embedded the predicate

and expression sub-language of classical B with the aim of generating constraints from external data sources [21]. This is comparable with αRby, an embedding of Alloy in Ruby [17]. Yet, αRby focuses on mixed execution of both specification and programming language, allowing interaction with partial solutions. No data-oriented representation of the generated model seems to be accessible.

PlusCal [14] is an imperative language in which arbitrary TLA$^+$ expressions can be inserted, catering to programmers. Such programs can be translated to pure TLA$^+$ for verification with TLC. A similar approach is an algorithm description language [7] by Clark, which provides a similar language but with Event-B expressions. A version of such DSLs is also shipped by *lisb*.

Another approach to programmatic transformations is to make use of a meta-model, i.e., a formal model that represents another model in the hosted formalism as data. Most notable for Event-B is the EB4EB framework [19] which extends the core formalism via a meta-theory.

5 Discussion

In this paper, we showed how we integrated the Event-B notation in *lisb*, which allows easy construction of DSLs and tools for the formalism. Further, the library can be used to implement translations *between* formalisms: As a proof of concept, we implemented an IR-to-IR transformation that translates (only) the B-specific IR to constructs that are shared or specific to Event-B — resulting in a tool that translates B machines to Rodin projects[2]. While a large portion of B is covered, some kinds of substitutions and ways to include machines are still missing. If extended by further formalisms (with a similar mathematical core), one could obtain a universal translation tool.

We think our approach ultimately combines several ideas stemming from different formal methods sub-communities:

- Rozier et al. [20] presented their vision of a yet to be named model checking framework[3]. The goal is to translate all specification languages to a common IR so that a singular tool implementing advanced algorithms suffices.
- By adding frontends (and translating only specifics), it is easy to *integrate* multiple formalism. This is in line with combined formalisms such as Circus [23] or CSP||B [22].
- Meta-approaches for specification languages allow easier implementation of DSLs, generation of specifications and advanced transformations tools.
- Better and easier integration of formal methods in general-purpose programming languages will allow non-expert to interact with and integrate formal models into their work. It can also make it more accessible for students [6].

We do not argue that *lisb* will be the ultimate answer. Instead, we hope to see that in the long run, specification languages will evolve and adopt similar, powerful mechanisms for meta-development.

[2] https://github.com/jarmbrue/b2eventb
[3] https://www.aere.iastate.edu/modelchecker/

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Voisin, L.: An Open Extensible Tool Environment for Event-B. In: Proceedings ICFEM (International Conference on Formal Engineering Methods). Lecture Notes in Computer Science, vol. 4260, pp. 588–605. Springer (2006)
4. Bendisposto, J., Fritz, F., Jastram, M., Leuschel, M., Weigelt, I.: Developing Camille, a text editor for Rodin. Software: Practice and Experience **41**, 189–198 (2011)
5. Butler, M., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L.F., Voisin, L.: The First Twenty-Five Years of Industrial Use of the B-Method. In: Proceedings FMICS (International Conference on Formal Methods for Industrial Critical Systems). Lecture Notes in Computer Science, vol. 12327, pp. 189–209. Springer (2020)
6. Cerone, A., Roggenbach, M., Davenport, J., Denner, C., Farrell, M., Haveraaen, M., Moller, F., Körner, P., Krings, S., Olveczky, P., Schlingloff, B.H., Shilov, N., Zhumagambetov, R.: Rooting Formal Methods within Higher Education Curricula for Computer Science and Software Engineering – A White Paper. In: Proceedings FMFun (International Workshop on Formal Methods - Fun for Everybody) 2019. CCIS, vol. 1301. Springer (2021)
7. Clark, J., Bendisposto, J., Hallerstede, S., Hansen, D., Leuschel, M.: Generating Event-B Specifications from Algorithm Descriptions. In: Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM and Z). Lecture Notes in Computer Science, vol. 9675, pp. 183–197. Springer (2016)
8. ClearSy: Atelier B, User and Reference Manuals. Aix-en-Provence, France (2016), available at http://www.atelierb.eu/
9. Hoang, T.S., Snook, C., Dghaym, D., Salehi Fathabadi, A., Butler, M.: The CamilleX framework for the Rodin platform. In: Proceedings ABZ (International Conference on Rigorous State-Based Methods). Lecture Notes in Computer Science, vol. 12709, pp. 124–129. Springer (2021)
10. Idani, A.: Meeduse: A Tool to Build and Run Proved DSLs. In: Proceedings IFM (International Conference on Integrated Formal Methods). Lecture Notes in Computer Science, vol. 12546, pp. 349–367. Springer (2020)
11. Iliasov, A., Lopatkin, I., Romanovsky, A.: The SafeCap Platform for Modelling Railway Safety and Capacity. In: Proceedings SAFECOMP (International Conference on Computer Safety, Reliability, and Security). Lecture Notes in Computer Science, vol. 8153, pp. 130–137. Springer (2013)
12. Körner, P., Mager, F.: An Embedding of B in Clojure. In: Companion Proceedings MODELS (International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings). p. 598–606. ACM (2022)
13. Körner, P., Bendisposto, J., Dunkelau, J., Krings, S., Leuschel, M.: Integrating formal specifications into applications: the ProB Java API. Formal Methods in System Design **57**, 160–187 (2020)
14. Lamport, L.: The PlusCal Algorithm Language. In: Proceedings ICTAC (International Colloquium on Theoretical Aspects of Computing). Lecture Notes in Computer Science, vol. 5684, pp. 36–60. Springer (2009)

15. Leuschel, M.: Spot the Difference: A Detailed Comparison Between B and Event-B. In: Logic, Computation and Rigorous Methods, Lecture Notes in Computer Science, vol. 12750, pp. 147–172. Springer (2021)
16. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Software Tools for Technology Transfer **10**(2), 185–203 (2008)
17. Milicevic, Aleksandar Erfrati, I., Jackson, D.: aRby—An Embedding of Alloy in Ruby. In: Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM and Z). Lecture Notes in Computer Science, vol. 8477, pp. 56–71. Springer (2014)
18. Rivera, V., Cataño, N., Wahls, T., Rueda, C.: Code generation for Event-B. Software Tools for Technology Transfer **19**(1), 31–52 (2017)
19. Rivière, P., Singh, N.K., Aït-Ameur, Y., Dupont, G.: Standalone Event-B Models Analysis Relying on the EB4EB Meta-theory. In: Glässer, U., Creissac Campos, J., Méry, D., Palanque, P. (eds.) Proceedings ABZ (Rigorous State-Based Methods). Lecture Notes in Computer Science, vol. 14010, pp. 193–211. Springer
20. Rozier, K.Y., Shankar, N., Tinelli, C., Vardi, M.: Developing an Open-Source, State-of-the-Art Symbolic Model-Checking Framework for the Model-Checking Research Community. In: Proceedings FMCAD (Tutorial at the Conference on Formal Methods in Computer-Aided Design). pp. 1–1 (2023)
21. Schneider, D.: Constraint Modelling and Data Validation Using Formal Specification Languages. Ph.D. thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf (2017)
22. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Aspects of Computing **17**(4), 390–422 (2005)
23. Woodcock, J., Cavalcanti, A.: The semantics of Circus. In: Proceedings ZB (International Conference of B and Z Users). Lecture Notes in Computer Science, vol. 2272, pp. 184–203. Springer (2002)
24. Yar, A., Idani, A., Collart-Dutilleul, S.: Merging Railway Standard Notations in a Formal DSL-Based Framework. In: Proceedings ECSA (European Conference on Software Architecture). CCIS, vol. 1269, pp. 411–419. Springer (2020)