Reasoning about Liveness Properties in Event-B

Author(s):
Hoang, Thai Son; Abrial, Jean-Raymond

Publication date:
2011

Permanent link:
https://doi.org/10.3929/ethz-a-006699021

Funding acknowledgement:
247277 - Automated Urban Parking and Driving (EC)

Reasoning about Liveness Properties in Event-B *

Thai Son Hoang[1] and Jean-Raymond Abrial[2]

[1] Deparment of Computer Science,
Swiss Federal Institute of Technology Zurich (ETH-Zurich), Switzerland
htson@inf.ethz.ch
[2] Marseille, France
jrabrial@neuf.fr

Abstract. Event-B is a formal method which is widely used in modelling safety critical systems. So far, the main properties of interest in Event-B are safety related. Even though some liveness properties, e,g, termination, are already within the scope of Event-B, more general liveness properties, e.g. progress or persistence, are currently unsupported. We present in this paper proof rules to reason about important classes of liveness properties. We illustrate our proof rules by applying them to prove liveness properties of realistic examples. Our proof rules are based on several proof obligations that can be implemented in a tool support such as the Rodin platform.
Keywords: Event-B, liveness properties, formal verification, tool support.

1 Introduction

Event-B [1] is a formal modelling method for discrete state transition systems and is based on first-order logic and some typed set theory. The backbone of the method is the notion of step-wise refinement, allowing details to be gradually added to the formal models. An advantage of using refinement is that any (safety) properties that are already proved to hold in the early models are guaranteed to hold in the later models. This is crucial in a method for developing systems correct-by-construction. System development using Event-B is supported by the RODIN platform [2]. It is an open and extensible platform containing facilities for modelling and proving Event-B models.

So far, most of the properties that are proved in Event-B are safety properties, i.e., something (bad) never happens, which are usually captured as invariants of the models. Although it is essential to prove that systems are safe, it might not be sufficient. Considering an elevator system, an important safety property is that *the door must be closed while the elevator is moving*. However, an unusable non-moving elevator also satisfies this safety property. Hence, it is desirable to be able to specify and prove that the system also satisfies some *liveness* properties, e.g., it is always that case that a request will eventually be served.

Currently, besides safety properties that are captured as *invariants*, Event-B can only be used to model certain liveness properties, e.g., *termination*. More general classes of liveness properties, such as *progress* or *persistence* are unsupported. On the one hand,

* Part of this research was supported by DEPLOY project (EC Grant number 214158).

we want to increase the set of properties that can be specified and verified in Event-B. On the other hand, we aim to keep the reasoning practical, so that we can easily have tool support to generate and discharge obligations.

We propose a set of *proof rules* for reasoning about three different classes of liveness properties. The rules are based on some basic *proof obligations*, that can be conveniently implemented in the supporting Rodin platform of Event-B. The first proof rule is for proving *existence* properties stating that something will *always eventually* occur ($\Box \diamondsuit P$). The second proof rule is for reasoning about *progress* properties: something must eventually happen if some condition becomes true ($\Box(P_1 \Rightarrow \diamondsuit P_2)$). The third proof rule is for proving *persistence* properties: *eventually*, something *always* holds ($\diamondsuit \Box P$).

The rest of the paper is organised as follows. Section 2 gives an overview of the Event-B modelling method and temporal logic. Our main contribution is in Section 3 including proof rules for the previously mentioned properties. Section 4 illustrates the applicability of our rules to some realistic examples. We briefly elaborate our ideas for tool supports in Section 3.3. Finally, we draw some conclusion (Section 6), discuss related work (Section 5), and investigate future research directions (Section 6.1).

2 Background

2.1 The Event-B Modelling Method

A model in Event-B, called *machine*, contains a vector of state variables v and a set of events $\mathsf{evt_i}$. Each event has the form $\mathsf{evt_i} \mathrel{\widehat{=}}$ **any** x **where** $G_i(x, v)$ **then** $A_i(x, v, v')$ **end**, where x are parameters of the event, $G_i(x, v)$ is the guard and $A_i(x, v, v')$ is the action. The guard of an event is the necessary condition for the event to be enabled. The action of an event comprises several assignments, each has one of the following forms: $v := E(x, v)$, $v :\in E(x, v)$, and $v :| Q(x, v, v')$.

Assignments of the first form deterministically update variables v with values of $E(x, v)$. Assignments of the latter two forms are nondeterministic. They update variable v to be either some member of a set $E(x, v)$ or satisfying a before-after predicate $Q(x, v, v')$. The first two forms of assignment can be represented using the last form with the corresponding before-after predicates $v' = E(x, v)$ and $v' \in E(x, v)$. Assignments of an action are supposed to be executed in parallel. Each event therefore corresponds to a before-after predicate $\mathsf{A}(x, v, v')$ by conjoining the before-after predicate of all assignments and the predicate $u = u'$ where u is the set of variables unchanged by the action. A dedicated event, called init, without parameters and guard is used as the initialisation of the machine. An after predicate Init is associated with init.

Variables v are constrained by invariant $I(v)$ which needs to be proved to hold in every reachable state of the system. This is guaranteed by proving that the invariant is *established* by the initialisation init and subsequently *preserved* by all events.

To overcome the complexity in system development, Event-B advocates the use of *refinement*: a process of gradually adding details to a model. A development in Event-B is a sequence of machines, linked by some refinement relationship: an *abstract* machine is refined by the subsequent *concrete* machine. Abstract variables v are linked to concrete variables w by some *gluing invariant* $J(v, w)$. Any behaviour of the concrete

model must be *simulated* by some behaviour of the abstract model, with respect to the gluing invariant $J(v, w)$.

An Event-B machine corresponds to a state transition system: the states s are captured as tuples $\langle \overline{v} \rangle$, representing the values of variables v; and the events correspond to transitions between states. An event evt is said to be *enabled* in a state s if there exists some parameter x such that the guard G of the event hold in that state s. Otherwise, the event is said to be *disabled*. A machine M is said to be *deadlocked* in s if all its events are disabled in that particular state.

Given an event evt, we say a state t is an evt-*successor state* of s if t is a possible after-state of the execution of evt from the before-state s. Lifting the definition to a machine M, we say that t is an M-*successor state* of s if there exists an event evt of M such that t is an evt-successor of s.

A *trace* σ of a machine M is a sequence of states (either finite or infinite) $s_0, s_1, \ldots$ satisfying the following conditions.

- s_0 is an initial state, i.e., satisfying the initial after predicate Init.
- For every two successive states s_i and s_{i+1}, s_{i+1} is an M-successor state of s_i.
- If the sequence is finite and ends in some state s_{final} then machine M is deadlocked in s_{final}.

Finally, a machine M is associated with a set of traces $\mathcal{T}(\mathsf{M})$ denoting all of its possible traces.

2.2 Temporal Logic

We give a summary of the (propositional) LTL temporal logic similar to the one defined by Manna and Pnueli [10]. We will consider temporal formulas to be interpreted over the sequences of states arising from machine traces.

The basic element of the language is a *state formula* P: any first-order logic formula. It describes some property that holds in some state s. It is built from terms and predicates over the program variables v. The extended temporal formulas are constructed from these basic state formulas by applying the Boolean operators $\neg, \wedge, \vee, \Rightarrow$ and temporal operators: always ($\Box$), eventually ($\Diamond$) and until ($\mathcal{U}$).

Let σ be a non-empty, finite or infinite, sequence of states of the form $s_0, s_1, \ldots$. We use the standard notation $\sigma \vDash \phi$ to denote that σ satisfies formula ϕ. We first define some notations that will be used in the interpretation of temporal formulas.

- States satisfying a state formula P are called P-states.
- The length of the trace σ denoted as $l(\sigma)$ is defined as follows. If σ is finite, i.e., of the form $s_0, \ldots, s_k$, $l(\sigma) = k + 1$. If σ is infinite, $l(\sigma) = \omega$, the least infinite ordinal number.
- Given a number $0 \leq k < l(\sigma)$, a k-suffix sequence of states of σ denoted as σ^k is the sequence of states obtained by dropping the first k elements from σ, i.e., $\sigma^k = s_k, s_{k+1}, \ldots$.

The interpretation of the LTL formulas over σ is as follows.

- For a state formula P, $\sigma \vDash P$ iff s_0 is a P-state.

– The Boolean operators are interpreted intuitively.

$$\begin{array}{lll} \sigma \vDash \phi_1 \wedge \phi_2 & \text{iff} & \sigma \vDash \phi_1 \text{ “and” } \sigma \vDash \phi_2 \ , \\ \sigma \vDash \phi_1 \vee \phi_2 & \text{iff} & \sigma \vDash \phi_1 \text{ “or” } \sigma \vDash \phi_2 . \\ \sigma \vDash \neg \phi & \text{iff} & \text{“not” } \sigma \vDash \phi \ , \\ \sigma \vDash \phi_1 \Rightarrow \phi_2 & \text{iff} & \sigma \vDash \phi_1 \text{ “then” } \sigma \vDash \phi_2 \ . \end{array}$$

– The temporal operators are interpreted as follows.

$$\begin{array}{lll} \sigma \vDash \Box \, \phi & \text{iff} & \text{for all } k \text{ where } 0 \leq k < l(\sigma), \text{ we have } \sigma^k \vDash \phi. \\ \sigma \vDash \diamondsuit \, \phi & \text{iff} & \text{there exists } k \text{ where } 0 \leq k < l(\sigma), \text{ such that } \sigma^k \vDash \phi. \\ \sigma \vDash \phi_1 \, \mathcal{U} \, \phi_2 & \text{iff} & \text{there exists } k, \text{ where } 0 \leq k < l(\sigma), \text{ such that} \\ & & \quad \sigma^k \vDash \phi_2, \text{ and} \\ & & \quad \text{for all } i \text{ such that } 0 \leq i < k, \text{ we have } \sigma^i \vDash \phi_1. \end{array}$$

In the case where we have some state predicates P, P_1, P_2, the combination with the temporal operators can be understood as follows.

$$\begin{array}{lll} \sigma \vDash \Box \, P & \text{iff} & \text{every state in } \sigma \text{ are } P\text{-state.} \\ \sigma \vDash \diamondsuit \, P & \text{iff} & \text{there exits some } P\text{-state in } \sigma. \\ \sigma \vDash P_1 \, \mathcal{U} \, P_2 & \text{iff} & \text{there exists some } P_2\text{-state } s_k \text{ in } \sigma, \\ & & \text{and every state until } s_k \text{ (excluding } s_k) \text{ is } P_1\text{-state.} \end{array}$$

Definition 1 *A machine* M *is said to satisfy property* ϕ *(denoted as* $\mathsf{M} \vDash \phi$*) if all its traces satisfy* ϕ*, i.e.,* $\forall \sigma \in \mathcal{T}(\mathsf{M}) \cdot \sigma \vDash \phi$.

In subsequent proof rules, we use the notation $\mathsf{M} \vdash \phi$ to denote that $\mathsf{M} \vDash \phi$ *is provable*.

3 Proof Rules

In this section we present some proof rules to reason about important classes of liveness properties. We progress by first presenting some basic proof obligations as building blocks for the later proof rules. We assume here that there is a machine M with events of the general form mentioned in Section 2.1.

3.1 Proof Obligations

Machine M leads from P_1 to P_2 Given two state formulas P_1, P_2, we say M *leads from* P_1 *to* P_2 if for any pair of successor states (s_i, s_{i+1}) of any trace of M, if s_i is a P_1-state then s_{i+1} is a P_2-state. We first define the *leads from* notion for events.

An event evt leads from P_1 to P_2 if starting from any P_1-state, execution of every event leads to a P_2-state. This is guaranteed by proving the following (stronger) condition.

$$P_1(v) \wedge G(x, v) \wedge \mathsf{A}(x, v, v') \Rightarrow P_2(v')$$

(We adopt the convention that free variables in proof obligations are universally quantified.)

Given the above definition, M *leads from* P_1 *to* P_2 if every event evt of M leads from P_1 to P_2.

Proof. From the definition of machine trace in 2.1 and of *leads from* notion for events.

In subsequent proof rules, we use the notation $\mathsf{M} \vdash P_1 \curvearrowright P_2$ to denote that this fact is provable. Note that the property can be stated in terms of k-suffix as follows. For any pair of successor state s_i, s_{i+1} of any k-suffix of any trace of M, if s_i is a P_1-state then s_{i+1} is a P_2-state.

Machine M is convergent in P The obligation allows us to prove that any trace of M *does not end with an infinite sequence of P-states*. Equivalently, the property can be stated as: any k-suffix of any trace of M also *does not end with an infinite sequence of P-state*. This can be guaranteed by reasoning about the *convergence* property of the events in M as follows.

- An integer expression $V(v)$ (called the *variant*) is defined.
- For every event evt of M, we prove that
  1. When in a P-state, if evt is enabled, $V(v)$ is a natural number[3].

$$P(v) \wedge G(x, v) \Rightarrow V(v) \in \mathbb{N}$$

  2. An execution of evt from a P-state decreases $V(v)$.

$$P(v) \wedge G(x, v) \wedge \mathsf{A}(x, v, v') \Rightarrow V(v') < V(v)$$

Proof. If a trace ends with an infinite sequence of P-states, then $V(v)$ will be decreased infinitely (condition 2). However, since in P-states, $V(v)$ is a member of a well-founded set (condition 1), this results in a contradiction.

In the subsequent proof rules, we use $\mathsf{M} \vdash \downarrow P$ to denote that this fact (i.e., M is convergent in P) is provable.

Machine M is divergent in P This obligation allows us to prove that any infinite trace of M *ends with an infinite sequence of P-states*. An equivalent property is that any infinite k-suffix of any trace of M also ends with an infinite sequence of P-states.

- An integer expression $V(v)$ (called the *variant*) is defined.
- For every event evt of M, we prove the following conditions.
  1. When in a $\neg P$-state, if evt is enabled, $V(v)$ is a natural number.
  2. An execution of evt from a $\neg P$-state decreases the value of the variant.

$$\neg P(v) \wedge G(x, v) \wedge \mathsf{A}(x, v, v') \Rightarrow V(v') < V(v)$$

  3. An execution of evt from a P-state *does not increase* $V(v)$ if the new value of the variant $V(v')$ is a natural number.

$$P(v) \wedge G(x, v) \wedge \mathsf{A}(x, v, v') \wedge V(v') \in \mathbb{N} \Rightarrow V(v') \leq V(v)$$

Proof. Condition 1 and 2 guarantees that the variant V is a member of an well-founded set (e.g., $\mathbb{N}$) and decreases when $\neg P$ holds, and condition 3 ensures that this decreasing *cannot be undone* when P holds. Hence if M has an infinite trace, then $\neg P$-states will eventually disappear from it.

In the subsequent proof rules, we use $\mathsf{M} \vdash \nearrow P$ to denote that this fact (i.e., M is divergent in P) is provable.

[3] More generally, the variant can be a member of any well-founded set.

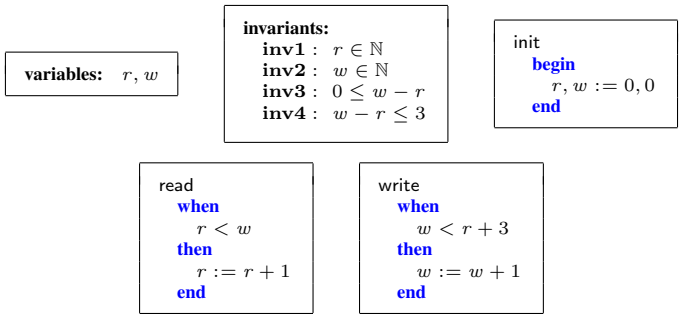


Fig. 1. Machine RdWr: Reader/Writer

Machine M is deadlock-free in P This obligation allows to prove that any (finite) trace of M does not end in a P-state. This is guaranteed by proving that in a P-state, at least one event of M is enabled, i.e., M is deadlock-free in any P-state.

$$P(v) \quad \Rightarrow \quad \bigvee_i (\exists x \cdot G_i(x, v))$$

Proof. From the definition of machine trace in Section 2.1.

Note that an equivalent property is as follows: any k-suffix of any trace of M is also deadlock-free in P, i.e., does not end in a P-state.

In the subsequent proof rules, we use $\mathsf{M} \vdash \circlearrowleft P$ to denote that this fact is provable.

3.2 Proof Rules

We are going to use the example of Reader/Writer in **Fig.** 1 to illustrate our proof rules in this section. The machine models a system with two processes *Reader* and *Writer* sharing a common bounded buffer. The machine has two variables, namely r and w (both initialised to 0), representing the current pointer of the reader and the writer. The "buffer" is the range $[r + 1, w]$ representing the data that have been written but not yet read. The size of the buff is $w - r$. Assuming that the buffer can hold a maximum of 3 pieces of data, we must have $0 \leq w - r$ and $w - r \leq 3$ as invariants of the model.

There are two events: read and write for reading and writing, respectively. Event read increases the read pointer r by 1, when there are some unread data. Similarly, event write advances the writer pointer w, when there is still some space in the buffer.

Invariance Safety properties are usually captured in Event-B machine as invariants. The proof rules for invariance properties are well-known and already *built-into* for Event-B. They are also used in [10]. We restate the rules mentioned in Section 2.1 in terms of the above proof obligations as follows.

$$\frac{\begin{array}{l} \vdash \mathsf{Init} \Rightarrow I \\ \mathsf{M} \vdash I \curvearrowright I \end{array}}{\mathsf{M} \vdash \Box I} \ \mathbf{INV}_{\mathbf{induct}}$$

The above rule $\mathbf{INV_{induct}}$ allows us to prove invariance properties which are inductive. Otherwise, i.e. when invariance properties are not inductive, the following proof rule can be used.

$$\frac{\vdash J \Rightarrow I \qquad \mathsf{M} \vdash \Box J}{\mathsf{M} \vdash \Box I}\ \mathbf{INV_{theorem}}$$

Informally, rule $\mathbf{INV_{theorem}}$ allows us to prove that I is an invariance property relying on a stronger additional invariant J. Subsequently, we can make use of the inductive rule $\mathbf{INV_{induct}}$ to prove that J is an invariance property,

Invariance properties are important in reasoning about the correctness of our models since it limits the set of reachable states. In the subsequent proof rules, we adopt the convention that already proved invariance properties can be assumed and hence, do not mention them explicitly.

Existence An existence property states that some (good) property, say P, will *always eventually* hold. The following proof rule allows us to prove that a machine M satisfies an existence property $\Box \diamondsuit P$ by reasoning about convergence and deadlock-freedom.

$$\frac{\mathsf{M} \vdash \downarrow \neg P \qquad \mathsf{M} \vdash \circlearrowleft \neg P}{\mathsf{M} \vdash \Box \diamondsuit P}\ \mathbf{LIVE}_{\Box \diamondsuit}$$

Proof. Consider a trace σ of machine M. Consider a k-suffix σ^k of σ. If σ^k is an infinite sequence of states, according to the first antecedent, it cannot end with an infinite sequence of $(\neg P)$-states, hence a P-state eventually appears. The second antecedent ensures that in the case where σ^k is finite, it does not end in a $(\neg P)$-state, i.e., it must end in a P-state.

Example 1 *We want to prove that for* RdWr*, eventually $r \geq L$ for some natural number L. Our reasoning is as follows.*

$$\boxed{\mathsf{RdWr} \vdash \Box \diamondsuit\, r \geq L} \xrightarrow{\mathbf{LIVE}_{\Box \diamondsuit}} \begin{cases} \boxed{\mathsf{RdWr} \vdash \downarrow (\neg r \geq L)} & (1) \\ \boxed{\mathsf{RdWr} \vdash \circlearrowleft (\neg r \geq L)} & (2) \end{cases}$$

(1) $\mathsf{RdWr} \vdash \downarrow (\neg r \geq L)$*: we propose a variant $V_1 = (L - r) + (L + 3 - w)$, a sum of two terms, which are decreased accordingly by* read *and* write*. The fact that the variant is a natural number when the events are enabled in a $(\neg r \leq L)$-state is a consequence of invariant* $\mathbf{inv4}$.

(2) $\mathsf{RdWr} \vdash \circlearrowleft (\neg r \geq L)$*: According to the proof obligation for proving deadlock-freedom, we have to prove that $\neg r \geq L \Rightarrow r \neq w \vee w \neq r + 3$ which holds trivially.*

Progress A progress property states that a P_1-state must always be followed eventually by a P_2-state. For a machine M, the property can be formalised as follows $\mathsf{M} \vdash \Box(P_1 \Rightarrow \Diamond\, P_2)$. In order to reason about progress properties, we introduce two proof rules. The first one $\mathbf{Until}$ deals with a special form of progress properties where P_1 is stable (i.e., holds until P_2 holds). The second one $\mathbf{LIVE_{progress}}$ deals with a more general form of progress properties by "inventing" an auxiliary property.

$$\frac{\mathsf{M} \vdash (P_1 \wedge \neg P_2) \leadsto (P_1 \vee P_2) \qquad \mathsf{M} \vdash \Box\Diamond(\neg P_1 \vee P_2)}{\mathsf{M} \vdash \Box(P_1 \Rightarrow (P_1 \,\mathcal{U}\, P_2))}\ \mathbf{Until}$$

$$\frac{\mathsf{M} \vdash \Box(P_1 \wedge \neg P_2 \Rightarrow P_3) \qquad \mathsf{M} \vdash \Box(P_3 \Rightarrow (P_3 \,\mathcal{U}\, P_2))}{\mathsf{M} \vdash \Box(P_1 \Rightarrow \Diamond\, P_2)}\ \mathbf{LIVE_{progress}}$$

Proof. The rules are justified as follows.

- $\mathbf{Until}$: Consider a trace σ of machine M. Consider a k-suffix σ^k of σ where s_k is a P_1-state. We have to prove that there exists a state P_2-state s_m in trace σ (with $k \leq m$) such that any state between s_k and s_m (excluding s_m) is a P_1-state.
  - If s_k is also a P_2-state, then we can take m to be k.
  - If s_k is a $\neg P_2$-state, then it is also a $(P_1 \wedge \neg P_2)$-state. From the second antecedent, we know that eventually, there is a $(\neg P_1 \vee P_2)$-state. Let s_m be the first such state (hence $k \leq m$). We will prove that s_m is indeed the state that we are looking for.
    - Since s_m is the first state after s_k satisfying $\neg P_1 \vee P_2$, any state in between s_k and s_m excluding s_m is a $(P_1 \wedge \neg P_2)$-state, i.e., is a P_1-state.
    - Since s_k is a $(P_1 \wedge \neg P_2)$-state, and s_m is a $(\neg P_1 \vee P_2)$-state, they must be different, i.e., $k \neq m$, hence s_{m-1} is a state in between s_k and s_m. Subsequently, s_{m-1} must be a $(P_1 \wedge \neg P_2)$-state. Together with the first antecedent of the rule, s_m is a $(P_1 \vee P_2)$-state. Since s_m is both a $(\neg P_1 \vee P_2)$-state and a $(P_1 \vee P_2)$-state, it must be a P_2 state.
- $\mathbf{LIVE_{progress}}$: Rule $\mathbf{LIVE_{progress}}$ relies on auxiliary state predicate P_3 and its justification is as follows. The first antecedent states that $P_1 \wedge \neg P_2 \Rightarrow P_3$ is an invariant of machine M and the second antecedent states an until-property where P_3 holds until eventually P_2 holds. Consider any k-suffix σ^k of a trace σ of machine M, where s_k is a P_1-state. If s_k is also a P_2-state then the progress property holds trivially. Otherwise, i.e., if s_k is a $\neg P_2$-state, according to the first antecedent, s_k must be a P_3-state. The second antecedent then allows us to conclude that there exits a P_2-state s_m where $k \leq m$.

Example 2 *Consider machine* RdWr. *We want to prove that the reader can always make some progress, which is formalised by* $\mathsf{RdWr} \vdash \Box(w = L \Rightarrow \Diamond\, r = L)$*, for some natural number* L*. Our reasoning starts by applying rule* $\mathbf{LIVE_{progress}}$ *with the auxiliary state predicate* P_3 *to be* $r < L$*.*

$$\boxed{\mathsf{RdWr} \vdash \Box(w = L \Rightarrow \Diamond\, r = L)} \xrightarrow{\mathbf{LIVE_{progress}}} \begin{cases} \boxed{\mathsf{RdWr} \vdash \Box(w = L \wedge \neg r = L \Rightarrow r < L)} & (3) \\ \boxed{\mathsf{RdWr} \vdash \Box(r < L \Rightarrow (r < L \,\mathcal{U}\, r = L))} & (4) \end{cases}$$

(3) *The fact that* $w = L \wedge \neg r = L \Rightarrow r < L$ *is a consequence of invariant* $\mathbf{inv3}$*, i.e. proved by rule* $\mathbf{INV_{theorem}}$.

(4) *We apply rule* **Until** *as follows.*

$$\boxed{\mathsf{RdWr} \vdash \square(r < L \Rightarrow (r < L \;\mathcal{U}\; r = L))}$$

$$\xrightarrow{\textbf{Until}} \begin{cases} \boxed{\mathsf{RdWr} \vdash (r < L \wedge \neg r = L) \curvearrowright (r < L \vee r = L)} & (4.1) \\ \boxed{\mathsf{RdWr} \vdash \square\Diamond(\neg r < L \vee r = L)} & (4.2) \end{cases}$$

(4.1) *This sub-goal can be simplified as* $\mathsf{RdWr} \vdash (r < L) \curvearrowright (r \leq L)$*, i.e., we must prove that* read *and* write *lead from* $r < L$ *to* $r \leq L$*, which is trivial. For* read*, we need to prove* $r < L \wedge r \neq w \Rightarrow r + 1 \leq L$*.*

(4.2) *This sub-goal is equivalent to* $\mathsf{RdWr} \vdash \square\Diamond(r \geq L)$ *which we proved in* ***Example 1****.*

Persistence A persistence property states that P must eventually hold forever. Formally, this is expressed for a machine M as $\mathsf{M} \vdash \Diamond\square P$.

$$\frac{\begin{array}{l}\mathsf{M} \vdash \nearrow P \\ \mathsf{M} \vdash \circlearrowleft \neg P\end{array}}{\mathsf{M} \vdash \Diamond\square P}\ \textbf{LIVE}_{\Diamond\,\square}$$

Proof. Consider any trace σ of machine M. The first antecedent guarantees that if the σ is infinite, it will end with an infinite sequence of P-states. The second antecedent ensures that if σ is finite, it cannot end with a $(\neg P)$-state. Together, we know that the σ ends with (finite or infinite) sequence of P-states.

Example 3 *Consider machine* RdWr*. We want to prove that* $\Diamond\square(L \leq w)$ *for some natural number* L*. We start by applying rule* $\textbf{LIVE}_{\Diamond\,\square}$*.*

$$\boxed{\mathsf{RdWr} \vdash \Diamond\square(L \leq w)} \xrightarrow{\textbf{LIVE}_{\Diamond\,\square}} \begin{cases} \boxed{\mathsf{RdWr} \vdash \nearrow (L \leq w)} & (5) \\ \boxed{\mathsf{RdWr} \vdash \circlearrowleft (\neg L \leq w)} & (6) \end{cases}$$

(5) *We use* $V_2 = (L - w) + (L - r)$ *as our variant.*
1. *In* $(\neg L \leq w)$*-states,* V_2 *is a natural number (thanks to invariant* $r \leq w$*).*
2. *Both* read *and* write *decrease* V_2*, hence, they satisfy conditions for decreasing the variant in* $(\neg L \leq w)$*-states and not increasing the variant in* $L \leq w$*)-states.*

(6) *We have to prove that* $\neg L \leq w \Rightarrow r \neq w \vee w \neq r + 3$*, which is trivial.*

3.3 Tool Support

Our proof rules are based on several basic proof obligations that can be easily realised in a tool support such as the Rodin platform. In particular, the proof obligations related to our illustrated examples in Section 4 are indeed simulated within the current Rodin platform, relying on the standard proof obligation generators. For example, the proof obligations for proving that *a machine* M *is convergent in* P is encoded by adding P to guards of all events and prove the new events are convergent with some variant V.

Other proof obligations, e.g., deadlock-freedom, are mostly encoded as *theorems* in the models. The generated obligations are discharged using the current proving support within the platform.

We propose to extend the Event-B models with clauses corresponding to the different liveness properties. For example, the existence property in Example 1 can be specified as follows.

existence:
exst1 : $r \geq L$
variant $(L - r) + (L + 3 - w)$

Note that we also need to include the declaration for the variant used in proving convergence properties. With this declaration, related proof obligations for ensuring the existence property can be generated accordingly.

Similarly, we can define the following clauses for progress and persistence properties. Note that we also indicate the auxiliary invariant for the progress property, in addition to the declaration of the variant.

progress:
prog1 : **from** $w = L$ **to** $r = L$
invariant $r < L$
variant $(L - r) + (L + 3 - w)$

persistence:
stbl1 : $L \leq w$
variant $(L - w) + (L - r)$

We are working on extending the Rodin platform to include these newly proposed clauses.

4 Examples

We illustrate our application of proof rules for *existence* and *progress* properties with Peterson's mutual exclusion algorithm (Section 4.1). The proof rule for *persistence* is illustrated in our example of a device calibration in Section 4.2.

4.1 Peterson's Algorithm

Description Peterson's algorithm [13] involves two processes P_a and P_b. It is a mutual exclusion algorithm: at most one process shall be in the, so-called, "critical section". It uses three variables: w_a, w_b, and $turn$, elements of the set $\{0, 1\}$.

When $w\_a = 1$ (resp. $w\_b = 1$), it means that P_a (resp. P_b) wishes to enter the critical section or is in the critical section. When $turn = 0$, it means that it is P_a's turn to enter the critical section (if it wishes to do so), and similarly with $turn = 1$ for P_b.

Initially, we have $w\_a = 0$, $w\_b = 0$ (i.e., no process wishes to enter the critical section), and $turn$ takes any value in $\{0, 1\}$. Here are the skeletal sequential programs supposed to be executed concurrently:

```
P_a
  while true do
    // Wishing to Enter the Critical Section
    w_a := 1;
    turn := 1;
    // Busy Waiting
    while ¬(w_b = 0 ∨ turn = 0) do
      SKIP
    end
    // Enter Critical Section
    ...
    w_a := 0
    // Leave Critical Section
    ...
  end
```

```
P_b
  while true do
    // Wishing to Enter the Critical Section
    w_b := 1;
    turn := 0;
    // Busy Waiting
    while ¬(w_a = 0 ∨ turn = 1) do
      SKIP
    end
    // Enter Critical Section
    ...
    w_b := 0
    // Leave Critical Section
    ...
  end
```

As can be seen, each process enters a "busy waiting" loop before entering the critical section. Each of them waits until the proper conditions to enter the critical section hold. For P_a, it corresponds to waiting either that P_b does not wish to enter the critical section ($w\_b = 0$) or that it is P_a's turn to enter the critical section ($turn = 0$). We have similar busy waiting conditions for P_b.

We would like to prove two things:

Mutual exclusion At most one process can be in the critical section at a time.
Progress A process wishes to enter the critical section will eventually do so.

Refinement Strategy We shall proceed with three models. The initial model will handle the **mutual exclusion** problem: it is independent from the Peterson's algorithm. The first refinement deals with Peterson's algorithm: we shall have to prove that this refinement indeed refines the initial model and thus obeys the mutual exclusion property. Finally, the second refinement deals with the **progress** property.

The Initial Model

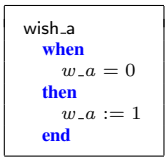
Variables and Invariants In this initial model, besides variables w_a and w_b as defined in the description, we introduce two more variables, a and b, members of the set $\{0, 1\}$. When $a = 1$, it means that P_a is in the critical section, and similarly with b for P_b.

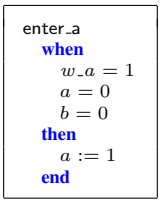
variables: a, b, w_a, w_b

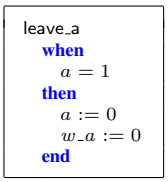
invariants:
inv0_1 : $w\_a = 0 \Rightarrow a = 0$
inv0_2 : $w\_b = 0 \Rightarrow b = 0$
inv0_3 : $a = 0 \vee b = 0$

Invariant **inv0_1** says that when P_a does not wish to enter the critical section ($w\_a = 0$) then it is certainly not in the critical section ($a = 0$). Invariant **inv0_2** defines a similar property for P_b. Invariant **inv0_3** formalises the mutual exclusion property: at most one process can be in the critical section at a time.

Events All variables are initialised with 0. Next are events associated with P_a. There are three events describing the way P_a can enter and leave each phase: wishing to enter the critical section, entering the critical section, or leaving it. The sequentiality of the events is ensured by the fact that P_a can be in exactly one situation at a time, either $w\_a = 0$ or $w\_a = 1 \wedge a = 0$ or $w\_a = 1 \wedge a = 1$.







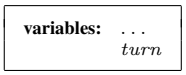
The events for P_b are similar. The two processes are obviously concurrent as there is clearly some non-determinacy between the events of one and those of the other.

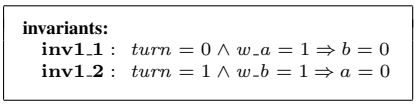
As can be seen, process P_a enters the critical section if it is not in it ($a = 0$) and if process P_b is also not in it ($b = 0$): the mutual exclusion property is ensured. However, we have two problems here: (1) the checking by P_a of the situation of P_b by looking at b, (2) we have no guarantee that one process does not always enter the critical section while the other one wants to do it as well. These problems will be addressed in the subsequent refinements.

Proofs There are 25 proof obligations all discharged automatically by the Rodin prover.

First Refinement

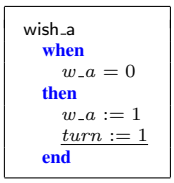
Variables and Invariants Besides variables introduced in the initial model, we add now the variable $turn$ as defined in the description.

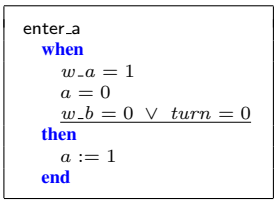


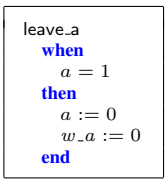


Invariant **inv1_1** is needed in order to prove guard strengthening in event enter_a. This is because in this event we remove the reference to variable b. Invariant **inv1_2** plays a similar role for P_b.

Events These events deal with P_a. In the event enter_a, the guard $b = 0$ has been replaced by the guard $w\_b = 0 \lor turn = 0$ that does not make any reference to b.







We have similar events for P_b which are omitted.

Proofs The Proof Obligation Generator of the Rodin platform produces 18 proof obligations all discharged automatically.

Second Refinement In this refinement, we shall prove the progress property for process P_a by encoding the proof obligations in the Rodin platform (see Section 3.3). The property can be stated as follows:

$$\mathsf{Peterson} \vdash \Box(w\_a = 1 \Rightarrow \diamondsuit\, a = 1)$$

i.e. if process P_a wishes to enter the critical section ($w\_a = 1$) then it will eventually be able to do so. In fact P_a remains wishing to enter the critical section until it enters it. Hence we can apply proof rule $\mathbf{LIVE_{progress}}$ with $w\_a = 1$ as the auxiliary invariant. The first subgoal is trivial, proving that $w\_a = 1 \wedge \neg a = 1 \Rightarrow w\_a = 1$, the second subgoal is $\mathsf{Peterson} \vdash \Box(w\_a = 1 \;\Rightarrow\; w\_a = 1 \;\mathcal{U}\; a = 1)$.

According to rule **Until**, we have to prove the following two statements:

$$\mathsf{Peterson} \vdash (w\_a = 1 \;\wedge\; a = 0) \curvearrowright (w\_a = 1 \;\vee\; a = 1)$$
$$\mathsf{Peterson} \vdash \Box\diamondsuit(w\_a = 0 \;\vee\; a = 1)$$

The first statement generates 6 proof obligations that are all discharged trivially. According to the proof rule $\mathbf{LIVE}_{\Box\,\diamondsuit}$, the second statement leads to the following:

$$\mathsf{Peterson} \vdash \downarrow (w\_a = 1 \;\wedge\; a = 0)$$
$$\mathsf{Peterson} \vdash \circlearrowleft (w\_a = 1 \;\wedge\; a = 0)$$

The first of these statement requires finding a decreasing variant, which we propose

$$V_3 = 2 * w\_b + 3 * \mathit{turn} - b - a$$ [4]

The fact that this variant is a natural number whenever an event is enabled and under the assumption $w\_a = 1 \;\wedge\; a = 0$ generates 6 proof obligations that are easily discharged. The fact that this variant is decreased by every event under the assumption $w\_a = 1 \;\wedge\; a = 0$ generates 6 proof obligations that are easily discharged provided we add the following additional invariant:$w\_a = 1 \wedge w\_b = 0 \Rightarrow \mathit{turn} = 1$. The second statement requires to prove that the model is deadlock free under the assumption $w\_a = 1 \;\wedge\; a = 0$. It is easily discharged.

Proofs We have to prove 24 proof obligations. The prover of the Rodin platform proves them all automatically except two easy ones that were proved interactively. All in all, we have 67 proof obligations all proved automatically except two of them.

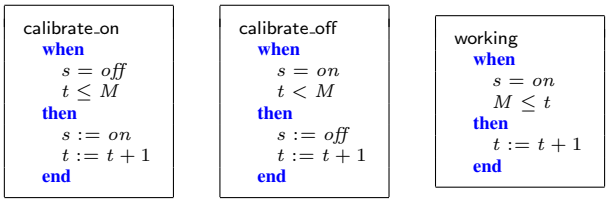
At this level of details, our Event-B model allows common variables to be accessed and modified concurrently. If we are interested in the precise atomicity assumption on common variables, e.g. turn, it is possible to decompose the events wish_a, wish_b, enter_a, and enter_b so that some new events treat with turn only (together with some address counters for sequencing). Introducing these details would only add some complication to the illustration of our proof rules.

4.2 Device Calibration

We now consider a second example. A certain device can be either on or off. Calibration allows the device to be adjusted. During calibration, the status of the device can alternate. We assume that the duration of the calibration process is limited and model the system as follows. A Boolean variable s denotes the status of the device, and an integer variable t denote the current time, initialised to be 0. The machine Calibration contains three events, each of them advances t by 1. When t is less than some constant M then calibration happens, alternating the status s between on and off (events

[4] V_3 is a lexicographic variant: $(\mathit{turn}, w\_b, -(a + b))$, with decreasing order of precedences.

calibrate_on and calibrate_off). When t is greater than M and the device is on then the device works normally, no more calibration occurs, only the time t advances (event working)



We want to prove that eventually, the device will be persistent in the on state, i.e. Calibration $\vdash \diamondsuit \square\, s = on$. Applying our $\mathbf{LIVE}_{\diamondsuit\,\square}$ results in two sub-goals.

- Calibration $\vdash \nearrow (s = on)$. We propose the following variant $V_4 = M - t$.
  - To prove that V_4 is a natural number when $\neg s = on$, we add the following invariant $\mathbf{inv0\_1}$ stating that $s = off \Rightarrow t \leq M$.
  - All events increase t hence decrease the variant V_4, hence they certainly decrease V_4 when $\neg s = on$ and do not increase V_4 when $s = on$.
- Calibration $\vdash \circlearrowleft (s = off)$. For this, we must prove that when $s = off$ then one of the events is enabled, and in our case, it is calibrate_on, according to $\mathbf{inv0\_1}$.

We encode the verification conditions in the Rodin platform, resulting in 19 proof obligations, all of them are discharged automatically by the built-in provers.

5 Related Work

The idea of combining different reasoning features, e.g., *invariant*, *event convergence* and *deadlock-freedom* to prove liveness properties has been presented in our earlier work [8]. There we prove liveness properties characterising when a system reaches stable states. Here, we extend this idea to prove some other important classes of liveness properties. In designing our proof rules, we have been inspired by the pioneering work of Chandy and Misra [4], of Lamport [9], and in particular of Manna and Pnueli [10].

Our proof rules for *progress* properties are similar to that of Manna and Pnueli [10], in the sense that we both use the variant technique to reason about convergence of events. However, our **Until** rule for proving $P_1 \Rightarrow (P_1 \,\mathcal{U}\, P_2)$ has the additional assumption $\neg P_2$ in its sub-goals, i.e., we need to prove the sub-goal only when the desirable condition P_2 has not yet been established. The use of the variant technique is clearly an advantage over the proof lattices approach from Owicki and Lamport [12] when the systems have infinitely many states. In UNITY [4], reasoning about progress properties is embedded within its logic by several proof rules. Our proof rules are comparable to a combination of their transitivity, implication and induction rules. An important motivation for us is to be able to realise the reasoning about liveness properties in a tool support. In our opinion the rules from [10,4] are not at the level which can be realised practically.

In [3], Abrial and Mussat have addressed the problem of verifying *progress* properties, by formulating the problem in terms of proving loops termination. Our proof rules

are stronger than those in [3]. In particular when proving $P_1 \Rightarrow (P_1 \,\mathcal{U}\, P_2)$, we allow the triggering condition P_1 to be invalidated as soon as the desirable condition P_2 holds, i.e., proving that $\mathsf{M} \vdash (P_1 \wedge \neg P_2) \curvearrowright (P_1 \vee P_2)$, whereas in [3], a stronger condition was proposed, i.e., $\mathsf{M} \vdash (P_1 \wedge \neg P_2) \curvearrowright P_1$.

Within our knowledge, there are no practical proof rules existing for *persistence* properties. A stronger persistence property can be defined in the work of Chandy and Misra [4] by combining an *existence* property, e.g., $\diamondsuit P$ and a *stable* properties, e.g., $P \Rightarrow \Box P$. Whereas in their work a persistence predicate remains hold once it holds, we allow a persistence predicate to be invalidated, before becoming stable. In particular, our notion *divergence* by proving a *non-increasing variant* is novel.

Proving general LTL properties in Event-B has been consider in [6]. The approach taken is to encode in B the Büchi automata equivalent to the LTL properties, and then synchronise the resulting machine with the original event system. Several analyses are done on the combined machine, including proving that eventually some accepting state of the Büchi automata will be reached. The downside of this approach is that the reasoning is done on the combined machine containing the original machine with the representation of the LTL property, which increases the complexity of the verification process, for example, finding the appropriate variant.

6 Conclusions

We have presented our proposed proof rules for reasoning about three types of liveness properties in Event-B: *existence*, *progress* and *persistence*. These classes cover a significant numbers of properties that are used in practice. According to the survey done by Dwyer, Avrunin and Corbett [5], amongst over 500 examples of property specifications that they have collected, 27% are invariance properties (in terms of global *absence* and *universality* properties). The class of existence and progress properties cover 45%. Altogether with our extension, we can model in Event-B 72% instead of 27% of the collected properties.

Another practical aspect of our proof rules is that they rely on some basic reasoning obligations which can be implemented straight away in a tool support such as the Rodin platform. This requires only to add to the platform a new declaration and to extend the proof obligation generator for generating appropriate proof obligations. These conditions can be proved within the scope of the existing provers, i.e., there is no need for extending the proving support of the platform.

6.1 Future Work

The main difference between our proof rules and those in [10,12,4] is that we have not yet considered (strongly/weakly) fairness assumptions. This will be necessary later, especially in modelling concurrent and distributed systems. At the moment, we regard this as future work and expect to have some proof rules using similar proof obligations.

In this paper, we do not attempt to have a complete set of proof rules (even for the set of properties under consideration). We rather to come up with some practical rules for some reasonable important subset of properties. Future work along the direction of having a relative complete set of rules can be inspired from [11].

A direct extension of our proof rules is to include the notion of *probabilistic* convergence [7]. This allows us to model systems with *probabilistic* behaviours and reason about properties such as "eventually certain condition holds with *probability one*". An example is the proof of Rabin's choice coordination algorithm to guarantee that eventually, with probability one, all processes agree on a particular alternative [14].

An important future research direction is to investigate how liveness properties can be maintained during refinement. While safety properties are maintained by refinement in Event-B, more investigation need to be done to ensure that liveness properties are preserved during refinement with the possibility of strengthening the refinement notion. In [6], the author proposes a notion of refinement *oriented by the property*. Since the definition depends on the LTL property of interest, references to this property will need to be carried along the refinement chain. We are looking for a notion of refinement preserving our interested set of liveness properties without confining to similar restriction.

Acknowledgement We would like to thank anonymous reviewers for their constructive comments. We also thank David Basin, Andreas Fürst, Dominique Méry and Matthias Schmalz for their help with various drafts of the paper.

References

1. J-R. Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 2010.
2. J-R. Abrial, M. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.
3. J-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B*, volume 1393 of *LNCS*, pages 83–128. Springer, 1998.
4. K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.
5. M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE*, pages 411–420, 1999.
6. J. Groslambert. Verification of LTL on B event systems. In *B*, volume 4355 of *LNCS*, pages 109–124, 2007.
7. S. Hallerstede and T.S. Hoang. Qualitative probabilistic modelling in Event-B. In *iFM*, volume 4591 of *LNCS*, pages 293–312, Oxford, U.K., July 2007. Springer Verlag.
8. T.S. Hoang, H. Kuruma, D. Basin, and J-R. Abrial. Developing topology discovery in Event-B. *Sci. Comput. Program.*, 74(11-12):879–899, 2009.
9. L. Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.
10. Z. Manna and A. Pnueli. Adequate proof principles for invariance and liveness properties of concurrent programs. *Sci. Comput. Program.*, 4(3):257–289, 1984.
11. Z. Manna and A. Pnueli. Completing the temporal picture. *Theor. Comput. Sci.*, 83(1):91–130, 1991.
12. S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.
13. G. Peterson. Myths about the mutual exclusion problem. *Inf. Process. Lett.*, 12(3):115–116, 1981.
14. E. Yilmaz and T.S. Hoang. Development of Rabin's choice coordination in Event-B. Technical report, University of Dusseldorf, 2010. Proceedings of AVoCS'10.