Heinrich Heine
Universität
Düsseldorf

INSTITUT FÜR INFORMATIK
Softwaretechnik und Programmiersprachen
Universitätsstr. 1 D–40225 Düsseldorf

An Event-B Backend for lisb

Bachelorarbeit

im Studiengang Informatik
zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

vorgelegt von
Peter Jakob Julius Armbrüster

Beginn der Arbeit: 14. April 2023
Abgabe der Arbeit: 14. Juli 2023

Erstgutachter: Prof. Dr. Michael Leuschel
Zweitgutachter: Dr. Jens Bendisposto

Selbstständigkeitserklärung

Hiermit versichere ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, den 14. Juli 2023

Peter Jakob Julius Armbrüster

Abstract

In the context of system specifications, the Rodin platform has proven to be a valuable tool for complex proofs. However, constructing formal specifications within Rodin is primarily a manual endeavor. This thesis addresses this limitation by exploring the programmatically-driven transformation of the Event-B specification language. To accomplish this, we extend lisb, an existing embedding of the B language in Clojure, to include support for Event-B.

Clojure, a modern Lisp dialect designed for the Java Virtual Machine (JVM), presents an ideal choice for this task. Its underlying philosophy treats programs as data, enabling powerful metaprogramming capabilities. Notably, Clojure's macro system empowers the implementation of domain-specific languages (DSLs) with relative ease.

Building on lisb's foundation and the ProB Java API, this thesis seeks to automate and streamline the transformation of Event-B specifications. Clojure's data-centric approach is compelling for manipulating and generating Event-B specifications programmatically. We aim to simplify and enhance the process of developing tools for Event-B, promoting the adoption of formal methods in practical applications.

Contents

1 Introduction

Event-B [Abr10] is a formal language and method for designing and verifying systems; it greatly simplifies many features of its predecessor, Classical B. On the one hand, it is advantageous because it simplifies proofs and makes the language more scalable, allowing large systems to be modeled and verified using Event-B. But on the other hand, it can be cumbersome to write large models, as it leads to code repetition. For example, Event-B has no substitution if-then-else, so the behavior has to be modeled with two different events, one with the condition as a guard and the other with the negated condition as in the guard. But this often leads to code duplication, especially when the events have many substitutions in common.

Experimenting with new features Event-B often involves manipulating the toolchain. Rodin, for example, offers a plugin system for which already many plugins were already developed. For instance, Camille [Ben+11] and CamilleX [Hoa+21] both try to resolve the lack of a textual representation in Rodin. A plugin for ProB [LB08] allows for model checking directly in Rodin. Still, creating a plugin for Rodin is often extreme and requires much effort. An embedding of Event-B into a programming language like Clojure can simplify the process of extending Event-B, especially in the prototyping phase.

The thesis aims to create an embedding of Event-B in Clojure by extending *lisb* [KM22], an embedding of Classical B in Clojure. The ProB Java API [Kör+20] will be used as the backbone, integrating it with the ProB animator and model-checker. We also hope to join the toolchain of Classical B and Event-B. Often, there are different ways to translate between Classical B and Event-B, depending on the intention of the user. *lisb* can be used to analyze and transform the model programmatically. For example, to create transformations between Classical B and Event-B in particular situations, as demonstrated manually in [LMW20].

The target audience for this thesis is tool developers for Event-B, modeling experts, and interested parties with expertise in formal methods and safety-critical systems.

Section 2 will begin with a summary of the technologies used in this thesis. We are starting with an outline of Clojure. After that, we describe the current state of *lisb*. Then, we give a brief comparison of classical B and Event-B. The Section is closed with a review of the different Classical B and Event-B tools.

In Section 3, we outline the components of our implementation. First, we describe the ProB Event-B Model as the backbone of our embedding. Then we present the extensions made to the internal representation to support Event-B. Next, we define a DSL for writing Event-B in a Clojure-Style. After that, we explain how we generated a ProB Event-B Model from the internal representation.

In Section 4, we use *lisb* to create a translation from Classical B to Event-B, beginning with operations. Following with a translation of substitutions. Then we describe two ways of

converting inclusion into Event-B. The Section is wrapped up with a comparison to manual translation from classical B to Event-B.

We complete the thesis with some related work, summarizing the most important results and referring to this thesis's goals. Last but not least, the thesis is finalized with prospects to *lisb*.

2 Background

First, we illustrate the necessary background information for this thesis to create a uniform understanding of Clojure, *lisb*, and Event-B. We will begin with Clojure because it is the host language of *lisb* and is not very popular amongst programmers. Then, we examine the current state of *lisb* as an embedding of Classical B. We briefly discuss the main differences between Classical B and Event-B. The following Section contains a summary of three popular tools for Classical B and Event-B.

2.1 Clojure

Clojure [Hic08; Cloj] is a dynamic, functional programming language built for the JVM. It combines the power and flexibility of Lisp programming with the robustness and interoperability of Java. All Clojure data structures are immutable by default, simplifying transformation and interaction with data. Most Clojure functions are pure, meaning they do not modify any state of the system and only transform their input. Pure functions are easy to reason about and to debug. An additional strength of Clojure is its rich macro system, as it allows the language to be extended with new syntax. We will briefly introduce critical concepts since Clojure is not a widely adopted programming language.

Clojure is a Lisp dialect, which means that the primary data structure is a list. Even function calls are lists, then the first element of the list is the function and the rest are the arguments. For example, the call `(f a b)` is equivalent to `f(a, b)` in Java. Every function in Clojure is called this way in, also arithmetic operations. For instance, `(+ 1 2 3)` is the same as `1 + 2 + 3` in infix notation. Newcomers to Clojure or other Lisps often find the number of parentheses overwhelming. A benefit of the parenthesis is that they remove the need for operator precedence, which is helpful for our DSL. Besides lists, Clojure also supports other data structures, such as vectors, maps, and sets, which are all immutable.

A REPL allows for interactive programming and rapid prototyping in Clojure, especially when experimenting and interacting with *lisb*.

In object-oriented languages like Java, Polymorphism is often achieved with interfaces or inheritance, where subtypes overwrite methods of their supertype. Clojure uses a dynamic and very flexible approach for polymorph functions called multimethods. Multimethods

allow for a dynamic dispatch on all parameters via a provided function, making it possible to dispatch on complex structures and not only on the class of an object, for example. In *lisb*, multimethods are used to implement an extensible for the implementation translations between different representations.

2.2 lisb

lisb [KM22] is an embedding for B in Clojure, with the goal of facilitating the work of tool developers treating the specification as data. It serves as an intermediate layer between a user program and the ProB Java API [Kör+20] and allows for programmatic construction and transformation of the B constraints and machines.

At the core of *lisb* is the internal representation (IR). It is a data-oriented representation of the Classical B model. Scalar values such as boolean, numbers, and sets are denoted using the appropriate Clojure data literals. Mathematical operators and B-specific machine nodes, on the other hand, are represented by maps. Each of these maps includes a `:tag` key for identification, as well as extra keys for their operands (see [KM22, Section 3.2]).

On top of the IR, *lisb* provides an internal DSL designed for humans to write Classical B machines with a syntax similar to Clojure. The internal DSL creates an abstraction of the internal representation. Pure functions that generate IR are the basis of the internal DSL. The complete set of operators and machine clauses of Classical B is available in the internal DSL. Furthermore, internal DSL and IR can be mixed. So a user program may generate internal DSL at one point and IR at another, combining them to complete the model.

The internal representation is then translated into a ProB AST for Classical B. This AST can be used in the ProB Java API, allowing *lisb* to interact with the ProB to run constraint solving and model checking on the model. A retranslation from ProB to *lisb* is also available, translating a ProB AST into the internal DSL. The retranslation layer also converts the results from model checking into Clojure data structures. Via ProB, *lisb* can also import and export text-based models of Classical B.

2.3 Event-B

In this Section, we look at Event-B [Abr10], a formal method for systems modeling. It is an evolution of the B-method, also known as Classical B. Event-B shares many similarities with Classical B, e.g. both are state-based formalisms and rooted in predicate logic, set theory, and arithmetic. We will briefly compare the key differences between Classical B and Event-B, to understand what parts of *lisb* need extension.

In Event-B, a model is split into static parts and dynamic parts, as opposed to Classical B. The concept of *contexts* was introduced by Event-B. Contexts contain the static information of an Event-B model, such as *sets* (user-defined types), *constants*, *axioms*, and *theorems*.

Event-B *machines* include only the dynamic parts of a system, comprising *variables*, *invariants*, and *events*. The variables define the states of the machine. The invariants determine the types of variables and the rules that the variables must respect. A machine can see contexts, allowing access to its information. Contexts often model characteristic information about an environment. Ideally, this separation allows a machine to be reused in multiple situations. Listing 1 shows an example of a machine in Event-B called `m0`. The machine sees a context called `c0`, not shown here, which contains a set called `PHONES` and a constant `m`.

Listing 1: Event-B machine example

machine m0 **sees** c0
  variables phones
  invariants
    inv0: phones $\in$ PHONES
    inv1: card(phones) $\leq m$
  events
  event INITIALISATION **then**
    init0 : phones $:= \varnothing$
  end
  event OpenPhones **when**
    grd0: card(phones) $\neq m$
    grd1: phones $\neq$ PHONES
    then
    act0: phones : $|$ phones $\subset$ phones' $\wedge$ card(phones$'$) $\leq m$
  end
  event ClosePhones **when**
    grd0: phones $\neq \varnothing$
    grd1: phones $\neq$ PHONES
    then
    act0: phones : $|$ phones' $\subset$ phones
  end

Another simplification regards the inclusion mechanism. Classical B has a relatively complex inclusion system, where a machine can include multiple other machines and many keywords with different behavior: `INCLUDES`, `EXTENDS`, `USES`, `SEES`, and `IMPORTS`. In contrast, Event-B follows a much simpler approach than Classical B. In Event-B, the only allowed mechanisms to structure a model are:

- A context can *extend* multiple contexts.
- A machine can *refine* one machine and *see* multiple contexts.

The events are Event-B's replacement of operations in Classical B and represent the behavior of a machine. Events are similar to operations but simplify the notion significantly. Formally, an event has the following form:

$$\texttt{event}\ e\ \texttt{any}\ t\ \texttt{when}\ G(v,t)\ \texttt{then}\ S(v,t,v')$$

The symbols v and v' represent the variables before and after the execution of the event e. The parameters of the event are denoted by the symbol t. The guard $G(v,t)$ restricts the parameters and disables the event when evaluating to false. All actions for the event are contained in $S(v,t,v')$. There are three types of actions in Event-B:

- $x := E$, a deterministic action that assigns the value of the expression E to the variable x,
- $x :\in S$, a non-deterministic action, which assigns non-deterministically a value of the set S to x, and
- $x_1, x_2, ..., x_n : |BA$, the most general form of an action in Event-B. The execution of the action assigns $x_1, x_2, ..., x_n$ non-deterministically so that the values make the before–after predicate BA true.

When the event is executed, it runs all actions contained simultaneously. All unassigned variables remain unchanged. The guard of an event can be empty, which is equivalent to the guard being truth ($\top$). An empty action clause is also known as skip.

The concept of refinement differs significantly between events and operations. Classical B is often used for software modeling. Therefore operations can be called directly and have return values. An abstract machine serves as a specification for the available operations, and refinement serves the purpose of generating an implementation corresponding to that specification. Hence, the operations in the abstract machine and the refinement have a one-to-one relationship. The signature of an operation can not change over the curse of refinement, as it would be incompatible with the abstract specification.

In Event-B, events serve a different purpose. They model the abstract behavior of a system. In this case, it makes sense to add new events, split into variations, or merge events. Hence, we have to specify the abstract event that is refined, as there is no one-to-one relationship between the events of an abstract machine and a refinement. Additionally, the signature of an event can change between refinements. For instance, parameters can be removed, changed, or new ones introduced. When changing the signature of an event, a witness predicate has to be given. This predicate provides a relation between the old and the new parameters.

The most general form of an event e' is:

$$\texttt{event}\ e'\ \texttt{refines}\ e\ \texttt{any}\ t'\ \texttt{with}\ W(t,t',v,v')\ \texttt{where}\ G'(v,t')\ \texttt{then}\ S'(v,t,v')$$

Where e is the name of an event that is refined by e'. The witness $W(t,t',v,v')$ gives a connection between the old parameters t, new parameters t', and non-deterministic variables. For the event e' to properly refine the event e, it has to be proven that from the old guard follows the new guard.

This proof obligation is often referred to as guard strengthening. This behavior differs from Classical B, where operations often have a precondition, which must be true for the operation to have the desired behavior; otherwise, the output is not defined. The precondition must be weakened to ensure that a refined operation is still callable. So the implication is the other way around. From the new precondition follows the old precondition.

2.4 Tool support for the B-Method

In this Section, we look at the landscape of tools for the B-Method. Many tools support Classical B and Event-B, but the features always differ. We concentrate on the three most popular: Rodin, AtelierB, and ProB.

Rodin [Abr10] is an open-source platform based on Eclipse. It offers a range of tools and features to support the entire development lifecycle of Event-B specifications. Rodin provides a graphical editor for creating Event-B models, allowing developers to define machines and contexts. It includes a powerful theorem prover that can automatically discharge proof obligations and detect inconsistencies in the model. The Rodin platform offers an extensible architecture via Eclipse's plugin mechanism. The Event-B models are stored in XML files, which makes it difficult for humans to read. However, there are some plugins that resolve this issue, namely Camille [Ben+11] and CamilleX [Hoa+21].

AtelierB [Atel] is an IDE for Classical B developed and maintained by the company CLEARSY. Atelier-B provides project management, static checking, proof obligation generation, automatic and interactive proof, and code generation. It is widely used in the industry. Originally AtelierB was designed only for Classical B, but today also has support for a dialect of Event-B. The dialect [Bar20] of Event-B used by AtelierB has some differences compared to Rodin (see [Leu21]). For example, it allows more complex substitutions than Rodin.

ProB [LB08] is a model checker and animator with a kernel written in Prolog. ProB supports different state-based formalisms, including Classical B and Event-B. It can be used via the CLI or in a dedicated GUI with many additional features, i.e. visualization. There is also a plugin for Rodin, so both animation and model checking of an Event-B model are possible inside Rodin. This allows a user to explore the problem space before proving the model.

Another way of using ProB is via its Java API [Kör+20]. It allows for a complete interaction with ProB and gives a layer of abstraction to the lower-level API, which directly communicates with ProB's kernel. All features available in the GUI, like animation, constraint solving and model checking are also available in the ProB Java API. A large part of the ProB Java API is the model abstraction that allows different formalisms to be represented in ProB. Models in different formats can also be imported and exported using the ProB Java API. There are already different implementations of the model abstraction for Classical B, Event-B, TLA+, and CSP-M.

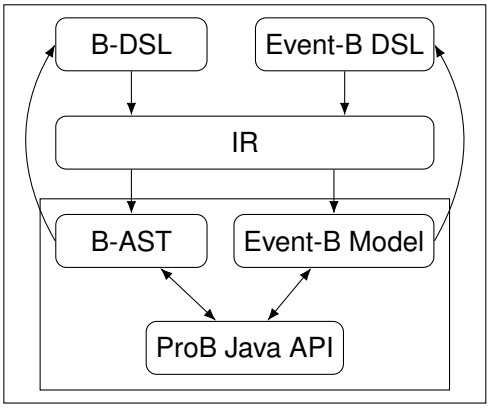

Figure 1: Architecture of *lisb* with our extension of Event-B

3 Components

We use a similar structure to the implementation for Classical B in *lisb*. In Figure 1, the extended architecture of *lisb* is shown. The main components of our implementation for Event-B in *lisb* are:

1. The Backbone of our implementation is the ProB Event-B Model.
2. The Internal representation is extended to capture the differences between Event-B and Classical B, pointed out in Section 2.3.
3. We added a Domain Specific Language for writing Event-B by extending the existing internal DSL
4. The Retranslation is adapted to handle the ProB Event-B Model

3.1 ProB Event-B Model

We use ProB's Model Abstraction for Event-B as our translation target. From now on, we will refer to Model Abstraction for Event-B as the *ProB Model*. In the ProB Model, the smallest building blocks are predicates and expressions. They are represented as strings, which are internally parsed to an AST. The syntax of these predicates and expressions is the same as in Rodin.

More complex semantics of machines and contexts are represented as a tree of Java objects, where the root node is an `EventBModel`. An `EventBModel` consists of `EventBMachine`, `EventBContext` and a `DependencyGraph`. The `DependencyGraph` describes the relationship between the machines and contexts. Possible relationships are: `SEES`, `REFINES`, and `EXTENDS`. However, an `EventBMachine` also contains direct references to the abstract machine and extended contexts. Similarly, the objects `Event` and `Context` have direct references to other `Event` or `Context` objects. All in all, the object graph of an Event-B ProB Model can

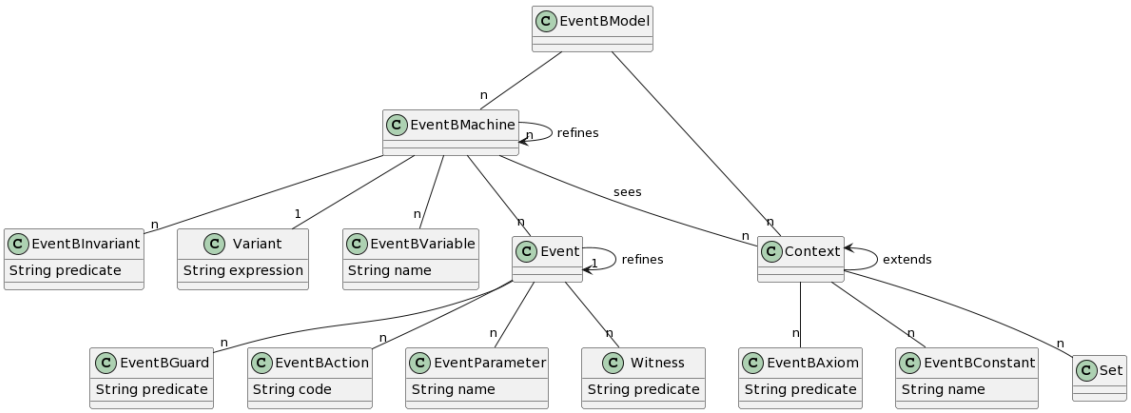

Figure 2: ProB Event-B Model structure

get very complex. Figure Figure 2 shows the structure of an `EventBModel`. Besides the dependencies, most nodes and their relationship shown in Figure 2 are self-explanatory, as they directly capture the semantics of Event-B.

3.2 IR for Event-B

This section describes how we represent an Event-B model in the internal representation (IR) of *lisb*. We will discuss what parts of the IR we could reuse and where we had to extend the IR. By reusing the existing IR, we are essentially translating Event-B into Classical B, so we try to reuse as much IR as possible. Furthermore, we used the detailed comparison of Classical B and Event-B by Leuschel [Leu21] as a guideline.

Predicates and expressions are the basis of Event-B. Event-B differentiates predicates from expressions at a syntactic level. In *lisb*, however, we blur the line a little, and expressions and predicates are more or less the same.

The mathematical sub-language in the IR is not specific to Classical B and thus can mostly be reused by Event-B. *lisb*'s IR also resolves some syntax differences between Classical B and Event-B. For example, in Classical B, the minus `-` and multiplication `*` operators are overloaded, with set difference and cartesian-product, respectively. Event-B resolves these ambiguities by using different operators for these operations. *lisb*'s IR is more explicit and has distinct operators for all operations. However, an operator called `cart-or-mult` still exists, which is generated when loading a Classical B model in ASCII form to *lisb*. But, direct use of `cart-or-mult` in *lisb* is not advised. In Table 1, the difference between operators in Classical B and Event-B is summarized.

For the new operators `finite` and `partition` in Event-B, we added IR nodes even though technically they could be represented in Classical B. Because, in the generation phase (see Section 3.4), it would be difficult to recognize the patterns.

Expression	Classical B	Event-B	IR Tag
integer difference	$A - B$	$A - B$	`:sub`
set difference	$A - B$, $A \backslash B$	$A \backslash B$	`:set-`
integer multiplication	$A * B$	$A * B$	`:mult`
cartesian-product	$A * B$	$A ** B$	`:cartesian-product`

Table 1: Comparison of operators between Classical B, Event-B and *lisb*

Contexts were introduced with Event-B; they only hold static information of a model. Representing an Event-B context in IR is straightforward because every clause in a context has a corresponding clause in Classical B. We introduced a new tag `:context`. Carrier sets and constants are the same for Classical B and Event-B. Therefore, we can reuse the existing `:sets` and `:constants` clauses. Axioms and theorems are stored in the `:properties` and `:assertions` clauses, respectively. We reuse the `:extends` clause to represent the relationship between different contexts.

Machines in Event-B model only dynamic behavior. In the IR, they are represented by the `:machine` tag, a name, and machine clauses. If the machine is a refinement, the `:refinement` clause is applied, which additionally contains the name of the refined machine. So the representation of an Event-B machine is very similar to that of a Classical B machine. Existing machine clauses allowed in an Event-B machine are:

- `:sees` contains references to contexts that are included in the machine,
- `:variables` contains a list of variables represented as Clojure keywords,
- `:invariants` contains a list of predicate representing the invariants,
- `:assertions` contains a list of predicate representing the theorems.

We include two new machine clauses: First, the `:variant` contains an expression representing the variant of the machine. Second, we create an `:events` clause instead of reusing the existing `:operations` clause for events.

Events are one of the main differences between Classical B and Event-B. Instead of trying to reuse the existing IR for operations and adapting it to events, we decided to create a new IR node with the `:event` tag. The map representing an event contains a `:name` and a list of `:clauses`. The new event clauses are:

- `:status` containing the convergence type of the event, if it is `:ordinary`, `:convergent` or `:anticipated`.
- `:event-reference` holds a reference to an event of the refined machine. Its type can be `:refines` or `:extends`.
- `:args` consists of the arguments to the event.
- `:guards` contains a list of guards for the event.

- `:witnesses` comprises a list of witnesses. A witness consists of a name and a predicate.
- `:actions` includes the actions of the event.

Actions in Event-B are significantly simpler than in Classical B. There are only three allowed substitutions in an event, all of which already have corresponding IR nodes:

- `:assignment` for the deterministic assignment,
- `:becomes-element-of` for non-deterministic assignment to an element of a set,
- `:becomes-such` for the non-deterministic assignment using a before-after predicate.

The IR is quite verbose, making it great for processing the model, but writing it by hand can be cumbersome. As an example, the IR for the machine in Listing 1 is given in Appendix A.

3.3 Event-B DSL

We developed an Event-B DSL similar to the internal DSL for classical B. Since the IR is not intended for humans to write directly, a DSL gives users of *lisb* more syntactic sugar. The Event-B DSL allows users to write complete models in Clojure. But one can also mix IR and DSL because we implemented the Event-B DSL as pure functions that generate the appropriate IR. This flexibility allows simple extensions to *lisb* without much boilerplate.

For predicates and expressions, we reused the existing functions from the Classical B DSL, where the function has a `b` prefixed to the operator name, to prevent name collisions with Clojure symbols. For Event-B-specific behavior, we added functions; these functions have an `eventb` prefixed. Most of the DSL functions directly correspond to the newly added IR nodes we described in the previous Section. For instance, the function `eventb-context` takes a name and any number of clauses.

The DSL function names for the event clauses do not correspond to the IR tag. They are more similar to clauses in Rodin:

- `eventb-status` yields the IR for `:status`
- `eventb-refines` generates an `:event-reference` with type `:refines`
- `eventb-extends` generates an `:event-reference` with type `:refines`
- `eventb-any` returns the IR for `:args`
- `eventb-when` returns the IR for `:guards`
- `eventb-with` takes an even number of arguments, where a name followed by the corresponding predicate is expected. Then the functions generates the individual witness maps and wraps them in a `:witnesses` clause.
- `eventb-then` returns the IR for `:actions`

We also added a macro called `eventb` to simplify the DSL, as there are now different prefixes, which can be confusing. So inside the macro, one can omit both the `b` and `eventb` prefix. Listing 2 shows the same machine as in Listing 1, but in the Event-B DSL.

```
(machine :m0
  (sees :c0)
  (variables :phones)
  (invariants
   (subset? :phones :PHONES)
   (<= (card :phones) :m))
  (init
   (assign :phones #{}))
  (events
    (event :OpenPhones
      (when
        (not= (card :phones) :m)
        (not= :phones :PHONES))
      (then
        (becomes-such [:phones] (and
          (subset? :phones :phones')
          (<= (card :phones') :m)))))
    (event :ClosePhones
      (when
        (not= :phones #{})
        (not= :phones :PHONES))
      (then
        (becomes-such [:phones]
          (subset? :phones' :phones))))))))
```

Listing 2: Machine in Event-B DSL

Theoretically, it is possible to use all functions that are present in the DSL for Classical B. However, some IR nodes generated by these functions are not valid Event-B seen in Section 3.2 and therefore not exportable to ProB or Rodin. In Section 4, we show how we used *lisb* to translate some features of Classical B to Event-B. For instance, allowing one to use the if substitution in the DSL and then translate it to valid Event-B. Other DSLs and translations could be added to *lisb* in the future.

3.4 Generating a ProB Model

The ProB Java API takes expressions and predicates as strings and then internally parses them, so we had to create a pretty printer for Event-B predicates, and expressions. This pretty printer is implemented as two multimethods for predicates and expressions called

```
user=> (require [lisb.translation.eventb.util]
    :refer [eventb ir->prob-model prob-model->rodin])
user=> (def c1 (eventb (context :c1 ...)))
user=> (def m1 (eventb (machine :m1 (sees :c1) ...)))
user=> (def model (ir->prob-model m1 c1))
user=> (prob-model->rodin model "model-name" "/path/to/model")
```

Listing 3: Generating ProB Model from IR and saving it as Rodin project

`ir-pred->str` and `ir-expr->str`. The strings are constructed recursively.

The nodes described in Section 3.2 are used for more specific Event-B semantics. Depending on the `:tag`, the multimethod `ir->prob` generates all these nodes except an `EventBModel`. When constructing nodes for axioms, invariants, and actions, we generate labels automatically, as we don't have support for user-defined labels.

To combine multiple machines and contexts into an `EventBModel`, we created the function `prob-model`. It takes any number of machines and contexts as ProB Model nodes and then generates a ProB Model with a correct `DependencyGraph`. For convenience, we combined the functions `ir->prob` and `prob->model` into the function `ir->prob-model`, which directly accepts IR.

In the ProB Model, refinements have direct references to their abstract version. In the IR, however, we only save the names of referenced machines and contexts. Therefore, we decided to create an empty machine or context every time we found a reference in the IR. For exporting to Rodin, this is fine, as it also uses names referencing other machines. When model checking, the correct machines or contexts have to be added manually.

In Listing 3, we see how one may create some machines and contexts using the DSL and generate a ProB Model with them. The model can then be exported as a Rodin project.

3.5 Retranslation from ProB Model

In this Section, we show how we implemented a retranslation for Event-B similar to that for Classical B. In *lisb*, a ProB AST for Classical B can be loaded and converted into the internal DSL; this step is called retranslation and allows for a complete cycle. In the original paper [KM22], they claim that passing a Classical B AST through *lisb* yields the same AST and does not change the model

Instead of an AST, we generate the Event-B DSL from a ProB Event-B Model, which we have discussed in Section 3.1. Compared to an AST, an `EventBModel` is structured differently, capturing the semantics of Event-B. So we could not reuse the existing `ast->lisb` function for `EventBModel`. Similar to the previous Section, we created a multimethod called `eventb-prob->lisb`, which recursively walks the model. We implemented a dispatch for

```
user=> (require [lisb.translation.eventb.util]
    :refer [rodin->lisb])
user=> (rodin->lisb "/path/to/machine_or_context")
[(context :c0 ...)
 (machine :m0 (sees :c0) ...)
 ...]
```

Listing 4: Loading a Rodin project and generating a Event-B DSL

all the nodes shown in Figure 2. Therefore, one can start at any node in the model, but the root node is an `EventBModel`. Luckily, the strings generated in Section 3.4 are parsed by ProB internally, and the underlying can be accessed using the `.getAst` method, which most nodes in the ProB Event-B Model provide. With some extensions, the existing `ast->lisb` function can translate this AST to the DSL.

In Listing 4, the ProB Java API is used to load an Event-B machine from a Rodin model, which is then converted to the Event-B DSL. A user can evaluate this DSL to IR or interact in any other kind. For example, the DSL of the model could be saved to a file, serving as an alternative to CamilleX.

A problem with the import of the ProB Java API is that many example files for ProB are available only as `.eventb` files. However, these files don't generate static information when they are loaded using the ProB Java API [BCL]. Thus, the `ProBModel` is empty. So `.eventb` files can, unfortunately, not be imported into *lisb* yet.

4 Translation: Classical B to Event-B

Leuschel et al. [LMW20] used a workflow for systems modeling that involved Classical B and Event-B. The modeling process was divided into three distinct phases:

Phase 1, the exploratory phase, heavily focuses on editing and animation. In this phase, Classical B was preferred because its rich substitution language allowed for quick changes in the model. Additionally, having a textual representation simplified editing, collaboration, and versioning (for example, with git).

In phase 2, the functionality of the individual subcomponent has stabilized, but the integration of the components needs some experimentation. For this phase, they again used Classical B, as the powerful inclusion mechanism allows for exploring different ways to combine the system. A refinement-based approach like Event-B requires deciding on a specific refinement order for component integration, which can be challenging and tedious to change. Safety invariants start to be defined during this phase to ensure proper system functioning. Verification in this stage primarily relied on model checking and animation.

Listing 5: Operation of the BlinkLamps machine from [LMW20]

```
SET_BlinkersOn(direction,rem) =
PRE direction:BLINK_DIRECTION &
    rem:BLINK_CYCLE_COUNTER &
    rem /= 0
    THEN
      active_blinkers := {direction} ||
      remaining_blinks := rem ||
      IF direction=right_blink THEN
          blinkLeft := lamp_off ||
          blinkRight := cycleMaxLampStatus(onCycle)
      ELSE
          blinkLeft := cycleMaxLampStatus(onCycle) ||
          blinkRight := lamp_off
    END
END;
```

The final phase is predominantly focused on proof. For this, the Rodin platform is used because of the simplicity of Event-B's proof obligations. With the system decomposition now stabilized, the inclusion can be liberalized into a refinement hierarchy. This translation was done by hand and involved turning machine inclusion into machine refinement or in-lining the machine. Additionally, all substitutions had to be translated into actions supported by Event-B. All of that is very tedious and error-prone, not ideal when dealing with safety critical systems.

In Listing 5, we see the operation `SET_BlinkersOn` of the `BlinkLamps`[1] machine. Manual translation to Event-B yields two events: `SET_RightBlinkersOn` and `SET_LeftBlinkersOn` one for each case of the if substitution. This leads to a considerable amount of code duplication because all substitutions outside the if-then-else have to be mirrored across the two events. In the next Section, we use *lisb* to translate operations with complex substitutions to Event-B by creating multiple events that emulate the same behavior.

4.1 Operations to Event-B

In software modeling, a notion of operations that can be called and can return values is very useful. We focus on general system modeling for which events are more useful. Events cannot be called and don't return any values. Clark [Cla16] created a translation from an Algorithm Description Language (ADL) to Event-B. With this ADL, one can create a procedure and translate it to Event-B. Operations could be translated similarly, but this is out of the scope of this thesis.

[1] All machines from the original paper [LMW20] can be found at `https://github.com/hhu-stups/abz2020-models`

We only concentrate on operations without return values. We handle operations essentially like events. Operation calls are supported but are translated differently than in the ADL; they are described in Section 4.4. The function `op->events` is responsible for converting operations to events. It takes the IR of an operation and returns a list of events also in IR. The function creates a base event representing the operation. Basic attributes of the operation, like name and arguments, are copied to this base event. This base event together with the body of the operation is passed into the function `sub->events`, which splits the event into multiple events, depending on the substitutions contained in the body of the operation. Based on the operation name and the substitutions unique event names are generated. The function `op->events` can be used like in Listing 6.

```
user=> (def op (b (op :foo [:a] (assign :x :a))))
user=> (op->events op)
[{:tag :event, :name :foo, :clauses (
   {:tag :args, :values (:a)},
   {:tag :actions, :values (
     {:tag :assignment, :id-vals (:x :a)})})}]
```

Listing 6: Simple Operation to Event with `op->events`

4.2 Substitutions to Events

In this Section, we discuss how to translate substitutions of Classical B into an equivalent construct in Event-B. Most substitutions can be represented using a before-after predicate. This model could be animated using ProB, but this method of translation would make refinement of the generated events very difficult. So if animation and model checking are the only goals, Classical B could have been used directly.

Alternatively, we have decided to split branching substitution into multiple events. This approach makes the refinement of individual paths through the system possible. The function `sub->event` takes an event and a substitution. It generates the events for the provided substitution. If multiple events are generated, a different string is appended to the name of each generated event. Otherwise, the name does not change. All guards and actions are copied to the new events. If the substitution is a deterministic or non-deterministic assignment it is just added as an actions. Substitutions inside other substitutions are translated recursively, by splitting the resulting event further. Following are the rules for translating substitutions to events. In the code listings, we only show the differences that have to be added to the new events.

The **PRE** clause creates a precondition. We model preconditions as guards, so the event is not split, but only the predicate is added to the guards of the event.

PRE P **THEN** S	**event** name **when** P **then** S

The **IF** substitution consists of a predicate P and two substitutions: S and T. The condition S is only executed when the condition P is true otherwise, T is executed. In Event-B a similar behavior can be achieved with two events. One event for the branch where the condition P is true. In this case, condition P is added as a guard, and the substitution S is applied to the event by calling `sub->events`. Similarly, for the other branch, with the negation $\neg P$ of the condition as a guard and the substitution T.

IF P **THEN** S **ELSE** T	**event** name_then **when** P **then** S **event** name_else **when** $\neg P$ **then** T

The **SELECT** substitution takes n guarded substitutions. A substitution S_i in a branch of the *select* is only executed if the corresponding guard P_i is true. If multiple guards are true, one of the corresponding substitutions is selected non-deterministically and executed. The nearest construct to a *select* substitution in Event-B are individual events for every branch.

SELECT P_1 **THEN** S_1 **WHEN** P_2 **THEN** S_2 ... **WHEN** P_n **THEN** S_n	**event** name_select1 **when** P_1 **then** S_1 ... **event** name_selectn **when** P_n **then** S_n

A select substitution can also contain an optional *else* clause, which is executed when all other cases are false. Therefore, an event with a negation of all other guards has to be created.

SELECT P_1 **THEN** S_1 ... **WHEN** P_n **THEN** S_n **ELSE** S_{else}	**event** name_select1 **when** P_1 **then** S_1 ... **event** name_selectelse **when** $\neg P_1$ ... $\neg P_n$ **then** S_{else}

The **CASE** substitution is similar to a select substitution, but an expression E is provided and case-matched against n different literal values l_i. Again, the case substitution can contain an else clause, which is executed if $E \notin \{l_1, \ldots, l_n\}$, i.e. when no literals are matched.

CASE E **of**
  EITHER l_1 **THEN** S_1
  WHEN l_2 **THEN** S_2
  ...
  WHEN l_n **THEN** S_n
  ELSE S_{else}

event name_l_1
  when $E = l_1$ **then** S_1
event name_l_2
  when $E = l_2$ **then** S_2
  ...
event name_l_n
  when $E = l_n$ **then** S_n
event name_caseelse
  when $E \notin \{l_1, \ldots, l_n\}$ **then** S_{else}

4.3 Parallel Composition

Events execute all their action in parallel. So the parallel composition of actions supported by Event-B can be translated into a single event. It is more complex when other substitutions are involved. But a parallel composition of the branching substitutions can be reduced to the simple form of actions. The reduction of two substitutions S and T executed in parallel, is done by applying T to all the branches of S same time. For example, the *if* substitution can be reduced as follows:

$$(\textbf{IF } P \textbf{ THEN } S_1 \textbf{ ELSE } S_2) \parallel T \quad \Longleftrightarrow \quad \textbf{IF } P \textbf{ THEN } S_1 \parallel T \textbf{ ELSE } S_2 \parallel T$$

Reductions for the other substitutions are similar. For more details, see [Sch01, Section 10.4].

4.4 Inclusion in Event-B

Event-B lacks an inclusion system like Classical B; the only way to structure components is by refinement. In the paper [LMW20], they use Classical B's inclusion mechanism to experiment with different arrangements of components, where operation calls serve as the communication between components. In this section, we want to give two ways for translation machine inclusion into Event-B: Either by in-lining the complete or by representing them as a refinement.

A machine in Classical B may include any number of other machines. By an included machine, we mean a machine that is referenced in the *INCLUDES* or *EXTENDS* clause. An example of a machine created in phase 2 [LMW20] is shown in Listing 7. This machine includes two other machines, and we will use it as a running example throughout this section. In the following, we describe some properties of the inclusion in Classical B. More details about the different types of inclusion can be found in [Sch01]. Let's consider a single machine *M0*, which is included by a machine *M1*.

Listing 7: Operation with two call operation calls from [LMW20]

```
MACHINE PitmanController
INCLUDES BlinkLamps, Sensors
...
OPERATIONS
ENV_Turn_EngineOn = BEGIN
  SET_EngineOn() ||
  IF pitmanArmUpDown :PITMAN_DIRECTION_BLINKING &
     hazardWarningSwitchOn = switch_off THEN
     SET_BlinkersOn(pitman_direction(pitmanArmUpDown),continuousBlink)
  END
END;
```

- All sets and constants of *M0* can be seen by *M1*.
- The machine *M1* has read access to all variables of *M0*. To ensure consistency of *M0* variables in *M0* may only be modified by operations of *M0*.
- The machine *M1* is permitted to call any of *M0*'s operations. It has to be ensured that the precondition of the called operation is met.
- The state of all included machines of *M1* is initialized first, followed by the statements in the parent's initialization.
- Machine *M1* makes operations of *M0* available to its interface by listing the operation in the *promotes* clause of *M1*.
- A machine listed in the *extends* clause is a special case of inclusion, where all operations become part of the interface.

When in-lining the inclusion, we copy all sets, constants, variables, properties, and invariants of the included machine *M0* into *M1*. For sets and constants, copying them is essentially the same behavior as including would yield because they are read-only nonetheless. The variables ,however, are now part of the machine *M1*, so operations in *M1* can now modify them. But as the invariants of *M0* are also copied, the state of the whole machine stays consistent. Operations of *M0* are only copied to *M1* if they are listed in the promotes clause of *M1*. For operation calls, the expressions used as arguments are substituted in the body. Then the body is in-lined into the calling operation. The function `inclusions->inline` performs this in-lining process. The first argument is the parent machine, and the second is the machine to be included.

So far, so good, but in-lining all included machines is not desirable as it would lead to a very large machine. It would be questionable why to translate it to Event-B, when not utilizing the refinement structure to simplify proofs. So we also want to translate an inclusion into a refinement.

We created the function `inclusions->refinement`, which also takes the parent machine and the included machine as arguments and converts the inclusion into a refinement. When a machine inclusion is represented as a refinement, we only have to copy the variables

Listing 8: Automatic translation of `ENV_Turn_EngineOn`

```
event ENV_Turn_EngineOn_then_1 extends SET_BlinkersOn_else
  where
    @grd0 engineOn=FALSE∧keyState=KeyInsertedOnPosition
    @grd1 pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING ∧
                hazardWarningSwitchOn=switch_off
    @grd2 direction=pitman_direction(pitmanArmUpDown)
    @grd3 rem=continuousBlink
  then
    @act0 engineOn :=TRUE
end
```

from the included machine *M0* to *M1*. All operation calls then become event extensions. However, most certainly, an operation in the included machine will be split into multiple events, one for each path of execution. So for every operation call in the refinement, we have to create as many events as the operation produced that was called. The arguments to the operation call are added to the guards as equations.

Now we have two ways of converting inclusions. But finding an optimal refinement chain and combing the two functions is still up to the user. The machine in Listing 7 contains an operation that calls two other operations. The operation `SET_EngineOn` will be in-lined, and `SET_BlinkersOn` will be extended. As we have seen in Listing 5, the operation `SET_BlinkersOn` contains an if-then-else and therefore generates two events. So `ENV_Turn_EngineOn` has to be split into two events, which extend the two cases of `SET_BlinkersOn`. Listing 8 presents one of the two generated events as they are identical except for their name and the extended event.

After resolving all inclusions with the functions described above, the resulting IR can be split into a context and a machine. For this, we created the functions `extract-context` and `extract-machine`. The first extracts the static parts of the machine. The second extracts the dynamic parts from the machine and converts the operation to events using the function `op->events` described in Listing 6.

In the next section, we compare the result from the automatic translation to manual translation performed in the paper.

4.5 Comparison to manual translation

Our translation is very close to the manual translation created in Phase 3. Listing 9 shows the manual translation [LMW20] of the operation ENV_Turn_EngineOn, where the `direction` is `left_blink`. It is nearly the same event as compared to the automatically generated event. However, there are two differences. First, the generated name may not be very readable because it just represents the path of execution. Second, in the

Listing 9: Manual translation of `ENV_Turn_EngineOn` by [LMW20]

```
event ENV_Turn_EngineOn_BlinkLeft extends SET_LeftBlinkersOn
when
    @grd11 engineOn=FALSE ∧keyState =KeyInsertedOnPosition
    @grd12 pitmanArmUpDown ∈PITMAN_DIRECTION_BLINKING
    @grd13 pitman_direction(pitmanArmUpDown)=left_blink
    @grd14 hazardWarningSwitchOn =switch_off
    @grd15 rem =continuousBlink
then
    @act11 engineOn :=TRUE
end
```

manual translation, the value for the parameter `direction` is in-lined. But we naively copy all arguments of operation to all generated events even though they could be in-lined, resulting in uniquely specified parameters. In Appendix B, the complete machines that were generated are listed.

5 Related Work

The ProB Java API itself has many of the same goals as *lisb*. It provides an abstraction to ProB's internals and therefore allows building models. This enables other programs to use ProB as a backbone, as we have seen in this work.

The Rodin plugin CamilleX [Hoa+21] is similar to *lisb*, as it provides an intermediate representation. On top, a text representation of Event-B is given, which can be extended with new features. For example, Hoang et al. created an inclusion mechanism for Event-B using CamilleX. The intermediate representation of CamilleX uses *Eclipse Modelling Framework* and is integrated directly into Rodin, which makes it difficult for other programs to interact with.

As we have seen, the translation from an algorithm description language [Cla16] to Event-B has a similar goal of automatic Event-B generation for Rodin. The ADL is designed for expressing sequential programs. On the other hand, *lisb* provides just a framework for programmatic transformation.

6 Future Work

The extensions to *lisb* developed in this thesis enable Event-B models to be written in Clojure and programmatically transformed. In the future, the following features can be implemented to improve the support for Event-B in *lisb* further:

- Support for user-defined labels in the Event-B DSL and IR.
- Add a way to include models in `.eventB` files.
- Integrating *lisb* into other tools, for example, Rodin.
- The existing translations from classical B could be improved. For example, generating better names.
- More translations from classical B to Event-B could be added.
- Translations from Event-B to Classical B.
- More complex static analysis and constraint solving could be used to create optimizations for the translations.

7 Conclusion

In this thesis, we have extended the existing implementation of *lisb* by the formalism of Event-B. We have created a DSL that is very similar to the Camille Event-B. The DSL generates a pure data representation of Event-B which can be transformed programmatically. Then this representation is converted into a ProB. The unrestrained nature of the internal representation makes it great for prototyping new features.

Another side effect of translating Event-B into Clojure data structures is that the model can be saved in `.edn` files, which is a text representation of the Clojure data structures. The `.edn` format can be versioned using git, for example. Both the DSL and the IR can be saved this way, depending on the verbosity needed.

We used *lisb* to translate machine inclusion and some substitutions to Event-B so that a model can be proven in Rodin. But we have seen that a fully automatic translation from Classical B to Event-B is difficult as there are multiple ways of translation.

Concluding with a discussion about some design decisions. The separation between Event-B and Classical B in the IR made it possible for a lossless representation of Event-B. But the two formalisms are still separated. A conversion between the different internal representations is needed. We have implemented some transformations from Classical B to Event-B. But not the other way around. Another approach to the architecture would be to completely reuse the *lisb* IR for Classical B and translate this IR directly to an Event-B ProB Model. The representation of Event-B is probably not lossless, but the overall architecture is cleaner, where *lisb* provides a common IR with different front- and backends.

References

[Abr10] J.-R. Abrial. *Modeling in Event-B: System and Software Engineering*. New York, NY, USA: Cambridge University Press, May 2010.

[Atel] *Atelier B, User and Reference Manuals*. ClearSy. URL: http://www.atelierb.eu (visited on 07/13/2023).

[Bar20] H. R. Barradas. *Event-B: Syntax and Proof Obligations in Atelier B*. Oct. 21, 2020.

[BCL] J. Bendisposto, J. Clark, and M. Leuschel. *ProB 2.0 Java API Documentation*. URL: https://stups.hhu-hosting.de/handbook/prob2/prob_handbook.html (visited on 07/13/2023).

[Ben+11] J. Bendisposto et al. "Developing Camille, a text editor for Rodin". In: *Software: Practice and Experience* 41 (Jan. 2011), pp. 189–198.

[Cla16] J. Clark. "An Algorithm Description Language for Event-B". MA thesis. Heinrich Heine Universität Düsseldorf, 2016.

[Cloj] *The Clojure Programming Language*. Cognitec Inc. URL: https://clojure.org (visited on 07/13/2023).

[Hic08] R. Hickey. "The Clojure programming language". In: *Proceedings of the 2008 symposium on Dynamic languages*. ACM. 2008.

[Hoa+21] T. S. Hoang et al. "The CamilleX Framework for the Rodin Platform". In: *Rigorous State-Based Methods*. Springer, 2021, pp. 124–129.

[KM22] P. Körner and F. Mager. "An Embedding of B in Clojure". en. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings*. ACM, Oct. 2022.

[Kör+20] P. Körner et al. "Integrating formal specifications into applications: the ProB Java API". In: *Formal Methods in System Design* 58.1-2 (Oct. 2020), pp. 160–187.

[LB08] M. Leuschel and M. Butler. "ProB: An Automated Analysis Toolset for the B Method". In: *International Journal on Software Tools for Technology Transfer* 10 (Mar. 2008), pp. 185–203.

[Leu21] M. Leuschel. "Spot the Difference: A Detailed Comparison Between B and Event-B". In: *Logic, Computation and Rigorous Methods*. Springer, 2021, pp. 147–172.

[LMW20] M. Leuschel, M. Mutz, and M. Werth. "Modelling and Validating an Automotive System in Classical B and Event-B". In: *Rigorous State-Based Methods*. Springer, 2020, pp. 335–350.

[Sch01] S. Schneider. *The B-method an Introduction*. Cornerstones of computing. New York, NY, USA: Palgrave, 2001.

Appendices

A Phones Example

In Listing 10 the internal representation for the machine in Listing 2 is show.

Listing 10: IR for a Event-B machine

```
{:tag :machine, :name :m0, :machine-clauses
({:tag :variables, :values (:phones)}
 {:tag :invariants, :values
  ({:tag :subset, :sets (:phones :PHONES)}
   {:tag :less-equals, :nums (
    {:tag :cardinality, :set :phones} :m)})}
 {:tag :init, :values (
  {:tag :assignment, :id-vals (:phones #{})})}
 {:tag :events, :values
  ({:tag :event,
    :name :OpenPhones,
    :clauses
    ({:tag :guards, :values
      ({:tag :not-equals,
        :left {:tag :cardinality, :set :phones},
        :right :m}
       {:tag :not-equals, :left :phones, :right :PHONES})}
     {:tag :actions, :values
      ({:tag :becomes-such,
        :ids [:phones], :pred
        {:tag :and,
         :preds
         ({:tag :subset, :sets (:phones :phones')}
          {:tag :less-equals,
           :nums ({:tag :cardinality, :set :phones'} :m)})}})})})}
   {:tag :event,
    :name :ClosePhones,
    :clauses
    ({:tag :guards, :values
      ({:tag :not-equals, :left :phones, :right #{}}
       {:tag :not-equals, :left :phones, :right :PHONES})}
     {:tag :actions, :values
      ({:tag :becomes-such,
        :ids [:phones],
        :pred {:tag :subset, :sets (:phones :phones')}})})})})})}
```

B Automotive lighting

Listings 11 to 14 contain the automatically generated Event-B context and machine, from the B machines `Sensors.mch` `BlinkLamps_v3.mch` and `PitmanController_v6.mch`.

Listing 11: Context generated from BlinkLamps

context BlinkLamps_v3_ctx

sets DIRECTIONS

constants
BLINK_DIRECTION
LAMP_STATUS
lamp_on
lamp_off
continuousBlink
BLINK_CYCLE_COUNTER
cycleMaxLampStatus
left_blink right_blink neutral_blink

axioms
@axm0 partition(DIRECTIONS,{left_blink},{right_blink},{neutral_blink})
@axm1 BLINK_DIRECTION={left_blink,right_blink}
@axm2 LAMP_STATUS={0,100}
@axm3 continuousBlink=−1
@axm4 lamp_off=0
@axm5 lamp_on=100
@axm6 BLINK_CYCLE_COUNTER=−1..3
@axm7 cycleMaxLampStatus∈(BOOL)→LAMP_STATUS
@axm8 cycleMaxLampStatus={FALSE↦lamp_off,TRUE↦lamp_on}
end

Listing 12: Machine generated from BlinkLamps

machine BlinkLamps_v3 **sees** BlinkLamps_v3_ctx

variables active_blinkers remaining_blinks onCycle blinkLeft blinkRight

invariants
@inv0 active_blinkers⊆BLINK_DIRECTION
@inv1 remaining_blinks∈BLINK_CYCLE_COUNTER
@inv2 blinkLeft∈LAMP_STATUS
@inv3 blinkRight∈LAMP_STATUS
@inv4 onCycle∈BOOL
@inv5 remaining_blinks=0∧blinkLeft=lamp_off∧blinkRight=lamp_off⇔active_blinkers=∅
@inv6 blinkRight≠lamp_off⇒right_blink∈active_blinkers
@inv7 blinkLeft≠lamp_off⇒left_blink∈active_blinkers
@inv8 active_blinkers=BLINK_DIRECTION⇒blinkLeft=blinkRight
@inv9 onCycle=FALSE⇒blinkLeft=lamp_off∧blinkRight=lamp_off

@inv10 onCycle=TRUE∧active_blinkers≠∅⇒¬(blinkLeft=lamp_off∧blinkRight=lamp_off)

events
event SET_AllBlinkersOff
then
@act0 active_blinkers :=∅
@act1 remaining_blinks :=0
@act2 blinkLeft,blinkRight :=lamp_off,lamp_off
end

event SET_AllBlinkersOn
then
@act0 active_blinkers :=BLINK_DIRECTION
@act1 remaining_blinks :=continuousBlink
@act2 blinkLeft :=cycleMaxLampStatus(onCycle)
@act3 blinkRight :=cycleMaxLampStatus(onCycle)
end

event SET_BlinkersOn_then
any direction rem
where
@grd0 direction∈BLINK_DIRECTION
@grd1 rem∈BLINK_CYCLE_COUNTER
@grd2 rem≠0
@grd3 direction=right_blink
then
@act0 active_blinkers :={direction}
@act1 remaining_blinks :=rem
@act2 blinkLeft :=lamp_off
@act3 blinkRight :=cycleMaxLampStatus(onCycle)
end

event SET_BlinkersOn_else
any direction rem
where
@grd0 direction∈BLINK_DIRECTION
@grd1 rem∈BLINK_CYCLE_COUNTER
@grd2 rem≠0
@grd3 ¬(direction=right_blink)
then
@act0 active_blinkers :={direction}
@act1 remaining_blinks :=rem
@act2 blinkLeft :=cycleMaxLampStatus(onCycle)
@act3 blinkRight :=lamp_off
end

event SET_RemainingBlinks
any rem
where

```
      @grd0 rem∈BLINK_CYCLE_COUNTER
      @grd1 rem≠0
      @grd2 remaining_blinks≠0
    then
      @act0 remaining_blinks :=rem
  end

  event TIME_BlinkerOn_then_then_then
    where
      @grd0 blinkLeft=lamp_off∧blinkRight=lamp_off∧remaining_blinks≠0
      @grd1 left_blink∈active_blinkers
      @grd2 right_blink∈active_blinkers
      @grd3 remaining_blinks>0
    then
      @act0 onCycle :=TRUE
      @act1 blinkLeft :=lamp_on
      @act2 blinkRight :=lamp_on
      @act3 remaining_blinks :=remaining_blinks−1
  end

  event TIME_BlinkerOff_then
    where
      @grd0 ¬(blinkLeft=lamp_off∧blinkRight=lamp_off)
      @grd1 remaining_blinks=0
    then
      @act0 blinkLeft,blinkRight :=lamp_off,lamp_off
      @act1 onCycle :=FALSE
      @act2 active_blinkers :=∅
  end

  event TIME_Nothing
    any newOnCycle
    where
      @grd0 blinkLeft=lamp_off∧
      blinkRight=lamp_off∧
      active_blinkers=∅∧newOnCycle=FALSE
    then
      @act0 onCycle :=newOnCycle
  end
end
```

Listing 13: Context generated from PitmanController

```
context PitmanController_v6_ctx extends BlinkLamps_v3_ctx

sets PITMAN_POSITION SWITCH_STATUS KEY_STATE

constants
pitman_direction
PITMAN_DIRECTION_BLINKING
```

PITMAN_TIP_BLINKING
Neutral Downward5 Downward7 Upward5 Upward7
switch_on switch_off
NoKeyInserted KeyInserted KeyInsertedOnPosition

axioms
@axm0 partition(PITMAN_POSITION,{Neutral},{Downward5},
{Downward7},{Upward5},{Upward7})
@axm1 partition(SWITCH_STATUS,{switch_on},{switch_off})
@axm2 partition(KEY_STATE,{NoKeyInserted},{KeyInserted},{KeyInsertedOnPosition})
@axm3 pitman_direction={
Upward7↦right_blink,Downward7↦left_blink,
Downward5↦left_blink,Upward5↦right_blink,
Neutral↦neutral_blink}
@axm4 PITMAN_TIP_BLINKING={Downward5,Upward5}
@axm5 PITMAN_DIRECTION_BLINKING={Upward7,Downward7}
end

Listing 14: Machine generated from PitmanController

machine PitmanController_v6 **refines** BlinkLamps_v3 **sees** PitmanController_v6_ctx

variables
active_blinkers
pitmanArmUpDown
remaining_blinks
onCycle
blinkRight blinkLeft
keyState hazardWarningSwitchOn engineOn

invariants
@inv0 hazardWarningSwitchOn=switch_off ∧
remaining_blinks=continuousBlink
⇒active_blinkers={pitman_direction(pitmanArmUpDown)}
@inv1 hazardWarningSwitchOn=switch_on⇒remaining_blinks=continuousBlink
@inv2 hazardWarningSwitchOn∈SWITCH_STATUS
@inv3 engineOn=FALSE∧hazardWarningSwitchOn=switch_off⇒active_blinkers=∅
@inv4 pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING ∧
engineOn=TRUE⇒{pitman_direction(pitmanArmUpDown)}⊆active_blinkers
@inv5 hazardWarningSwitchOn=switch_on⇒active_blinkers=BLINK_DIRECTION
@inv6 pitmanArmUpDown∈PITMAN_POSITION
@inv7 engineOn∈BOOL
@inv8 pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING ∧
engineOn=TRUE⇒remaining_blinks=continuousBlink
@inv9 keyState∈KEY_STATE
theorem @thm0 pitman_direction∈PITMAN_POSITION→DIRECTIONS

events
event ENV_Turn_EngineOn_then_0 **extends** SET_BlinkersOn_then
where

```
    @grd0 engineOn=FALSE∧keyState=KeyInsertedOnPosition
    @grd1 pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING ∧
          hazardWarningSwitchOn=switch_off
    @grd2 direction=pitman_direction(pitmanArmUpDown)
    @grd3 rem=continuousBlink
  then
    @act0 engineOn :=TRUE
end

event ENV_Turn_EngineOn_then_1 extends SET_BlinkersOn_else
  where
    @grd0 engineOn=FALSE∧keyState=KeyInsertedOnPosition
    @grd1 pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING ∧
          hazardWarningSwitchOn=switch_off
    @grd2 direction=pitman_direction(pitmanArmUpDown)
    @grd3 rem=continuousBlink
  then
    @act0 engineOn :=TRUE
end

event ENV_Turn_EngineOff_then_0 extends SET_AllBlinkersOff
  where
    @grd0 engineOn=TRUE
    @grd1 hazardWarningSwitchOn=switch_off
  then
    @act0 engineOn :=FALSE
end

event ENV_Pitman_DirectionBlinking_then_0 extends SET_BlinkersOn_then
  any newPos
  where
    @grd0 newPos≠pitmanArmUpDown
    @grd1 hazardWarningSwitchOn=switch_off∧engineOn=TRUE
    @grd2 direction=pitman_direction(newPos)
    @grd3 rem=continuousBlink
    @grd4 newPos∈PITMAN_DIRECTION_BLINKING∧newPos≠pitmanArmUpDown
  then
    @act0 pitmanArmUpDown :=newPos
end

event ENV_Pitman_DirectionBlinking_then_1 extends SET_BlinkersOn_else
  any newPos
  where
    @grd0 newPos≠pitmanArmUpDown
    @grd1 hazardWarningSwitchOn=switch_off∧engineOn=TRUE
    @grd2 direction=pitman_direction(newPos)
    @grd3 rem=continuousBlink
    @grd4 newPos∈PITMAN_DIRECTION_BLINKING∧newPos≠pitmanArmUpDown
  then
```

@act0 pitmanArmUpDown :=newPos
end

event ENV_Pitman_Reset_to_Neutral_then_0 **extends** SET_AllBlinkersOff
where
@grd0 pitmanArmUpDown≠Neutral
@grd1 hazardWarningSwitchOn=switch_off∧remaining_blinks=continuousBlink
then
@act0 pitmanArmUpDown :=Neutral
end

event ENV_Pitman_Tip_blinking_short_then_0 **extends** SET_BlinkersOn_then
any newPos
where
@grd0 newPos∈PITMAN_TIP_BLINKING
@grd1 newPos≠pitmanArmUpDown
@grd2 newPos∈PITMAN_TIP_BLINKING∧newPos≠pitmanArmUpDown
@grd3 hazardWarningSwitchOn=switch_off∧engineOn=TRUE
@grd4 direction=pitman_direction(newPos)
@grd5 rem=3
then
@act0 pitmanArmUpDown :=newPos
end

event ENV_Pitman_Tip_blinking_short_then_1 **extends** SET_BlinkersOn_else
any newPos
where
@grd0 newPos∈PITMAN_TIP_BLINKING
@grd1 newPos≠pitmanArmUpDown
@grd2 newPos∈PITMAN_TIP_BLINKING∧newPos≠pitmanArmUpDown
@grd3 hazardWarningSwitchOn=switch_off∧engineOn=TRUE
@grd4 direction=pitman_direction(newPos)
@grd5 rem=3
then
@act0 pitmanArmUpDown :=newPos
end

event TIME_Tip_blinking_Timeout_0 **extends** SET_RemainingBlinks
where
@grd0 pitmanArmUpDown∈PITMAN_TIP_BLINKING ∧
remaining_blinks>1 ∧
active_blinkers={pitman_direction(pitmanArmUpDown)}
@grd1 rem=continuousBlink
end

event ENV_Hazard_blinking_select0_0 **extends** SET_AllBlinkersOn
any newSwitchPos
where
@grd0 newSwitchPos≠hazardWarningSwitchOn

```
    @grd1 newSwitchPos=switch_on
    @grd2 newSwitchPos∈SWITCH_STATUS∧newSwitchPos≠hazardWarningSwitchOn
  then
    @act0 hazardWarningSwitchOn :=newSwitchPos
end

event ENV_Hazard_blinking_select1_cond0_0 extends SET_AllBlinkersOff
  any newSwitchPos
  where
    @grd0 newSwitchPos≠hazardWarningSwitchOn
    @grd1 newSwitchPos=switch_off
    @grd2 pitmanArmUpDown=Neutral ∨engineOn=FALSE
    @grd3 newSwitchPos∈SWITCH_STATUS∧newSwitchPos≠hazardWarningSwitchOn
  then
    @act0 hazardWarningSwitchOn :=newSwitchPos
end

event ENV_Hazard_blinking_select1_cond1_0 extends SET_AllBlinkersOff
  any newSwitchPos
  where
    @grd0 newSwitchPos≠hazardWarningSwitchOn
    @grd1 newSwitchPos=switch_off
    @grd2 ¬(pitmanArmUpDown=Neutral ∨engineOn=FALSE)
    @grd3 ¬(pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING)
    @grd4 newSwitchPos∈SWITCH_STATUS∧newSwitchPos≠hazardWarningSwitchOn
  then
    @act0 hazardWarningSwitchOn :=newSwitchPos
end

event ENV_Hazard_blinking_select1_condelse_0 extends SET_BlinkersOn_then
  any newSwitchPos
  where
    @grd0 newSwitchPos≠hazardWarningSwitchOn
    @grd1 newSwitchPos=switch_off
    @grd2 ¬(pitmanArmUpDown=Neutral ∨engineOn=FALSE)
    @grd3 ¬(¬(pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING))
    @grd4 direction=pitman_direction(pitmanArmUpDown)
    @grd5 rem=remaining_blinks
    @grd6 newSwitchPos∈SWITCH_STATUS∧newSwitchPos≠hazardWarningSwitchOn
  then
    @act0 hazardWarningSwitchOn :=newSwitchPos
end

event ENV_Hazard_blinking_select1_condelse_1 extends SET_BlinkersOn_else
  any newSwitchPos
  where
    @grd0 newSwitchPos≠hazardWarningSwitchOn
    @grd1 newSwitchPos=switch_off
    @grd2 ¬(pitmanArmUpDown=Neutral ∨engineOn=FALSE)
```

@grd3 ¬(¬(pitmanArmUpDown∈PITMAN_DIRECTION_BLINKING))
@grd4 direction=pitman_direction(pitmanArmUpDown)
@grd5 rem=remaining_blinks
@grd6 newSwitchPos∈SWITCH_STATUS∧newSwitchPos≠hazardWarningSwitchOn
then
@act0 hazardWarningSwitchOn :=newSwitchPos
end
end

List of Figures

List of Listings