A case for data-oriented specifications: simpler implementation of B tools and DSLs

Philipp Körner, Florian Mager & Jan Roßbach

Suggested Citation:
Körner, P., Mager, F., & Roßbach, J. (2025). A case for data-oriented specifications: simpler implementation of B tools and DSLs. Innovations in Systems and Software Engineering, 21(3), 939–959. https://doi.org/10.1007/s11334-025-00596-3

Innovations in Systems and Software Engineering (2025) 21:939–959
https://doi.org/10.1007/s11334-025-00596-3

S.I.: MODELS/MODEVVA'22 & SAM'22

A case for data-oriented specifications: simpler implementation of B tools and DSLs

Philipp Körner[1] · **Florian Mager**[1] · **Jan Roßbach**[1]

Received: 20 February 2023 / Accepted: 14 January 2025 / Published online: 15 March 2025

Abstract
Considering programs as data enables powerful meta-programming. One example is Lisp's macro system, which gives rise to powerful transformations of programs and allows easy implementation of domain-specific languages. Formal specifications, however, usually do not rely on such mechanisms and are mostly written by hand in a textual format (or using specialised domain-specific language (DSL) tools). In this paper, we investigate the opportunities that stem from considering *specifications* as data. For this, we embedded the B specification language in Clojure, a modern Lisp. We use Clojure as a functional meta-programming language and the ProB Java API to capture the semantics of B, i.e., to find solutions for constraints or animate machines. From our experience, it is especially useful for tool development and generation of constraints and machines from external data sources. It can also be used to implement language extensions and to design DSLs.

Keywords B · ProB · Clojure · Language embedding · Meta-programming · Tool development · Domain-specific languages

1 Introduction

Formal specification languages are usually employed to gain a mathematical description of problems, algorithms and state machines. The syntax and features of many formalisms typically are set in stone in order to capture a precise semantics of a small core language. As a consequence, infrastructure for domain-specific languages (DSLs) and model transformation tools is usually not available. These points—domain-specific modelling environments and transforming models—have been identified as two of the core challenges in model-driven engineering (MDE) [18] and still remain so today [43].

On the other hand, DSLs and code transformation have a long history and strong support in other programming languages. One example is the family of Lisp programming languages, where meta-programming is easily accessible through a rich macro system. Because Lisp code is written in the data structures of the language, i.e., lists, it can be easily programmatically transformed into other code. Thus, even complex transformations of code snippets or DSL elements can be implemented with relatively low effort.

As our main contribution of this paper, we explore the combination of these two paradigms: We embed the language of an MDE methodology, the B method [2], as a DSL in Clojure [26], a modern Lisp that runs on the JVM. Further, we employ the ProB Java API [33] to capture the semantics of B, to construct B machines under the hood, to find solutions for constraints and for all verification & validation activities (V&V).

We argue that, regardless of the embedded formalism or the hosting programming language, strong support for language extensions via DSLs as well as model transformations is worth exploring. In this paper, we demonstrate the concept by borrowing the infrastructure that Clojure provides—an advanced macro system, a rich standard library and access to the Java eco-system of libraries. Ultimately, we hope that these features will be supported by specification languages themselves, especially so that such transformations will be captured more formally than we do in our approach. The focus of this article for now is to facilitate the work of tool developers (rather than modelling experts); but we hope to convince the reader that the modelling expert *can and should be* the tool developer.

✉ Philipp Körner
p.koerner@hhu.de

Florian Mager
florian.mager@hhu.de

Jan Roßbach
jan.rossbach@hhu.de

[1] Institut für Informatik, Heinrich Heine Universität Düsseldorf, Universitätsstraße 1, 40225 Düsseldorf, NRW, Germany

Following, we give a brief introduction to B and Clojure in Sect. 2, and introduce the internals of our embedding—*lisb* library—in Sect. 3. In Sect. 4, we present *lisb*'s capabilities for tool development based on an automatic refinement tool and give our insights on how well our chosen representation synergises with widespread libraries. Further, the implementation of a small imperative DSL is described in Sect. 5. Implementation details are presented in Sect. 6 by transferring the technique to LTL formulas. Finally, we give our conclusions in Sect. 8.

1.1 Motivation

Overall, the development of *lisb* was driven by two experiments of our group:

The initial idea came during implementation of a case study on data validation of university curricula [49, 51]. Briefly summarised, the goal was to verify that all combinations of major and minor subjects at the faculties of Arts & Humanities and Business Administration & Economics at Heinrich Heine University Düsseldorf can be studied in a legal standard time (typically six semesters). One idea was to generate conforming timetables from scratch using a constraint-based approach [50].

A first version of *lisb* thus only covered the required B sub-language that contains expressions and predicates (but covered no state changes via variable substitutions) and was created with two design goals in mind: First, to address several shortcomings with the B language—in particular, the lack of convenience operators (such as let and if on the level of expressions instead of substitutions[1]) and an error-prone definition system that was used to avoid repetition of predicates and substitutions that are shared between several operations (see Appendix A for details)—and, second, to interact with (partial) solutions that the constraint solver provides. The main application was to transform the course information obtained from the electronic course catalogue into constraints that can be programmatically manipulated, combined and extended.

Later, another motivator was a student project aiming at translating Solidity contracts [14] to B machines. Thus, it was required to extend *lisb* to cover the entirety of the B language in order to capture state changes. This experiment helped to expose the potential for DSLs. At this point, *lisb* was mature enough so that more complex tools can be created on top of it. As mentioned above, an example is a tool that applies certain refinement steps in order to obtain an equivalent version of the machine that exposes more information during a specific static analysis (Sect. 4).

[1] During the project, ProB was extended to introduce these operators.

1.2 Additional contributions

The article is based on our MoDeVVa 2022 workshop paper [31] and extends it by

- providing a more inclusive background (Sect. 2),
- including lessons learned regarding library support (Sect. 4.3),
- a revised step-by-step tutorial of creating a similar embedding for the small LTL language, tools and DSLs (Sect. 6),
- a more detailed discussion of related work (Sect. 7),
- giving an outlook on challenges regarding a unified intermediate language for B and Event-B (in Sect. 8.1).

2 Background

In this section, we give a brief primer on the languages involved in the embedding, the B specification language and Clojure.

2.1 The B specification language

Roughly, the B methodology supports a correct-by-construction approach: Starting with an abstract, state-based model that specifies the desired behaviour, one adds more details by refining the model. Each refinement step is linked to the one before by proof obligations one has to discharge in order to show that the specification did not diverge. The models are written in the B language [2], which is based on first-order logic and set theory. A rather simple B model specifying Peterson's algorithm is given in Listing 1 (where we prefer standard mathematical symbols over the ASCII notation).

In the `SETS` clause, an enumerated set of statuses is defined (equivalent to enumerated types in programming languages). In the `CONSTANTS` clause, the constant `other` is introduced, which is constrained to the sequence `[2,1]` (equal to the relation $\{(1, 2), (2, 1)\}$) in the `PROPERTIES` clause. The state variables are declared in `VARIABLES` clause and initially assigned during the `INITIALISATION` (note that x is chosen non-deterministically and two possible initial states exist). The state is then manipulated as specified by the guarded substitutions in the `OPERATIONS` clause; here, it is encoded that either process must acquire the mutex before it may enter the critical section. Afterwards, it leaves the critical section again. The safety property that both processes may not be in the critical section at the same time is encoded as part of the `INVARIANT` clause. The invariant is typically verified by proof (e.g., using AtelierB [35]) or exhaustive model checking (e.g., using ProB [39]).

```
MACHINE Peterson
SETS Status = {noncrit,wait,crit}
CONSTANTS other
PROPERTIES other = [2,1]
VARIABLES pc,b,x
INVARIANT b ∈ 1..2→𝔹 ∧ x∈1..2
    ∧ pc ∈ 1..2→Status
    ∧ not(pc(1)=crit ∧ pc(2)=crit)
    ∧ ∀i(i∈1..2 ⇒
        (b(i)=⊤ ⇔ pc(i)=wait ∨ pc(i)=
            crit))
INITIALISATION
     b := [⊥,⊥] ‖ x :∈ 1..2
  ‖ pc := [noncrit,noncrit]
OPERATIONS
RequestCS(Proc) =
  PRE Proc∈1..2 ∧ pc(Proc)=noncrit
  THEN pc(Proc) := wait
    ‖ b(Proc) := ⊤
    ‖ x := other(Proc)
  END;
EnterCS(Proc) =
  PRE pc(Proc)=wait ∧ Proc∈1..2
      ∧ (x=Proc ∨ b(other(Proc))=⊥)
  THEN pc(Proc) := crit
  END;
LeaveCS(Proc) =
  PRE Proc∈1..2 ∧ pc(Proc)=crit
  THEN pc(Proc) := noncrit ‖ b(Proc)
      := ⊥
  END
END
```

Listing 1 B specification of Peterson's algorithm

Indeed, such a mathematical notation is very powerful and expressive. Different tools have been created or extended to exploit the high abstraction level in order to concisely capture constraints and perform data validation on large data sets. Examples include ProB [25], OVADO [1], PredicateB or DTVT [36]. Further industrial uses are described in [10].

We built *lisb* on top of the ProB [39] toolchain in order to programmatically interact with solutions. ProB is an animator, model checker and constraint solver for the B language that has been used in various industrial settings [10]. While written in Prolog, a Java API exists to interact with the interpreter core and even to write entire applications that embed B specifications [24, 33].

2.2 Clojure

Clojure is a functional programming language that runs on top of the JVM. It has facilities to interoperate with other JVM languages and, thus, code can call and be called from Java, Scala, Groovy, etc. We chose Clojure over other JVM languages because its strengths complement several weaknesses of B: Clojure offers a rich standard library that facilitates generation and processing of data from disk or other sources; it taps into the rich JVM ecosystem for library and tool support; and, most importantly, the rich macro system simplifies code transformation and empowers development of DSLs. Finally, all of Clojure's data structures are immutable by default (as required for parallel substitutions in B), which also eases transformation, re-combination of and interaction with parts of variable values, predicates or state machines. As Clojure is not a mainstream language, we will briefly introduce some key concepts here.

Syntax: function calls As a Lisp dialect, function calls are written as lists. For example, the call `(f a b)` is equivalent to calling `f(a, b)` in Java. In nested calls, arguments are evaluated before calling the function.

Code is data Clojure code is written as data structures of the language: The function call `(f a b)` above is simply a list of three elements, where the first element is a symbol that evaluates to a function object, and the other elements are the arguments. Similarly, part of the syntax are also vectors, such as `[1 2 3]`, or maps, such as `{:key1 :value1, "key2" 42}`. The property that code can be regarded as data allows easy programmatic transformation of code, which is usually achieved via macros.

Macros and Meta-Programming Macros are functions that do not take part in the normal evaluation rules: Their arguments are *not* evaluated. Instead, arbitrary code can be called to re-write the arguments and to produce new code that the macro-call is replaced with. Such macros facilitate DSL development (even of entire DSL stack, e.g., [27]).

A built-in macro that we use later is the threading macro `->>`. It takes a form and iteratively inserts it as the last argument of all preceding forms. As an example, `(->> a (b c) (d e))` is re-written to `(d e (b c a))`.

Macros can be significantly more complex than producing small re-orderings. A macro `b` might override the call to the addition in `(b (+ 1 2))` and instead return code that loads a constraint solving tool, re-writes the expression into an input format it accepts, and uses that tool to evaluate the expression instead of using Clojure's + operator. As arbitrary code can be called, a macro can perform very complex tasks: The functionality of a macro could even include program analysis tasks, such as type checking, or transformation, such as partial evaluation.

2.3 Internal versus external DSLs

With Clojure macros, one (typically) creates a so-called *internal* DSL. It is built on top of the hosting language, using the constructs it offers. Thus, a call to a DSL in Clojure must always be in parentheses, as the outermost form must be a macro or function call.

Another kind is the *external DSL*: Typically, it introduces its own syntactical rules and may differ significantly from the language that is used to process it. Thus, a specialised parser is needed in order to generate the (abstract) syntax tree for further processing. As an example, even the ASCII notation of B machines can be regarded as an external DSL.

2.4 Language workbenches versus Clojure macros

There are a number of *language workbenches*,[2] that include JetBrain's MPS [9], Xtext [8], or, specific to the B-method, Meeduse [28]. A new DSL can be defined in three parts: First, by a schema for the abstract representation of the DSL's elements (a so-called meta-model); Second, by an editor that generates the meta-model's elements (which often is a structured editor, and in some instances a graphical editor); And third, by a generator that transforms the instances of the meta-model into executable code (which is often done using templating languages).

An advanced language workbench will also provide IDE support for the external DSL, and—due to the editor—will not require the implementation of a parser. The workbench might also offer support for testing, debugging, etc.

Opposed to Clojure macros, a language workbench provides a more structured workflow for the creation of DSLs. A macro includes both the *definition* of the language element and the *generator*, i.e., the code it is re-written to. Since we consider internal DSLs for Clojure macros, a specialised editor is not required. However, we do not automatically gain additional support for error messages, e.g., due to syntactical errors or wrong argument types.

3 *lisb*—internals

The *lisb* library is, ultimately, an embedding of the B language in Clojure. Mainly, three representations of B constraints for different tasks are offered, which are presented in Sect. 3.1. Aside from the syntactical representations, the semantics of B are available by loading constraints or entire state machines in the ProB tool. How a user program might make use of this is discussed in Sect. 3.2.

3.1 Three representations

```
1  user=> (require '[lisb.translation.util :as util]
2                  '[lisb.translation.lisb2ir
3                  :refer :all])
4  user=> (def lmch  (util/b->lisb (slurp
5          "Peterson.mch")))
```

[2] For a more in-depth discussion, we refer to Martin Fowler's essay on language workbenches, see https://martinfowler.com/articles/languageWorkbench.html.

```
6  user=> (clojure.pprint/pprint lmch)
7   ;; pretty print, shortened
8  (machine :Peterson
9   (sets (enumerated-set :Status :noncrit
10         :wait :crit))
11  ...
12  (variables :pc :b :x)
13  (invariants
14   (member? :b (--> (interval 1 2) bool-set)) ...
15   (for-all [:i]
16    (member? :i (interval 1 2))
17    (or (<=> (= (fn-call :b :i) true) (= (fn-call
18          :pc :i) :wait))
19        (= (fn-call :pc :i) :crit))))
20  (init (parallel-sub (assign :b (sequence
21         false false))
22                      (becomes-element-of [:x]
23         (interval 1 2))
24                      (assign :pc (sequence
25                  :noncrit :noncrit))))
26  (operations ...
27   (:LeaveCS [:Proc]
28    (pre (and (member? :Proc (interval 1 2))
29              (= ...))
30         (parallel-sub (assign (fn-call :pc :Proc)
31                  :noncrit)
32                       (assign (fn-call :b :Proc)
33                  false))))))
34  user=> (def ir (eval `(b ~lmch))) ;; generate IR
```

Listing 2 Loading the Peterson Machine in *lisb*

lisb contains three representations of the B language that can be seen in Fig. 1, each with a different purpose. We first give a quick overview, and will discuss them in more detail afterwards:

- An *internal DSL* for B (and, more recently, Event-B [3, 5]) that can be used to express constraints and B machines. This DSL can be easily read, written and generated by (Clojure) developers. Sometimes, this DSL is referred to as "lisb code", as, for example, the function name `b->lisb` in l. 3 of Listing 2 indicates. This text, however, will always use the term "internal DSL".
- An *intermediate representation* (IR) of the mathematical concepts, which are represented as Clojure maps. This IR is, due to its verbosity, less readable (compared to the internal DSL); However, programmatic manipulation of such data structures is easier.
- An embedding of the ProB *Java API* [33], including the parser suite of ProB. This allows us to transform the IR into an AST that ProB can directly process without further parsing.

The internal DSL is implemented as a macro that re-writes DSL code to an expression that *evaluates* against the corresponding IR. The IR can be transformed into the parser nodes that make up the ProB AST. This AST can be (a) pretty printed into the standard ASCII notation of B machines, (b) transformed into the internal DSL or (c) directly be passed to functions in the ProB Java API.

In Listing 2, the machine from Listing 1 is loaded from disk. The `b->lisb` function will call ProB's parser, generate an AST and translate it into the internal DSL. Next, in

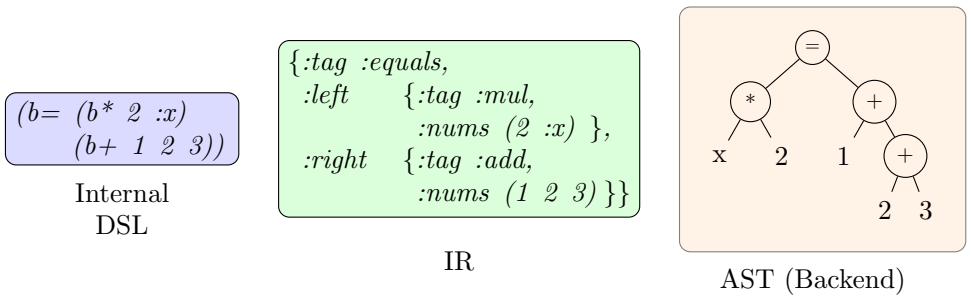

Fig. 1 Frontend, Intermediate Representation and Backend

```
1  new Start(
2    new APredicateParseUnit(
3      new AEqualPredicate(
4        new AMultOrCartExpression(
5          new AIntegerExpression(
6            new TIntegerLiteral("2"))
                ,
7          new AIdentifierExpression(
8            Collections.singletonList
                ("x"))),
9        new AAddExpression(
10         new AAddExpression(
11           new AIntegerExpression(
12             new TIntegerLiteral("1"
                  )),
13           new AIntegerExpression(
14             new TIntegerLiteral("2"
                  ))),
15         new TIntegerLiteral("3"))))
                ,
16     new EOF()));
```

Listing 3 Creating the Java AST for $x * 2 = 1 + 2 + 3$

l. 4, we ask for a pretty print of the internal DSL representation (which is shortened for brevity's sake). By wrapping the DSL code in the `b` macro and evaluating it (l. 23), we will obtain the IR (which is less readable and, thus, not shown).

Below, we will discuss these three representations bottom-up in more detail, in particular their advantages and shortcomings.

ProB Java AST The backbone of *lisb* is the AST library provided by the ProB parser. It is the argument type of many ProB API functions, e.g., when constraints should be solved or a model checking process is started. The parser nodes are represented as Java classes and are instantiated in Clojure (due to its interoperability with Java).

Unfortunately, the Java classes themselves are automatically generated by the parser generator tool SableCC [19] and are not intended to be constructed manually: First, the unwieldy code depicted in Listing 3 is required in order to create a small predicate such as `x*2=1+2+3` (cf. the *lisb* code in Fig. 1). Second, since AST nodes are automatically generated, the usage of many nodes is unintuitive (e.g., a singleton list is required in line 8 of Listing 3) in order to instantiate an identifier node. Third, as the nodes are mutable, inserting the same sub-tree in multiple locations of the AST is not allowed. Overall, this AST is not suitable for easy manipulation, is hard to read without a pretty print and is very hard to write.

Intermediate representation (IR) The IR is intended to address one of the shortcomings of the ProB AST: Its main goal is to ease programmatic processing and transformation. The equivalent IR of the code in Listing 3 is depicted in the middle box of Fig. 1.

The main difference is that the IR is a pure data representation (as nested maps), which offers the following advantages: First, it avoids encapsulation of the AST's information and, thus, yields a data literal that can be written and accessed without too much boilerplate. Second, we claim that the IR is more intuitive because it is based on the semantics of the actual operators in the language instead of grammar rules used for parsing. Third, as the IR is just data, one may re-use sub-trees without worrying to break something.

Scalar values (booleans, numbers, sets and strings) do not require to be wrapped and are simply the corresponding Clojure data literal. Variable identifiers are represented as Clojure keywords (identifiers prefixed with a colon, such as `:x`). All mathematical operators contained in the B language are represented by maps containing the key `:tag` for identification and an additional key for their operands. The representation of the mathematical sub-language for predicates and expressions is, in principle, agnostic to B and can be re-used for other formalism. Nodes of state machines, e.g., the invariant or operations clauses, are represented in the same way as operators; however, their representation is B-specific and aligns with ProB's AST nodes.

Internal DSL *lisb*'s internal DSL is built on top of its IR. It is designed to address the other shortcomings of the Java AST: The goal is to offer a Clojure-style representation that humans can read and write. However, programmatic manipulation was explicitly not a goal of this representation.

The foundation of the internal DSL consists of pure functions that generate the corresponding IR. Note that this allows mixing the internal DSL *with* the IR: The DSL parts will evaluate to IR, and since the IR is data, it will just evaluate to itself. All operators and machine clauses of B are available in

Table 1 Examples of *lisb* syntax

B (ASCII)	*lisb*	Intermediate representation	Description
42	42	42	Number
"foo"	"foo"	"foo"	String
x	:x	:x	Variable
{1,2,3}	#{1 2 3}	#{1 2 3}	Enumerated set
NATURAL	natural-set	{:tag :natural-set}	Set of natural numbers
1 |-> 2	(maplet 1 2)	{:tag :maplet,:left 1,:right 2}	Tuple
	(|-> 1 2)		Tuple (alternative)
1 + 2	(+ 1 2)	{:tag :add,:nums (1 2)}	Addition
a + b + c	(+ :a :b :c)	{:tag :add,:nums (:a :b :c)}	Addition
0 < x & x < 42	(< 0 :x 42)	{:tag :less,:nums (0:x 42)}	Less than
a: {1,2}	(member? :a #{1,2})	{:tag :member, :elem :a,:set #{1 2}}	Membership
#(x).(x > 42)	(exists [:x] (< :x 42))	{:tag :exists,:ids [:x],	Existential quantification
		:pred {:tag :less,:nums (:x 42)}}	
MACHINE foo ...	(machine foo...)	{:tag :machine,:machine-clauses...,	B machine
		:name :foo,:args []}	
OPERATIONS ...	(operations...)	{:tag :operations,:values...}	Operations clause
RequestCS(Proc) = ...	(RequestCS [:Proc]...)	{:tag :op,:returns [],:args [],	Operation definition
		:name :RequestCS,:body...}	
PRE X THEN Y	(pre (lisb X) (lisb Y))	{:tag :precondition,:pred (IR X)	Precondition
		:subs (IR Y)}	

the internal DSL with a one-to-one mapping to AST nodes in B. For example, in Fig. 1, we made use of the functions `b=`, `b+` and `b*`. The operator names are prefixed with `b` in order to avoid name clashes with the default Clojure core functions (so that they remain available). However, a separate `b` macro will replace all instances of =, + and * in its argument by the corresponding functions that generate the IR. Thus, `(b (+ 1 1))` will not yield 2, but instead the IR that represents this addition.

An excerpt of the syntax of the internal DSL is given in Table 1.[3] Note that no new semantics is defined; all functions of the internal DSL map directly to B operators and clauses. In fact, the exact same AST nodes ProB uses to represent B machines are generated.

There are however minor differences to ease usage for Clojure developers. Some operators have multiple aliases (for example, we deemed it sensible to include the name `partial-surjection` additionally to providing the ASCII symbol >+>). Others take a variable number of arguments (such as <) to accommodate a Clojure-style of writing predicates. In a pre-processing step, it is replaced by a proper conjunction of predicates, for example, `(< 1 2 3)` will be internally re-written to $1 < 2 \wedge 2 < 3$.

[3] A full overview can be found https://github.com/pkoerner/lisb/blob/master/doc/Lisb.md.

3.2 Architecture overview and user programs

In this subsection, we will first discuss how ProB's advanced capabilities are embedded in *lisb*. Afterwards, we will describe how a user program is situated in this stack and give an overview about expected uses.

3.2.1 The re-translation module

The ProB tool can be used to solve constraints, and to animate and model checking state-based specifications. The different representations in *lisb*, that we discussed above, seamlessly integrate with the ProB Java API.

In Fig. 1, we wrote the constraint `(b= (b* 2 :x) (b+ 1 2 3))` in the internal DSL. *lisb* provides the means to solve it (as the IR can be translated to the AST), and to obtain the solution mapping x to 3. Similarly, the IR of a state machine can be loaded in ProB and model checking tasks can be executed. Step-wise exploration of the state space is possible as well.

The ProB Java API has different representations of such solutions, traces, states and model checking results as Java objects. In order to work with them, for example, including calculated solutions in new constraints, we need to translate them back into a Clojure-like representation.

This re-translation module consists of a small translation layer that transforms aforementioned Java objects back into

Clojure data. For numbers, strings, or enumerated sets, this is straightforward and the default Clojure literals are used. Yet, sets may be infinite and be stored as a symbolic value in ProB (for example, the set containing all even numbers). Though they do not have a corresponding value in Clojure, they can be translated to the corresponding internal DSL or IR snippet.

3.2.2 User program

lisb is a library that is intended to be included in a user program. Such user program will typically be situated as depicted in Fig. 2.

Fundamentally, *lisb* serves as an intermediate layer between the user program and the ProB Java API: It can be used for (a) accessing the internal DSL in order to create models or generate constraints; (b) obtaining the IR to transform specifications; or (c) interacting with ProB to calculate solutions and use the results in the program.

On top of *lisb*, users can implement their own DSLs (User DSL in Fig. 2) (in Clojure, or, in principle, any JVM language). Such a user DSL might directly generate the IR representation, or make use of any DSL that already is implemented.

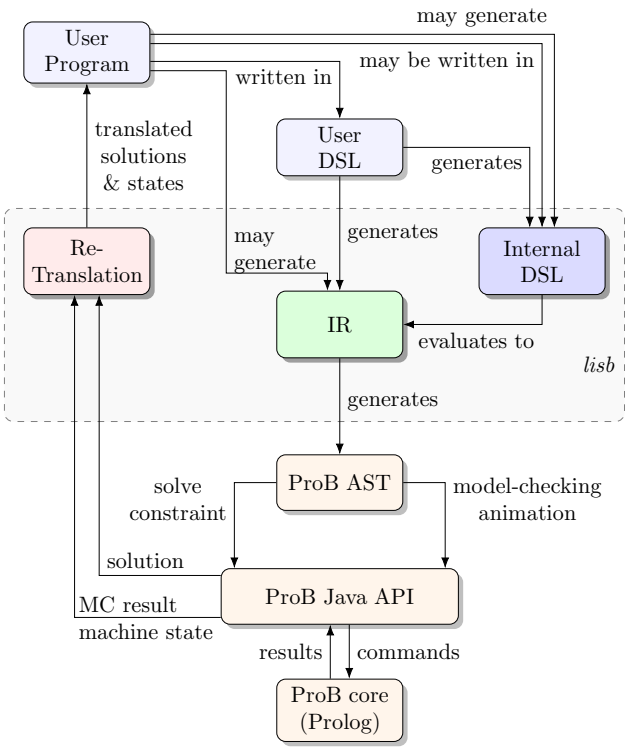

Fig. 2 Architecture of a program using *lisb*—arrows denote possible data flows

3.2.3 Potential use cases

In the following, we want to give our expectations of how *lisb* could be used. Note that this does not cover all possibilities: the point of *lisb* is that it facilitates implementation of *any* tool that needs to re-structure or generate a formal (B) specification. Yet, at the core of each tool will be at least one of the following activities:

Modelling activities A modelling expert (in B) who is not acquainted with Clojure already would not be interested too much in *lisb*: The internal DSL would be a new syntax to learn, and they would probably not be very comfortable creating a custom DSL. For them, *lisb* would not offer an obvious benefit. However, modellers experienced in Lisp-like languages would use a mixture of the internal DSL and, potentially, write their own DSLs. Regardless of expertise, writing or reading the IR is not necessary for any modelling activity.

The main benefit is that a formalism can be extended syntactically without the need to wait for tool developers to integrate changes. Examples specific to B are given in Appendix A.

Specification transformation Some users may be interested in transforming existing constraints and specifications; We think this is especially the case for tool developers. A typical workflow for them would include to (a) load a machine from disk and transform it into the IR, and (b) write some IR-to-IR transformation. The resulting tool might combine the IR with snippets of *lisb*'s internal DSL or a custom DSL, as they can be mixed arbitrarily. Resulting models can be loaded and animated in combination with the original specification; simply be saved to a new B machine file; or the transformation can be validated using Clojure's testing framework. More detail about how *lisb* can be used like this is presented in Sect. 4.

Specification generation and DSLs Finally, it can be interesting to capture data from external data sources and use *lisb* to generate constraints (and solve them), or to generate B machines. Again, one would generate bits of the internal DSL, evaluate them to the IR, and, finally, re-combine such IR snippets programmatically. User DSLs would typically be implemented so that they are re-written (or generate directly) a lower-level DSL or *lisb*'s internal DSL. Technically, it can make sense to directly emit IR code; but the translation rules will be less readable. How a DSL might be implemented is outlined in Sect. 5.

Runtime embedding Another possible use case is that the ProB constraint solver is queried at runtime. One way is to execute a loaded machine based on incoming data (e.g., from a web application, a socket, a user, …). This mimics the style of embedding B machines in applications, which has been used in Java to control trains [23].

Similarly, one can solve constraints at runtime and work with the results. As an example, *lisb* is used to create its own documentation[4]: The documentation includes a static site generator with different types of code blocks (Clojure, B or *lisb*'s internal DSL). Constraints included in the code blocks are parsed, evaluated via *lisb* and both the ASCII B pretty-print and the solutions are injected. This allows documentation in the style of literate programming. It can also be used without exposing *lisb* or Clojure at all, with the goal to document the semantics of ASCII B expressions.

```
1 VARIABLES pc1, pc2, b1, b2, ...
2 INVARIANT b1∈𝔹 ∧ b2∈𝔹 ∧ pc1∈Status
      ∧ pc2∈Status ∧ ...
3 OPERATIONS ...
4 LeaveCS1 = PRE pc1 = crit
5            THEN pc1:=noncrit ‖ b1:=⊥
                 END
6 LeaveCS2 = PRE pc2 = crit
7            THEN pc2:=noncrit ‖ b2:=⊥
                 END
8 END
```

Listing 4 Excerpt of desired re-writes

4 Case study: machine transformation

In the following, we want to give a concrete example on how a machine-to-machine transformation might be implemented using *lisb*. The exact rules we apply are not relevant for this (we will have to refer to a paper that describes the re-writes in detail [30]); instead, we want to highlight some synergies of the components above that might not be obvious at first glance.

4.1 Motivation and idea

ProB offers an implementation [15] of partial order reduction (POR) [44], a state space reduction technique. We will skip complex details of how this technique works and focus on points relevant to the implemented transformation.

Before the actual POR techniques are applied, the implementation in ProB has an analysis phase of the model. The ultimate goal of the analysis is to locate as many "independent" pairs of operations as possible. Independent operations will always commute, i.e., first applying operation α and then β will yield the same state as β and α; further, they cannot disable each other. A fast syntactical approximation is: If two operations do not read or write a shared variable, they must be independent.

In the description of Peterson's algorithm in Listing 1, the three operations `RequestCS`, `EnterCS` and `LeaveCS` all write (different values) to the state variable `pc`. Thus, no independent operations will be found by the aforementioned approximation, and POR will yield no reduction.

However, all three operations accept only two possible parameter values, 1 and 2. Further, the state variables `b` and `x` are always collections of size 2. This information can be determined statically.

The idea, thus, is to re-write the model as follows (mostly as a data refinement): If we can determine that there is only a finite number n of possible parameter values for an operation, we transform this operation into n operations that instantiate all possible values. Further, if we can determine that a state variable is a function with a fixed-sized finite domain, we will generate a new state variable for each mapping. Similarly, sets over a known finite domain are split into a number of boolean values, indicating whether they are contained or not (which is often referred to as bit-blasting). Of course, all accesses of the original function have to be replaced as well (for which we implemented the rules Kodkod applies for its SAT encoding [53]).

The resulting machine would look like the excerpt in Listing 4: The original `LeaveCS` operations allowed two parameters, so two operations are generated. Further, both `pc` and `b` mapped the numbers 1 and 2 to a Boolean or Status value. Instead, the new model has two new variables each, `pc1` and `pc2` that stem from `pc`, and `b1` and `b2` from `b`.

If we now consider the operations `LeaveCS1` and `LeaveCS2`, we can determine only from the accessed identifiers (`pc1` and `b1` vs. `pc2` and `b2`) that these operations must be independent of each other. This equivalent model will now yield some state space reduction without changing the implementation of POR.

4.2 Implementation

The concept of *lisb*'s usage in this transformation tool is given below. To no surprise, a number of re-writes have been implemented, based on the intermediate representation. Though, a relevant question is: *How is it possible to obtain the relevant static information?*

After all, we need to determine that domains of functions and parameters are finite. In the case of the Peterson machine, we might encounter, for example, the IR for Proc $\in$ 1..2. We could implement a special case for such membership constraints; but, ultimately, we require a constraint solver for more complex machines. This is where we can make use of the integration with ProB!

The idea is that we can simply evaluate constraints guarding operations or for set cardinality in the context of the original machine. We just have to transform the IR snippet into ProB's AST and ask the constraint solver for all solu-

[4] https://pkoerner.github.io/lisb-doc/

tions. This combination of the *specification transformation* and *runtime embedding* aspects allows easy implementation even of complex tools.[5]

Summarising, the high-level workflow of this transformation makes use of the following features of *lisb*:

1. The original machine is parsed and the resulting AST is transformed into the IR (for programmatic manipulation).
2. To determine whether set variables have a fixed-size domain, the machine is loaded via the ProB Java API and the set cardinality constraints are solved. Via ProB, we also gain type information (e.g., elements of custom-defined sets).
3. In order to split the operation, the IR representation of the guards are fed to the constraint solver in order to find all solutions for operation parameters.
4. With the information gained in the previous steps, the actual IR transformation is performed in multiple steps. First, generating the new operations that will eliminate parameters. Afterwards, the bit-blasting of set variables and re-writing their usages in expressions. Finally, a simplification step eliminates redundant conjuncts, assignments, etc.
5. The pretty printer of ProB's parser suite is used to emit a new B machine.

Based on *lisb*, a prototype of such a rather large automatic refinement tool that is capable to work with complex machines (such as the one discussed in [38]) can be written in about 750 lines of Clojure code,[6] of which about 60 lines are devoted to a small DSL.

4.3 Lessons learned: library synergy

During the work on this refinement tool, we found that the IR harmonises greatly with a couple of libraries: first, Specter offers a so-called navigator abstraction allowing us to specify paths in the nested IR data structure which then can be used to extract and transform information; second, the pattern matching library core.match, which allows us to define transformations based not only on specific node types, but also on defined additional properties; third, the meander library that offers a term-rewriting engine.

[5] An implementation directly in ProB is possible, but would be significantly harder. We used this prototype to determine whether it is worth the hassle.

[6] The tool can be found at https://github.com/JanRossbach/fset.

4.3.1 Data transformation—specter

The IR harmonises—because it is plain data—with widespread transformation libraries in Clojure, such as Specter.[7] For example, Listing 5 shows that one can retrieve the IR of all guard conjuncts (without duplicates) in a few lines of code.

```
user=> (require '[com.rpl.specter
          :as s])
user=> (defn TAG [t] (s/path
         #(= (:tag %) t)))
user=> (def CLAUSES (s/if-path
         (s/must :ir) [:ir :clauses]
         [:machine-clauses]))
user=> (defn CLAUSE [name] (s/path
         [CLAUSES s/ALL (TAG name)]))
user=> (->> ir
            (s/select [(CLAUSE
          :operations) :values s/ALL
          :body :pred :preds])
            (apply concat)
            (map ir->b)
          ;; pretty-print
            set)
#{"Proc:1..2",   "pc(Proc)=noncrit",
"x=Proc or b(other(Proc))=FALSE",
"pc(Proc)=wait"}
```

Listing 5 Retrieving the IR of all unique guard conjuncts from the Peterson machine (continues Listing 2)

One can generate a modified copy by simply calling Specter's `transform` instead of `select`. Using the path from Listing 5, one would transform all guards based on an argument function. Naturally, the described refinement tool has to transform more parts of the machine than just the guards.

Specter also has a walking feature, that recursively searches the data-structure for sub-structures that match a particular pattern and then applies the transformation function to the matching structure.

4.3.2 Pattern matching—core.match

```
user=> (require '[clojure.core.match
         :refer [match]])
user=> (defn simplify-formula [formula]
           (match formula
         {:tag :not :pred {:tag :not :pred p}} p
                          _ nil))
user=> (defn simplify-ir [ir]
           (s/transform [(s/walker simplify-
         formula)] simplify-formula ir))
user=> (-> "#FORMULA x=y & not(not(y=x))"
           b->ir
           simplify-ir
           ir->b)
"x=y & y=x"
```

Listing 6 Applying a simple transformation rule with core.match and Specter

[7] https://github.com/redplanetlabs/specter

The `core.match` library is an implementation for pattern matching in Clojure and, in this context, allows us, to write conciser transformation rules than a more native construct like multi-methods.

In combination with `core.match`, Specter's walker can be used to make IR transformation and simplification trivial. In Listing 6 one can see an example of codifying a single simplification rule of $\neg\neg P \Leftrightarrow P$ for any predicate P, in order to rewrite the IR for $\neg(x \neq y)$ into $x = y$. This can easily be extended to very elaborate transformation rules which, only take a single line of code each in the match statement.[8]

When using this kind of setup, one has to be careful to not use excessively large IR for performance reasons, but that can usually be easily avoided.

4.3.3 Term rewriting—meander

```
1  (require '[meander.epsilon :as m])
2  (require '[meander.strategy.epsilon :as s])
3
4  (defn replace-var [data old-name new-name]
5    (let [f (s/top-down-until #(m/find % ('for-all
6          [_ ... ~old-name . _ ...] ?body) true)
7              (s/match ~old-name new-name
8                       ?value ?value))]
9      (f data)))
```

Listing 7 A small renaming tool using meander

The last library we employed in a *lisb*-related project is meander.[9] Specter was very useful for extracting information that is located at a given path, i.e., *in a specific position* of the machine. One example is to find or manipulate all elements in the `OPERATIONS` clause. The main strength of meander, on the other hand, is to locate or re-write terms at *any* position of the machine.

As an example, consider a small tool that implements a refactoring operation on a B machine that re-names a specific variable. A naive solution would replace, for example, all occurrences of `:x` with `:y`. However, there are also operators that introduce a local scope for variables, such as the universal and existential quantification, as well as the set comprehensions or lambda expressions. If such an operator introduces a variable also named `:x`, it should *not* be renamed within its scope.

The code in Listing 7 outlines how such a refactoring tool might be implemented on top of the internal DSL; an implementation on top of the IR would be very similarly. The `match` expression in l. 6–7 specifies how the replacement should occur: If the old variable name is found, instead the new variable name should be inserted. All other values will remain as-is. This is wrapped inside a so-called *strategy*: The parse tree is walked top-down, until a `for-all` expression is found that introduces the old-name as any element of its bindings (the underscores and dots are part of meander's syntax for this and do not indicate an omission). If such an expression is found, its sub-tree will not be explored any further.

In consequence, if we include all operators that introduce a fresh scope in the `find`-expression in l. 5, we are able to implement a small refactoring tool in very little code.

5 Case study: algorithm description language DSL

In the following, we give an impression of how to implement a DSL in *lisb*. We chose to re-implement a (slightly simplified version of) an external DSL for algorithm description (ADL) as an internal DSL in *lisb*. The ADL was originally translated to the Event-B notation [13]; Event-B is similar to B, but does not contain while-loops or if-then-else statements [37]. For a fair comparison, we will not use these constructs in *lisb* either.

ADL Overview

The ADL roughly aligns with imperative pseudo-code. The constructs the external DSL offers are:

- variable assignments,
- sequential composition of statements,
- assertion statements,
- if-then-else statements,
- while-loops with loop invariants.

For expressions, strings with Event-B formulas are used. A small example of this language can be seen in Listing 8, depicting a small algorithm that calculates the multiplication of two numbers via repeated addition (or doubling one number and halving the other if the latter is even). The same example can be written in the internal ADL-DSL that we developed, as is shown in Listing 9.

Translation Idea

The translation is not straightforward because the branching rules of conditionals and loops have to be simulated. The main idea is that each construct in the language corresponds to (at least) one operation in the resulting state machine. If-statements and while-loops will be modelled by two operations: one with the if- or while-condition as a guard, and another with the negated condition. Further, an artificial program counter variable (PC) is added to the model. Each operation then will be guarded by a conjunct $\text{pc} = x$, testing whether the "virtual instruction pointer" is currently at this position in the code.

[8] More elaborate examples: https://github.com/JanRossbach/fset/blob/main/components/simplifier/src/hhu/fset/simplifier/core.clj

[9] https://github.com/noprompt/meander/

```
procedure(name: "mult") {
    argument "x", "NAT"
    argument "y", "NAT"
    result "product", "NAT"
    precondition "x >= 0 & y >= 0"
    postcondition "product = x * y"
    implementation {
        var "x0", "x0 : NAT", "x0 := x"
        var "y0", "y0 : NAT", "y0 := y"
        var "p", "p : NAT", "p := 0"
        algorithm {
            While("x0 > 0",
                invariant: "p + x0*y0 = x*y") {
                If("x0 mod 2 /= 0") {
                    Then("p := p + y0")
                }
                Assign("x0,y0 := x0/2,y0*2")
        }
        Assert("p = x*y")
        Return("p")
}}}
```

Listing 8 Multiplication example from [13]

```
(adl :Multiply
  (var :x (in :x nat-set) 5)
  (var :y (in :y nat-set) 3)
  (var :p (in :p nat-set) 0)
  (algorithm
    (while (> :x 0)
      (assert (= (+ :p (* :x :y)) (* 5 3)))
      (if (not= 0 (mod :x 2))
        (assign :p (+ :p :y)))
      (assign :x (/ :x 2), :y (* :y 2)))
    (assert (= :p (* :x :y))))))
```

Listing 9 Example usage of algorithm DSL in *lisb*

```
(defn assign [pc & kvs]
  {:pc (+ pc 1)
   :ops (fn [jump?]
          (let [opname (str "assign" pc)
                pc' (if jump? jump? (+ pc 1))]
            [`(bop ~opname []
                   (bprecondition
                    (b= :pc ~pc)
                    (bassign ~@kvs :pc ~pc')))]))})
```

Listing 10 Implementation of assignments in *lisb*'s algorithm DSL

Typically, one would simply increase the PC with each operation. However, it might be the last assignment in a then-branch (and one has to skip over the else-branch); or it might be the last assignment in a while-loop (where one might have to jump back to the evaluation of the condition). Thus, one issue is that, for example when generating the path with a negated if-condition, one does not know yet how many instructions the else-branch requires.

We further sketch the idea based on the translation of assignments in Listing 10. The `assign` function take the next PC value that is available and a sequence of variable names and expressions. It will generate one operation only (ll. 6–9): the guard will verify that the PC enables the operation, and the variables are assigned. However, it is not yet known that the PC value should be after the assignment, as a jump forward (in case of if-statements) or backward (in case of while-loops) in the program is required. Thus, instead of returning that operation immediately, *a function* is returned (l. 3) that can be instantiated with the correct next PC value, indicating whether a jump to a certain position is required or that the PC can simply be incremented. In the outer map, the next available PC is contained as well (l. 2): as the assignment contains only one instruction, we can simply increment the original PC.

If-statements and while-loops are more complex and will generate more than one operation (two for the positive and negated condition, and also including all the operations for their bodies). These constructs also manage the correct instantiation of the PC. Finally, assertions add an operation (similar to an empty assignment) as well as a conjunct containing their expression as an invariant: This conjunct has the form of $\text{pc} = \text{loc} \Rightarrow \text{expr}$. Finally, all generated operations and invariants have to be merged into a single machine.

Evaluation The entire code for the ADL in *lisb* can be written in about 100 lines of Clojure.[10] As it is an internal DSL, no new parser is required.

In order to understand the code, only little knowledge of Clojure is required — in particular, the applied compiler construction techniques for introducing the program counter are harder to grasp than the actual implementation. We think that any capable programmer who has such knowledge would also be able to write such a DSL.

6 Technique transfer: LTL pattern languages

In this section, we show implementation details by transferring *lisb*'s technique to linear temporal logic (LTL). The goal of this section is to show a complete implementation of the concepts of *lisb* based on a smaller language than B. It also demonstrates the amount of Clojure knowledge that is required to create or understand such a tool; and might give better insights on the feasibility of an implementation in other languages.

LTL is a formalism that reasons about the *temporal* behaviour of programs. A basic building block is the atomic proposition, i.e., any predicate that argues about a singular program state. If the atomic proposition holds true, then the state will satisfy the formula.

There are many dialects of LTL. Here, we consider a very small subset of operators that is complete, i.e.:

- The *next* operator $\bigcirc\phi$, which holds true iff the formula ϕ holds in all successor states of the current state.

[10] The implementation can be found at https://github.com/pkoerner/lisb/blob/f22cb5962b87c047f6ab107dcee28f81d3b8aaf0/src/lisb/adl/adl2lisb.clj.

- The *until* operator $\phi_1 U \phi_2$, which holds true iff the formula ϕ_1 holds — on all pathes — until at least once ϕ_2 is true. ϕ_2 must be reached at least once on all pathes.

LTL suits particularly well as an example: first, it is a formalism that is well-known; second, it consists of a small amount of required operators; third, plenty convenience operators exist which typically are not part of definitions in order to ease proofs; and fourth, larger formulas are very hard to understand entirely, so some kind of DSLs or patterns are certainly useful (see, e.g., [17, 20]).

This example follows the textbook definitions of Baier and Katoen [6, Ch. 5] that we will repeat here — first, a limited set of LTL operators is considered and an internal DSL and IR is defined. Passing the IR to a backend is then simulated by simply pretty printing the formula. Afterwards, the language is extended by adding operators in two different ways. As an example for an IR-level transformation, we implement a tool that generates a positive normal form. Finally, as a DSL, we show how to implement LTL patterns on top based on the patterns by Dwyer et al. [17].

Considered LTL-Flavour

First, we adopt the definition of an LTL formula ϕ to be of the form [6, Ch. 5]:

$$\phi := \text{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \bigcirc\phi \mid \phi_1 U \phi_2$$

Here, a may be any atomic proposition that is given as a string.

Note, in particular, that this definition does not include common operators such as the globally, finally, weak until and the release operators; further, the negation and conjunction together are functionally complete set of logic operators — operators such as disjunction, equivalence, etc. are derived from them. This is done for shorter and more elegant proofs in Baier and Katoen's book.

In the same spirit, we define this minimal core in Sect. 6.1 and simulate a tool backend for this in Sect. 6.2. Other common operators will be added as a small DSL on top of the LTL core language in Sect. 6.3. As more complex LTL formulas are hard to understand, we will stack another DSL on top in Sect. 6.4. We also demonstrate how an LTL-to-LTL transformation can easily be implemented in Sect. 6.5.

6.1 Internal DSL definition and IR data generation

In a first step, we define functions that generate IR code from the basic operator set. These functions will serve as our internal DSL. Note that we do not employ Clojure macros here. The reason is that this DSL is so small, that we do not require analysis of the code. The four functions in Listing 11 correspond to one of the operators each; Instead of a dedicated *true* value, we use the Clojure boolean value `true`. Atomic propositions are also not created by calling a function but instead are a primitive in the proposed internal DSL. For example, we may construct a representation for the LTL formula $\bigcirc a = 1$ by calling `(next "a=1")`, and we will obtain the IR `{:tag :next, :ltl "a=1"}`.

```
1 (defn ltl-and [x y]
2   {:tag :and, :lhs x, :rhs y})
3 (defn negate [x]
4   {:tag :not, :ltl x})
5 (defn next [x]
6   {:tag :next, :ltl x})
7 (defn until [x y]
8   {:tag :until, :lhs x, :rhs y})
```

Listing 11 Code that generates the IR

```
1  (defmulti pp #(or (:tag %) (class
       %)))
2  (defmethod pp Boolean [x] (str x))
3  (defmethod pp String [x] x)
4  (defmethod pp :next [x]
5    (str "○(" (pp (:ltl x)) ")"))
6  (defmethod pp :until [x]
7    (str "(" (pp (:lhs x)) ") U ("
8             (pp (:rhs x)) ")"))
9  (defmethod pp :and [x]
10   (str "(" (pp (:lhs x)) ") ∧ ("
11            (pp (:rhs x)) ")"))
12 (defmethod pp :not [x]
13   (str "¬(" (pp (:ltl x))")"))
```

Listing 12 Pretty printer code

6.2 Obtaining formula strings

The next step is the translation of the IR to a pretty print of the formula. Such a string representation usually can be passed to most model checking tools; it can also be regarded as an *external DSL*.

The code to generate this pretty print is given in Listing 12—it is a somewhat naive implementation. The pretty printing function `pp` is defined as a multimethod in Clojure; it chooses the implementation of the function based on the keyword stored under `:tag`, or, in case no value is found, it uses the class of object for the dispatch. This brings the advantage that the IR can be extended later (as in Sect. 6.3).

The definitions of `pp` all are small and rather straightforward: the Booleans *true* and *false* are transformed into a string; Strings (i.e., atomic propositions) are kept as-is; and the other operators generate the corresponding symbol as well as parentheses to avoid clashes in the operator precedence. As an example, e.g., calling `(pp`

```
(defn finally [x]
  (until true x))

(defn globally [x]
  (negate (finally (negate x))))
```

Listing 13 Introducing convenience operators

`(until (negate "a=1") "foo=42"))` first creates the IR `{:tag :until, :lhs {:tag :not, :ltl "a=1", :rhs "foo=42"}}` and finally constructs the pretty-printed formula `"(¬(a=1) U (foo=42)"`.

6.3 Extending the language

In the following, we show how to extend the language in two ways: First, operators can be introduced only to the language frontend and map to existing operators. This approach is typically used when a new DSL is created. Second, if the tool backend offers support for new constructs, it can also be sensible to include them in the IR and extend the backend translation.

Frontend extension

As mentioned before, more operators than included in the basic core are widespread. Examples include the *finally* operator $\Diamond\phi$, that holds true iff on all possible paths in a program, the formula ϕ will be satisfied eventually by some program state; and the *globally* operator $\Box\phi$, that holds true iff on all possible paths in a program, the formula ϕ will be satisfied in every single program state.

We can define the *finally* operator in terms of the small core language we already implemented; and the *globally* operator in terms of *finally*. The corresponding rules are: $\Diamond\phi = \text{true}\ U\ \phi$ (ϕ has to hold true at some point, and we do not care about what happens before), and $\Box\phi = \neg\Diamond\neg\phi$ (there cannot be a point in time when ϕ does not hold true).

Let's assume that we do not wish to extend our IR for these operators, as if they only made sense in our domain. Instead, we will rewrite them using the definitions above.[11] In Listing 13, one can see the small implementation of these two operators. Note that even though *globally* is defined in terms of *finally*, the formula that is generated will *not* make direct use of the *finally* operator, as it will be re-written to the *until* first.

Backend extension For a second approach, we argue that the operators we want to add are more general and, thus, we want to include them in the IR. This is even more worthwhile if the underlying tool supports a straightforward translation. In Listing 14, we follow this approach for the weak until operator (which is similar to the until operator, but ϕ_1 might hold forever and ϕ_2 might not occur) and logic disjunction.

```
;; internal DSL functions
(defn weak-until [x y]
  {:tag :weak-until, :lhs x, :rhs y
     })
(defn ltl-or [x y]
  {:tag :or, :lhs x, :rhs y})

;; extending the backend
(defmethod pp :weak-until [x]
  (str "(" (pp (:lhs x)) ") W ("
           (pp (:rhs x)) ")"))
(defmethod pp :or [x]
  (str "(" (pp (:lhs x)) ") ∨ ("
         (pp (:rhs x)) ")"))
```

Listing 14 Extending the IR

Note that, first, we implement functions for the frontend that creates the IR nodes in order to expose them to the user. It might also be valid to add nodes to the IR that the user cannot directly instantiate—an example is a performance-specialised node that is generated during a pre-processing step. However, the second part which extends the backend is important as the backend must be able to process the entirety of the IR; otherwise, an extra IR-to-IR transformation is necessary so that unsupported nodes are eliminated.

6.4 Stacking DSLs: Dwyer patterns

While the formal semantics of LTL is clear, nested LTL formulas often become hard to understand for humans. Thus, several abstractions (or patterns) have been suggested, which can be regarded as DSLs. Examples include the patterns identified by Dwyer et al. [17]. Below, we will show how to implement these.

In the "response pattern", four atomic propositions P, S, Q and R are considered. We implement three versions of the S *responds to* P pattern. (i) The global version: every time P occurs, S must hold eventually. The other two versions make us of R (and Q) to define a scope in which the response must happen: (ii) The response must only occur before R is observed; And (iii) that the response always occurs after Q was observed, but before R happened (but such an interval may occur again).

The LTL formulas for these versions are [17]:

(i) $\Box(P \Rightarrow \Diamond S)$
(ii) $(P \Rightarrow (\neg R\ U\ (S \vee \neg R)))\ U\ (R \vee \Box\neg R)$
(iii) $\Box((Q \wedge \circ\Diamond R) \Rightarrow (P \Rightarrow (\neg R\ U\ (S \wedge \neg R)))\ U\ R)$

[11] This is similar to adding the if-then-else operator in Sect. A.2.

While the first formula can be quickly understood, the other two are significantly less intuitive. We will simply assume that they are correct (as it does not make any difference in the implementation technique).

For brevity and better alignment with the given formulas, we create a small DSL: first, symbolic aliases are defined for better alignment (such as `(def □ globally)`). Further, we define the logical implication $x \Rightarrow y$ by re-writing it in the frontend to the formula $\neg x \ \wedge \ y$. The Dwyer patterns then are implemented by simply writing them in prefix notation. On top, we wrap this in another light DSL that makes instantiations more readable, for example, `(dr S :responds-to P :between Q :and R)` (for "S responds to P between Q and R"). The full code and example instantiations are given in Listings 18 and 19 in Appendix B.1.

6.5 Transformation implementation

In [6, Def. 5.20] a normal form (more precise, a Weak-Until positive normal form) is specified. Baier and Katoen then show that any LTL formula can be transformed into this normal form. The main idea of this normal form is that negations only occur in front of atomic propositions and not in front of LTL operators. More formally, the idea is to transform any formula in our current IR of the form:

$$\phi := \text{true} \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \boldsymbol{\neg\phi} \mid \bigcirc\phi \mid \phi_1 U \phi_2 \mid \phi_1 W \phi_2$$

into a formula of the form:

$$\phi := \text{true} \mid \textbf{false} \mid a \mid \boldsymbol{\neg a} \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \bigcirc\phi \mid \phi_1 U \phi_2 \mid \phi_1 W \phi_2$$

The basic idea exploits the duality of operators to push negation of operators inwards, towards the arguments. A few transformation rules suffice for this task [6, Def. 5.20]. An example is De Morgan's law $\neg(\phi \wedge \psi) = \neg\phi \vee \neg\psi$. The other rules can be found in Appendix B.2.

The implementation[12] is done in two steps: first, the data structure is simply traversed (by the `traverse` function). Second, if a negation is encountered, it is pushed inwards (by the push-negation-inwards (`pni`) function) according to the rules above. The full code and an example can be found in Listings 20 and 21 in Appendix B.

The `traverse` function is quite simple: Boolean and String values are kept as they are; the unary next operator recurs on its single operand; and all binary functions traverse both operands. If a logical negation is encountered, however, the `pni` function is called.

The `pni` function actually encodes the re-writing rules. For String literals, a negation layer is added; Boolean values are immediately negated; and in case a second negation is encountered, both are deleted and standard traversal is continued. The other functions also fall back to the `traverse` function after one re-writing step.

6.6 Discussion

Note that up to here, we were able to give the entire implementation and only introduced very few Clojure-specific constructs. Asides from using (open) multimethods instead of using a (closed) switch-statement, any programmer should be able to follow the definitions (though the syntax might be unfamiliar).

This admittedly changes if transformations and DSLs become more complex: One could think of a DSL that hides most parentheses from the user and provides an infix notation. Such a call could be `(ltl ¬ □ ((a ∧ ¬ b) W ○ ¬ c))`. Then, analysis of this language (or, in the case of the refinement tool in Sect. 4, analysis of the specification itself) would require an additional parsing stage (as reordering of the elements according to the operator precedences is required). Additionally, as parentheses in Clojure overlap with function calls (e.g., in `(a ∧ ¬ b)`, the symbol `a` should not be called), the use of a macro now is required.

In such cases, the developer requires significant knowledge of Clojure's standard library and the macro expansion mechanism. While we think a software engineer can become acquainted with the necessary features in a relatively short timespan (a couple of weeks), the required code will be significantly more complicated than what we presented here. A DSL design tool for Clojure that only makes use of a small subset of Clojure could assist developers and domain experts here.

7 Related work

High-level formalisms have already been embedded into programming languages: as already discussed, we drew inspiration from αRby [41], as well as from PlusCal [34]. However, these tools seem to be more tailored towards modelling experts rather than tool developers who have to examine and interact with specifications on a more fine-grained basis.

PlusCal PlusCal is an imperative, pseudo-code-like language that allows embedding of arbitrary TLA$^+$ expressions, which can be mathematical formulas. It is particularly useful to let programmers with little to none modelling experience

[12] The basic concept is similar to the refinement tool in Sect. 4, but the example here is significantly smaller.

express specifications of (concurrent) algorithms. The language then can be translated into TLA$^+$ and tools such as TLC can process the specification. Similarly, *lisb*'s ƀ macro (see Sect. 3.1, internal DSL) allows translation of idiomatic Clojure code to B. However, the main difference is that the PlusCal language is neither intended to be extended by the user, nor that the obtained model can be inspected or transformed.

αRby At the first glance, the core concept of αRby and *lisb* seem very similar, and they share many advantageous approaches: an internal DSL is embedded into a host language so that no additional parsing tools are required; both languages can be easily extended; solutions that the solver provides can be processed for further constraint solving tasks; and, finally, external data from disk, network, etc. can be transformed into a format suitable for the formal language and its tooling. Yet, the goals of *lisb* and αRby ultimately differ: where αRby seems to be mostly motivated by mixed execution, usage of partial solutions and stages model finding, no data-oriented representation of the specification itself is available. While, technically, it is possible to access the Ruby AST of the code, again, special knowledge of library-generated AST nodes is required. The main design goal of *lisb* is separating the IR from the backend code, and, thus, allow easy transformation on this IR. Thus, we also differ in our judgement concerning the programming language a formalism should be embedded into: we deem a functional language more suitable[13] for data transformation than an imperative language, which is more suitable for interacting with a solver and its (partial) solutions.

B-specific meta-approaches For B and Event-B, approaches have been presented that build a model of the formalism in the formalism itself: The language workbench Meeduse [28] has been used to create proven DSLs. A possible approach is to specify the structure of a DSL and to use B expression in order to instantiate a new B machine. This has been demonstrated on the example of Petri-Nets [29]. In Event-B, the EB4EB meta-theory [47] contains an Event-B model of Event-B models. So far, the intended use seems to be extending the Event-B language so that new kinds of properties are integrated in the proof obligation mechanism. Nonetheless, one could also use this meta-theory in order to specify DSLs.

Construction via APIs The idea of programmatic construction of specifications is not new: solvers such as Z3 [42], Coq [7] and also ProB itself have APIs that allow building constraints. *lisb* attempts to hide low-level details (such as creation of suitable types) and provides a more abstract DSL to this end.

[13] A worthwhile project could be to specify transformations in the embedded formalism itself, in our case B.

Rosette The Rosette framework is written in Racket and employs similar macro-based techniques[14] in order to construct solver-aided domain-specific languages (dubbed SDSLs) [52]. Its main idea is that a developer only needs to program an interpreter for a (domain-specific) language that may even use advanced Racket features such as pattern matching, dynamic evaluation, higher-order functions or functions with side effects — which are typically hard to translate into logic constraints — if they can be eliminated using partial evaluation. Rosette also accepts libraries and APIs for this purpose. That interpreter and a program in the newly-defined language are both translated into SMT constraints. Under the hood, Rosette uses a symbolic virtual machine in order to provide testing, debugging, verification and program synthesis tools for the DSL.

Generic model checking framework A yet to be named model checking framework[15] by Rozier et al. follows a similar overall approach: at its core shall also be an intermediate language (IL) that can be used to define finite-state systems (that may be extended to infinite-state as well). The idea is that the IL will be powerful enough so that other formalisms can be translated into the IL. Then, the model representation can be fed into different model checking algorithms and solver backends, e.g., SMT and SAT solvers. This shares a vision with *lisb*: while the IR itself (currently) is very B-centric, the mathematical foundation of B is expressive enough that several formalisms already have been successfully translated into B, such as TLA$^+$ [22], Alloy [32] and Lustre [54]. Then, the rich variety of B-specific tools can be regarded as backends, such as animation and model checking via ProB, machine repair and synthesis tools (e.g., [12, 48]) and code generators (e.g., [55]).

8 Conclusions

In this paper, we have presented *lisb*, which embeds the B language into Clojure in order to meta-program specifications. While it may be less appealing for modelling experts, as they are confronted with another programming language, *lisb* certainly is a helpful library for rapid tool development.

By embracing the ideas of Lisp and treating specifications as pure data, existing specifications can easily be transformed and new ones can be generated from external data sources. Especially for large datasets, it can be significantly faster and more memory-efficient to avoid parsing a textual representation and to generate the AST programmatically instead. Moreover, Clojure's macro systems provides support for easy creation of DSLs.

[14] Macros in Racket are fundamentally the same as in Clojure.

[15] https://www.aere.iastate.edu/modelchecker/

Many new applications can quickly be implemented due to the combination of *(i)* an easy way to pick apart and recombine formal specifications, *(ii)* a language interpreter to give meaning to the specification snippets and *(iii)* a Turing-complete general-purpose programming language. We think that such mechanisms are needed so that tool developers can thrive; so that formal methods appeal to a more general audience (as users can overcome limitations of existing tools and come up with their own syntax, visualisations or testing tools, …) Yet, as transformation tools and DSLs can become arbitrarily complex, bugs might be introduced as well. Rules engines and term rewriting libraries can assist in capturing translation rules more formally; proof might be required to ensure that such translation rules are correct wrt. the programmer's expectations.

Overall, we conclude that formal methods tools will heavily benefit from such a data-oriented approach. As we assume that the majority of formal methods experts does not have a background in Clojure, facilities for generating (parts) of specifications, e.g., a proper macro system, could also be a useful part of formal languages. Transformations and DSL creation can maybe be limited to a structured subset of a macro language in order to enforce correctness.

Overall, embedding formalisms and their supporting tools into programming languages makes formal methods also more accessible for programmers. An interesting idea is that DSLs could generate parts of a model *and* of traditional code at the same time. Co-simulation tools [21] can then be used to validate that the program adheres to the specification.

8.1 Future work

lisb opens doors leading to many directions: first, many higher-level specification languages such as TLA$^+$ or Kodkod share a similar abstraction level. Large parts of their corresponding IR overlap with the mathematical language of B. One could incorporate the work of existing translations to transform constructs that are specific to a given formalism. *lisb* could then serve as a tool to translate specifications into *all* (supported) formalisms. This would also allow multi-paradigm modelling and decomposing parts of models that are more suitable for a specific tool. Here, DSLs and pattern matching libraries can help to reduce awkward or inefficient translations by providing constructs closer to a language's idioms.

Second, in contrast to the current focus on model checking, animation and embedding into applications, one could provide a DSL that generates constructs known to work well with provers. This can be useful since many models written to work with animators such as ProB often do not work well with proving tools, and vice versa. This requires further research in which language constructs work well with what tool (e.g., based on the work of Dunkelau et al. [16]).

Third, one goal of *lisb* is to provide more DSLs so that model extraction from existing software becomes feasible. As demonstrated, constructs such as if-statements and loops can easily be expressed, whereas function calls, classes and interfaces require more complex translations. Polymorphic and recursive functions are known to be particularly challenging to express in B [37]. FASTEN [45, 46] demonstrates the power of entire DSL stacks that can be composed, e.g., support for components with inputs and outputs, contract-based design, and allows unit testing of particular components for test-driven development. We are currently working on a translation from Solidity [14] to B, and plan to extract a subset of rules as a foundation of a DSL that mimics the control flow of traditional programming languages (in particular, stack-based function calls).

Fourth, we aim to strengthen our support for Event-B [3]. While we are already able to generate projects [5] compatible with the Rodin platform [4], there is more work to be done. One goal is to create an intermediate representation for Rodin's proof infrastructure (so that it becomes easier to double-check a proof with external provers; or generate a proof from a different tool) and theories [11] (as they include structured language extensions and thus are part of the specification). Additionally, we are working on a complete translation between B and Event-B: Though these two formalisms share their roots and are very similar, they have some differences (of which many are very subtle) [37]. Some operators are available in only one of the dialects; and the possibilities of machine refinement and inclusion in B are more complex compared to what Event-B offers. Switching between these dialects can assist modellers, as B is more suitable for prototyping due to its flexibility, but proof support in the Event-B eco-system is better [40].

Appendix A Addressing B-specific issues

In this section, we show how *lisb* can also be used to fix certain shortcomings of the B language en passant. The B language has a so-called definition system that is based on text replacement (similar to macros in the C language). One could argue that certain (local) transformations and DSLs can be implemented directly using definitions. Below, we examine drawbacks of this system and illustrate why *lisb* offers a cleaner solution.

A.1 Language semantics—definitions

The definition mechanism of the B language is similar to C preprocessor macros and has similar drawbacks: Actual operator precedences may be misleading and differ based on the tool. Further, it is also possible to capture variables on

```
DEFINITIONS
add(xx,yy) == xx+yy
egt(xx) == (∃ yy.(yy ∈ 1..99 ∧ xx < yy))
```

Listing 15 Two suspicious definitions

```
(defpred add [xx yy]
  (+ xx yy))
(defpred egt [x]
  (exists [:y] (and (in :y
        (interval 1 99)) (< x :y)))
            )
```

Listing 16 Two safe predicates

```
(defpred ifte [condition then else]
  (fn-call (union (lambda [:t]
        (and (member? t #{true})
        condition) then))
                  (lambda [:t]
        (and (member? t #{true})
        (not condition) else)))
           true))

(defpred abs [x] (ifte (> x 0)
        x (- x)))
```

Listing 17 Manual implementation of if-then-else and its usage

accident. Below, we present two examples which have been discussed by Leuschel [37] in detail.

Operator precedences One issue is that the definition mechanism is interpreted differently by different tools. Consider the first definition in Listing 15: When calling the definition in AtelierB as `2*add(0,5)`, the result will be 5 because it will be expanded to $2*0+5$. However, evaluating the expression with ProB will expand the definition to $2*(0+5)$ and 10 will be returned.

Variable capturing The second definition in Listing 15 shows the issue of variable capturing. Calling $egt(5)$ will yield true (since $yy = 6$ exists). However, $yy = 5 \wedge egt(yy)$ will result in the rewritten predicate $yy = 5 \wedge (\exists yy.(yy \in 1..99 \wedge yy < yy))$, which is false.[16]

lisb's alternative to definitions *lisb*'s solution to code reuse is the predicate abstraction (`pred`). It is a macro that internally replaces all variables (i.e., keywords) with new variable names. The code snippet in Listing 16 contains the safe equivalent expressions to the B definitions in Listing 15. First, the `add` predicate is unambiguous wrt. operator precedence as, highlighted by the parenthesis, the result is an addition that is directly inserted into the AST. Second, the `egt` predicate

[16] For such cases, ProB will generate a warning.

```
(def □ globally)
(def ◊ finally)
(def ∘ next)
(def U until)
(def ∧ ltl-and)
(def ∨ ltl-or)
(def ¬ negate)

(defn => [x y]
  (ltl-or (negate x) y)
```

Listing 18 Small DSL for shorthand notation

cannot capture the variable `y` because of the renaming of all prefixes of all local variables with `lisb_`. As an example, the B code $\exists \text{lisb}_{5355}.(\text{lisb}_{5355} \in 1..99 \wedge 5 < \text{lisb}_{5355})$ results from calling $egt(5)$ in *lisb*.

A.2 Introducing convenience operators

The B language only supports a branching if-then-else construct on the level of variable substitutions. However, it is missing if-then-else on the expression level, e.g., one cannot get the absolute value of an integer by writing `IF x > 0 THEN x ELSE -x END`. During initial development of *lisb*, ProB's dialect introduced support for such an expression, which required changes to its parser and its constraint solver core. We argue that one should be able to define an operator based on the (admittedly unwieldy) tool-agnostic re-writing rule below presented by Hansen [22].

$$(\lambda t.(t \in \{\text{TRUE}\} \wedge (x > 0) \mid x)$$
$$\cup\, \lambda t.(t \in \{\text{TRUE}\} \wedge \neg(x > 0) \mid -x))(\text{TRUE})$$

In *lisb*, one can introduce such a ternary operator easily by simply defining the re-writing rule. The entire implementation of an `ifte` expression and of an absolute value function, which require no further changes to ProB or its parser, is given in Listing 17.

Appendix B Full code listings

Below, we give the full source code for the examples in Sect. 6.

B.1 Dwyer pattern implementation

Listing 18 shows how a small DSL wrapper can be defined. Listing 19 makes use of this DSL wrapper in order to encode the formulas given by Dwyer et al. [17] Further, the func-

```
(defn dwyer-s-responds-p-globally
  "this generates the LTL formula
     □(P ⇒ ◊S)"
  [S P]
  (□ (=> P (◊ S))))
(defn dwyer-s-responds-p-before-r
  "this generates the LTL formula
  (P ⇒ (¬R U (S ∨ ¬R))) U (R ∨ □¬R)"
  [S P R]
  (U (=> P (U (¬ R) (∧ S (¬ R))))
     (∨ R (□ (¬ R)))))
(defn dwyer-s-responds-p-between-q-
  and-r
  "this generates the LTL formula
  □((Q ∧ o◊R) ⇒ (P ⇒ (¬R U (S ∧ ¬R))) U R)"
  [S P Q R]
  (□ (U (=> (∧ Q (o (◊ R)))
            (=> P (U (¬ R) (∧ S (¬ R)))
                ))) R)))
(defn dr [S P & [opt-kw opt-Q
    opt-kw2 opt-R]]
  (pp (cond (not opt-kw) (dwyer-s-
  responds-p-globally S P)
      (= opt-kw :between) (dwyer-s-
      responds-p-between-q-and-r S P
           opt-Q opt-R)
      (= opt-kw :before) (dwyer-s-
      responds-p-before-r S P opt-Q)
          )))
;; example calls
user=> (dr "x=1":responds-to "y=2")
;; S responds to P globally
"□((y=2) => (◊(x=1)))"
user=> (dr "x=1" :responds-to "y=2"
     :before "a=2")
;; S responds to P before R
"((y=2) => ((¬(a=2)) U ((x=1) ∨ (¬(
    a=2))))) U ((a=2) ∨  (□(¬(a=2)))
    )"
user=> (dr "x=1" :responds-to "y=2"
     :between "a=3" :and "b=42")
;; S responds to P
;; between Q and R
"□((((a=3) ∧ (o(◊(b=42)))) => ((y=2)
     => ((¬(b=42)) U ((x=1) ∧
(¬(b=42)))))) U (b=42))"
```

Listing 19 Definition of the response pattern and example calls

tion `dwyer-response` is a more user-oriented entry point which leads to more readable expressions.

B.2 Positive normal-form transformation

This subsection contains the required transformation rules and the code implementing them.

Transformation rules

In the following, the transformation rules to obtain a positive-normal form from Baier and Katoen [6, Def. 5.20] are given:

$$\begin{aligned}
\neg true &= false \\
\neg false &= true \\
\neg\neg\phi &= \phi \\
\neg(\phi \wedge \psi) &= \neg\phi \vee \neg\psi \\
\neg\phi &= \neg\phi \\
\neg(\psi U \phi) &= (\phi \wedge \neg\psi) W (\neg\phi \neg\psi)
\end{aligned}$$

Implementation

Listing 20 shows the code (`pni`) implementing the rules above. The `traverse` function recursively walks through the data structure and calls the `pni` function to apply the rules once a negation is encountered.

And, indeed, for the formula $\neg\Box((a\ U\ b) \vee \circ c)$ (which is an example used by Baier and Katoen [6, Ex. 5.21]), we obtain the equivalent LTL formula in the normal form, as shown in Listing 21.

```
(declare pni traverse)

(defmulti traverse
 (fn [x] (or (:tag x) (class x))))
(defmethod traverse Boolean [x] x)
(defmethod traverse String [x] x)
(defmethod traverse :next [x]
  (update :ltl traverse))
(defmethod traverse :until [x]
  (-> x (update :lhs traverse)
        (update :rhs traverse)))
(defmethod traverse :and [x]
  (-> x (update :lhs traverse)
        (update :rhs traverse)))
(defmethod traverse :or [x]
  (-> x (update :lhs traverse)
        (update :rhs traverse)))
(defmethod traverse :weak-until [x]
  (-> x (update :lhs traverse)
        (update :rhs traverse)))
(defmethod traverse :not [x]
  (pni (:ltl x)))

(defmulti pni
  (fn [x] (or (:tag x) (class x))))
(defmethod pni String [x]
  (negate x))
(defmethod pni Boolean [x]
  (not x))
(defmethod pni :not [x]
  (traverse (:ltl x)))
(defmethod pni :next [x]
  (next (traverse
        (negate (:ltl x)))))
(defmethod pni :until [x]
 (traverse (weak-until
  (ltl-and (:lhs x)
           (negate (:rhs x)))
  (ltl-and (negate (:lhs x))
           (negate (:rhs x))))))
(defmethod pni :and [x]
 (traverse (ltl-or
             (negate (:lhs x))
             (negate (:rhs x)))))
(defmethod pni :or [x]
 (traverse (ltl-and
           (negate (:lhs x))
           (negate (:rhs x)))))
(defmethod pni :weak-until [x]
(traverse
 (until (ltl-and (:lhs x)
          (negate (:rhs x)))
        (ltl-and (negate (:lhs x))
          (negate (:rhs x))))))
```

Listing 20 Entire source code of a tool that transforms LTL formulas into Positive Normal Form

```
user=> (pp (negate (globally
        (ltl-or (until "a" "b")
        (next "c")))))
"¬(¬((true) U (¬(((a) U (b))
        W (o(c))))))"
user=> (pp (traverse (negate
        (globally (ltl-or (until
        "a" "b") (next "c"))))))
"(true) U ((((a) ∧ (¬(b))) W
        ((¬(a)) ∧
        (¬(b)))) ∧ (o(¬(c))))"
```

Listing 21 REPL interaction in which we generate a formula in PNF

Acknowledgements The authors would like to thank Kristin Rutenkolk for her feedback, and David Geleßus for his quick fixes in the ProB toolchain. The first author also thanks David Schneider, Jens Bendisposto and Michael Leuschel for their fruitful suggestions and support. Henrik Hinzmann provided the code in Listing 7. The authors also thank the anonymous referees for their comments which helped significantly in improving the paper.

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability All code of *lisb* and implemented tools is available on GitHub:

Declarations

Conflict of interest The authors have no Conflict of interest to declare that are relevant to the content of this article.

lisb https://github.com/pkoerner/lisb

Algorithm DSL case study https://github.com/pkoerner/lisb/blob/master/src/lisb/adl/adl2lisb.clj

Machine transformation case study https://github.com/JanRossbach/fset

References

1. Abo R, Voisin L (2014) Formal implementation of data validation for railway safety-related systems with OVADO. In: Proceedings SEFM (international conference on software engineering and for-

mal methods) 2013, Lecture Notes in Computer Science, vol. 8368, Springer, pp 221–236
2. Abrial JR (1996) The B-Book: assigning programs to meanings. Cambridge University Press, Cambridge
3. Abrial JR (2010) Modeling in event-B: system and software engineering. Cambridge University Press, Cambridge
4. Abrial JR, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L (2010) Rodin: an open toolset for modelling and reasoning in event-B. Softw Tools Technol Transf 12(6):447–466
5. Armbrüster J, Körner P (2024) Meta-programming event-B—advancing tool support and language extensions. In: Proceedings ABZ (International conference on rigorous state-based methods), Lecture Notes in Computer Science, vol 14759. Springer, pp 233–240 https://doi.org/10.1007/978-3-031-63790-2_17
6. Baier C, Katoen JP (2008) Principles of model checking. MIT Press, Cambridge
7. Bertot Y, Castran P (2010) Interactive theorem proving and program development: Coq'Art the calculus of inductive constructions. Springer, Berlin
8. Bettini L (2016) Implementing domain-specific languages with Xtext and Xtend. Packt Publishing Ltd., Birmingham
9. Bucchiarone A, Cicchetti A, Ciccozzi F, Pierantonio A (2021) Domain-specific languages in practice: with JetBrains MPS. Springer. https://doi.org/10.1007/978-3-030-73758-0
10. Butler M, Körner P, Krings S, Lecomte T, Leuschel M, Mejia LF, Voisin L (2020) 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. Springer, pp 189–209
11. Butler M, Maamria I (2013) Practical theory extension in event-B. In: theories of programming and formal methods, Lecture Notes in Computer Science, vol 8051. Springer, pp 67–81
12. Cai CH, Sun J, Dobbie G, Hóu Z, Bride H, Dong JS, Lee SUJ (2022) Fast automated abstract machine repair using simultaneous modifications and refactoring. Form Asp Comput 34:1–31
13. Clark J, Bendisposto J, Hallerstede S, Hansen D, Leuschel M (2016) 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. Springer, pp 183–197
14. Dannen C (2017) Introducing Ethereum and solidity, vol 1. Springer, Berlin
15. Dobrikov I, Leuschel M (2016) Optimising the ProB model checker for B using partial order reduction. Form Asp Comput 28(2):295–323
16. Dunkelau J, Schmidt J, Leuschel M (2020) Analysing ProB's Constraint Solving Backends: What Do They Know? Do They Know Things? Let's Find Out! In: Proceedings ABZ (International Conference on Rigorous State-Based Methods), Lecture Notes in Computer Science, vol 12071. Springer, pp 107–123
17. Dwyer MB, Avrunin GS, Corbett JC (1998) Property specification patterns for finite-state verification. In: Proceedings FMSP (workshop on formal methods in software practice), ACM, pp 7–15
18. France R, Rumpe B (2007) Model-driven development of complex software: a research roadmap. In: Proceedings FOSE (future of software engineering), IEEE. pp 37–54 https://doi.org/10.1109/FOSE.2007.14
19. Gagnon EM, Hendren LJ (1998) SableCC: an object-oriented compiler framework. IEEE
20. Giannakopoulou D, Mavridou A, Rhein J, Pressburger T, Schumann J, Shi N (2020) Formal Requirements Elicitation with FRET. In: Proceedings REFSQ (international working conference on requirements engineering: foundation for software quality), ARC-E-DAA-TN77785
21. Gomes C, Thule C, Broman D, Larsen PG, Vangheluwe H (2018) Co-simulation: a survey. ACM Comput Surv 51(3):1–33
22. Hansen D, Leuschel M (2012) Translating TLA+ to B for validation with ProB. In: Proceedings IFM (International conference on integrated formal methods), Lecture Notes in Computer Science, vol 7321. Springer, pp 24–38
23. Hansen D, Leuschel M, Körner P, Krings S, Naulin T, Nayeri N, Schneider D, Skowron F (2020) Validation and real-life demonstration of ETCS hybrid level 3 principles using a formal B model. Softw Tools Technol Transf 22:315
24. Hansen D, Leuschel M, Schneider D, Krings S, Körner P, Naulin T, Nayeri N, Skowron F (2018) Using a Formal B Model at Runtime in a Demonstration of the ETCS Hybrid Level 3 Concept with Real Trains. In: Proceedings ABZ (International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z), Lecture Notes in Computer Science, vol 10817. Springer, pp 292–306
25. Hansen D, Schneider D, Leuschel M (2016) Using B and ProB for data validation projects. In: Proceedings ABZ (International conference on abstract state machines, Alloy, B, TLA, VDM, and Z), Lecture Notes in Computer Science, vol 9675. Springer, pp 167–182
26. Hickey R (2020) A History of Clojure. In: Proceedings HOPL (History of Programming Languages), ACM, pp 1–46
27. Humm BG, Engelschall RS (2010) Language-Oriented Programming Via DSL Stacking. In: Proceedings ICSOFT (International conference on software and data technologies), pp 279–287
28. Idani A (2020) 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. Springer, pp 349–367
29. Idani A (2024) Transpilation of petri-nets into b: Shallow and deep embeddings. In: Proceedings ABZ (International Conference on Rigorous State-Based Methods), Lecture Notes in Computer Science, vol 14759. Springer, pp 80–98
30. Körner P, Leuschel M (2023) Towards practical partial order reduction for high-level formalisms. In: Proceedings VSTTE (international conference on verified software: theories, tools, and experiments) 2022, Lecture Notes in Computer Science, vol 13800. Springer
31. Körner P, Mager F (2022) An embedding of B in Clojure. In: Companion proceedings MODELS (international conference on model driven engineering languages and systems: companion proceedings), ACM, pp 598-606
32. Krings S, Leuschel M, Schmidt J, Schneider D, Frappier M (2020) Translating alloy and extensions to classical B. Sci Comput Program 188:1–25
33. Körner P, Bendisposto J, Dunkelau J, Krings S, Leuschel M (2020) Integrating formal specifications into applications: the ProB Java API. Form Methods Syst Des 57:160–187
34. Lamport L (2009) The PlusCal algorithm language. In: Proceedings ICTAC (international colloquium on theoretical aspects of computing), Lecture Notes in Computer Science, vol 5684. Springer, pp 36–60
35. Lecomte T (2014) Atelier B, chap. 2, Wiley, pp 35–46
36. Lecomte T, Burdy L, Leuschel M (2012) Formally checking large data sets in the railways. CoRR **abs/1210.6815**. Proceedings of DS-Event-B
37. Leuschel M (2021) Spot the difference: a detailed comparison between B and Event-B. In: Logic, computation and rigorous methods, Lecture Notes in Computer Science, vol 12750. Springer, pp 147–172
38. Leuschel M, Bendisposto J, Hansen D (2014) Unlocking the mysteries of a formal model of an interlocking system. In: Proceedings Rodin Workshop 2014
39. Leuschel M, Butler M (2008) ProB: an automated analysis toolset for the B method. Softw Tools Technol Transf 10(2):185–203
40. Leuschel M, Mutz M, Werth M (2020) Modelling and validating an automotive system in classical B and Event-B. In: Proceedings

ABZ (International Conference on Rigorous State-Based Methods), Lecture Notes in Computer Science, vol 12071. Springer, pp 335–350
41. Milicevic Aleksandar Erfrati I, Jackson D (2014) αRby—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. Springer, pp 56–71
42. de Moura LM, Bjørner N (2008) Z3: an efficient SMT solver. In: Proceedings TACAS (International conference on tools and algorithms for the construction and analysis of systems), Lecture Notes in Computer Science, vol 4963. Springer, pp 337–340
43. Ozkaya M, Erata F (2020) Understanding practitioners' challenges on software modeling: a survey. J Comput Lang 58:100963. https://doi.org/10.1016/j.cola.2020.100963
44. Peled D (1994) Combining partial order reductions with on-the-fly model-checking. In: Proceedings CAV (International Conference on Computer Aided Verification), Lecture Notes in Computer Science, vol. 818. Springer, pp 377–390
45. Ratiu D, Gario M, Schoenhaar H (2019) FASTEN: an open extensible framework to experiment with formal specification approaches. In: Proceedings formalise (workshop on formal methods in software engineering), IEEE, pp 41–50
46. Ratiu D, Nordmann A, Munk P, Carlan C, Voelter M (2021) FASTEN: an extensible platform to experiment with rigorous modeling of safety-critical systems. In: Domain-specific languages in practice, Springer, pp 131–164
47. Rivière P, Singh NK, Aït-Ameur Y (2022) EB4EB: a framework for reflexive Event-B. In: Proceedings ICECCS (International Conference on Engineering of Complex Computer Systems), IEEE, pp 71–80 https://doi.org/10.1109/ICECCS54210.2022.00017
48. Schmidt J, Krings S, Leuschel M (2018) Repair and generation of formal models using synthesis. In: Proceedings iFM (international conference on integrated formal methods), Lecture Notes in Computer Science, vol 11023. Springer, pp 346–366
49. Schneider D (2017) Constraint modelling and data validation using formal specification languages. Ph.D. thesis, Universitäts- und Landesbibliothek der Heinrich-Heine-Universität Düsseldorf
50. Schneider D, Leuschel M, Witt T (2015) Model-based problem solving for university timetable validation and improvement. In: Proceedings FM (international symposium on formal methods), Lecture Notes in Computer Science, vol 9109. Springer, pp 487–495
51. Schneider D, Leuschel M, Witt T (2018) Model-based problem solving for university timetable validation and improvement. Form Asp Comput 30:545–569
52. Torlak E, Bodik R (2013) Growing Solver-Aided Languages With Rosette. In: Proceedings Onward! (International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software), ACM, pp 135–152
53. Torlak E, Jackson D (2007) Kodkod: A relational model finder. In: Proceedings TACAS (international conference on tools and algorithms for the construction and analysis of systems), Lecture Notes in Computer Science, vol 4424. Springer, pp 632–647
54. Vu F (2020) Simulation and verification of reactive systems in Lustre with ProB. Master's thesis, Heinrich Heine Universität Düsseldorf
55. Vu F, Hansen D, Körner P, Leuschel M (2019) A multi-target code generator for high-level B. In: Proceedings iFM (International Conference on integrated Formal Methods), *Lecture Notes in Computer Science*, vol 11918. Springer, pp 456–473