Software and System Modelling manuscript No.
(will be inserted by the editor)

The original publication is available at http://dx.doi.org/10.1007/s10270-015-0456-2
Software & Systems Modeling February 2016, Volume 15, Issue 1, pp 1091-1116

The Unit-B Method — Refinement Guided by Progress Concerns

Simon Hudon · Thai Son Hoang · Jonathan S. Ostroff

Friday 6th March, 2015

Abstract We present Unit-B, a formal method inspired by Event-B and UNITY. Unit-B aims at the stepwise design of software systems satisfying safety and liveness properties. The method features the novel notion of coarse and fine schedules, a generalisation of weak and strong fairness for specifying events' scheduling assumptions. Based on events schedules, we propose proof rules to reason about progress properties and a refinement order preserving both liveness and safety properties. We illustrate our approach by an example to show that systems development can be driven by not only safety but also liveness requirements.

Keywords progress properties · refinement · fairness · scheduling · Unit-B · proof-based formal methods · verification of cyber-physical systems.

1 Introduction

Developing systems satisfying their desirable properties is a non-trivial task. Formal methods have been seen as a possible solution to the problem. Given the increasing complexity of systems, many formal methods adopt refinement techniques, where systems are developed step-by-step in a property preserving manner. In this way, a system's details are gradually introduced into its design within a hierarchical development.

System properties are often categorised into two classes: *safety* and *liveness* [15]. A safety property ensures that undesirable behaviours will never happen during system executions. A liveness property guarantees that eventually desirable behaviours will happen. Ideally, systems should be developed in such a way that they satisfy both their safety and liveness requirements. Although safety properties are often considered the more important ones, we argue that having *live* systems is also important. A system that is safe but not live can be useless. For example, consider an elevator system that does not move. Such an elevator system is safe (nobody gets hurt), yet worthless. According to a survey [8], liveness properties (in terms of *existence* and *progress*) amount to 45% of the overall system properties.

1.1 Motivation

In many refinement-based development methods (e.g., B [1], Event-B [2], VDM [14], Z [22]), the focus is on preserving safety properties. A common problem for such safety-oriented methods is that when applying them to the design of a system, it is possible to make the design so safe that it becomes unusable. This would happen if we strengthened the guards of the events (in Event-B) or choose strong preconditions (in B, Z, VDM) to facilitate the proof of safety properties but in such a way that, in cases where the operations or events are needed to make the system progress, they are not enabled. Concretely, in an elevator system, this might result in a controller which eventually stops opening the door to the elevator (possibly despite there being people inside) in order to satisfy the safety property that the door not be opened between floors. It is hence our aim to design

This is an extended version of [13]

Simon Hudon
Electrical Engineering & Computer Science, York University, Toronto, Canada
E-mail: simon@cse.yorku.ca

Thai Son Hoang
Institute of Information Security, ETH Zurich, Switzerland
E-mail: htson@inf.ethz.ch

Jonathan S. Ostroff
Electrical Engineering & Computer Science, York University, Toronto, Canada
E-mail: jonathan@cse.yorku.ca.

a refinement framework preserving both safety and liveness properties.

UNITY [4] has a calculus for liveness but does not support refinement of programs. Specifications are written in the UNITY logic (a subset of temporal logic) and implementations are programs (or transition systems). The initial specification can be refined by a stronger set of temporal properties but once the temporal properties are implemented, further refinement of the programs is not possible.

Event-B [2] has a calculus for refinement of safety properties, but does not provide much support for liveness and fairness. Instead, Event-B provides the notion of convergence. In a system, a set of events are convergent if they cannot prevent the other events from happening. This can be used, for instance, to develop model of a sequential program, to prove that the program terminates. Convergence is proven by choosing a variant for the system, i.e., an expression whose type is well ordered (e.g., natural numbers or finite sets). Then, it must be proved that all convergent events are guaranteed to decrease the variant whenever they are executed.

In Event-B, liveness properties cannot be directly expressed and proved. One justifies the validity of a liveness property (e.g. $\Box\Diamond$evt, i.e., infinitely often, event evt occurs) by showing that the system is deadlock free and that events other than evt are convergent. However, one must show that deadlock freedom is preserved in each following refinement, and that all new events are convergent. If a spontaneous event (i.e. non-convergent) is needed in a refinement (e.g., an event representing an environmental action), liveness is no longer preserved. Also, only one liveness property per system can be supported.

Our Unit-B method [11] is inspired by the treatment of liveness in UNITY and refinement in Event-B. It improves on both methods by offering a notion of refinement that preserves liveness applicable to reactive and distributed systems. It does this by the introduction of coarse and fine schedules on events and event indices.

In the subsequent, we present a small example to contrast Event-B's safety-based style of reasoning with Unit-B's liveness-driven style. We present two high level models (one in Event-B and the other in Unit-B) of a mutual exclusion protocol. In each model, we show the important safety and liveness properties that one can prove. More specifically, we study three requirements (1) mutual exclusion (safety), (2) minimal progress and (3) individual progress.

1.1.1 An Event-B Model

The Event-B model, in Figure 1, formalises a set of processes (*Pcs*) each of which is in one of three states: *idle*, *waiting*, and *cs* (i.e., in their *critical section*). The state of every process is recorded in the (global) variable *st* (see invariant inv0). The safety requirement of the protocol, that of mutual exclusion, is captured by invariant inv1 and can be proved at this level of abstraction.

variables : st

invariants :
- inv0 : $st \in Pcs \to \{idle, waiting, cs\}$
- inv1 : $(\forall p,q:\ p \neq q:\ \neg(st.p = cs \wedge st.q = cs))$

events :

request $\widehat{=}$ status *convergent*
- any p where $st.p = idle$ then $st.p := waiting$ end

enter $\widehat{=}$ status *ordinary*
- any p where $st.p = waiting \wedge (\forall q:\ p \neq q:\ st.q \neq cs)$ then
  - $st.p := cs$
- end

exit $\widehat{=}$ status *convergent*
- any p where $st.p = cs$ then $st.p := idle$ end

Note: $p, q \in Pcs$ is implicitly assumed.

Fig. 1 Event-B mutual exclusion specification

In order to reason about liveness in this Event-B system, we need a variant. The variant is chosen on the basis of the exact property that we want to demonstrate. We are interested in proving continuous progress, i.e., as long as there are processes waiting to enter their critical section, some process will get to enter. In linear time temporal logic, this is formulated as:

$$\Box(\ (\exists p ::\ st.p = waiting)\ \Rightarrow\ \Diamond(\exists p ::\ st.p = cs)\ ) \qquad \text{(prg0)}$$

In this property, the p in $st.p = waiting$ and the p in $st.p = cs$ are not necessarily the same and individual processes might wait forever. There are two ways in which an execution of the Event-B model might fail to satisfy this (weak) liveness property:

1. The system executes forever but, after a point where some processes are waiting, only events request and exit are taken.
2. The system deadlocks, i.e. terminates, in a state where some processes are still waiting.

Issue 1 can be addressed by using the following variant and by making events request and exit convergent (as in Figure 1).

variant : $2 \times (\#p ::\ st.p = cs)\ +\ (\#p ::\ st.p = idle)$

In the above expression, the notation $(\#x:\ R:\ T)$, the counting quantifier, is used to designate the number of values of x that satisfy T given that they satisfy R. Convergent events are required to decrease this natural number variant. Event exit decreases the first term by 2 and increases the second term by 1 while event request decreases the second term and

leaves the first term unchanged. The two events are therefore convergent.

This line of reasoning proves that any (possibly partial) execution of the system where enter is not executed must be finite. It also follows that any infinite execution of the model includes an infinite number of occurrences of enter. In this case, it means that enter has to occur infinitely many times in infinite executions.

Issue 2 is addressed by ensuring that, at any time, there is at least one enabled event. This is known as a proof of *deadlock freedom*.

$$
\begin{array}{lll}
dlf: & (\exists\, p ::\ st.p = idle) & \text{// request} \\
\vee & \left(\exists\, p ::\ \begin{array}{l} st.p = waiting \\ \wedge\ (\forall\, q :\ q \neq p :\ st.q \neq cs) \end{array}\right) & \text{// enter} \\
\vee & (\exists\, p ::\ st.p = cs) & \text{// exit}
\end{array}
$$

In this small system, deadlock freedom is easy to prove. However, the size of its formulation grows with the number of events of the system and it cannot, in general, be broken down into smaller proof obligations.

In addition, the (weak) liveness property (prg0) is not automatically satisfied by refinements of the system. In order to preserve it, we need to make convergent all the events introduced in successive refinements and we need to prove relative deadlock freedom at each level of abstraction, a burden that only grows more daunting as a development progresses.

As mentioned earlier, this does not prove individual progress of the processes involved in the protocol. We would like to prove the following (strong) liveness property:

$$(\forall\, p ::\ \Box(\ st.p = waiting\ \Rightarrow\ \Diamond st.p = cs\ ))$$

i.e., every process waiting eventually enters its critical section. It is not possible to prove such individual progress using the Event-B model of Figure 1. In order to do such a proof, we would need to include in the model a description of a scheduler. In other words, a low level design is necessary even for a high level liveness property. This is contrary to the idea of refinement: properties should be provable at the level of abstraction and the level of details to which they pertain. This is what Unit-B accomplishes.

Notational Convention. The examples in this paper rely heavily on discrete mathematics and predicate calculus. With the exception of function application, we borrow the set-theoretic and relational notation from the Event-B book [2]; function application, written $f.x$ with f the function and x the argument, as well predicate calculus and generalized quantifier notation are taken from E.W. Dijkstra [6].

In Dijkstra's relativised quantifier notation, $(\forall x : R : T)$ and $(\exists x : R : T)$, with R the range of the quantifications and T the term, are equivalent to the more common $(\forall x \bullet R \Rightarrow T)$ and $(\exists x \bullet R \wedge T)$. The notation for Temporal Logic is taken from [17].

```
events :
  request [p] ≙
    when st.p = idle then st.p := waiting end
  enter [p] ≙
    during st.p = waiting upon (∀ q : p ≠ q : st.q ≠ cs) begin
      st.p := cs
    end
  exit [p] ≙
    during st.p = cs begin st.p := idle end
```

Fig. 2 Unit-B mutual exclusion specification

1.1.2 A Unit-B Model

Figure 2 shows a Unit-B model for the same problem as Figure 1. The Unit-B has the same set of variables and invariants as the Event-B model from Figure 1. Figure 2 only shows the events of the Unit-B model.

In addition to the Event-B constructs, the Unit-B model features three new ones: event coarse schedules, introduced by the keyword during; event fine schedules, introduced by the keyword upon and event indices, denoted by the square brackets next the the event names. Intuitively, if the coarse schedules of an event hold *continually* and its fine schedules becomes true *infinitely often* then the event is executed infinitely often.

For all events (i.e., request, enter, exit) , p is an index instead of an Event-B parameter (declared with the keyword any). While a parameter is conceptually a value chosen non-deterministically, an index suggests that there exists a distinct version of the event, including a separate scheduling assumption, for every one of the index's values. This allows us to prove individual progress for each process p.

Event request is syntatically similar to its Event-B counter part. Semantically, the difference is subtle but important. In Event-B, if request is the only enabled event, i.e. its guard is true and the guard of every other event is false, request will be taken eventually. In Unit-B, even if request is the only enabled event, it might never occur. This is because request is not scheduled: it features neither a coarse schedule (declared with during) nor a fine schedule (declared with upon).

Events enter and exit are scheduled events: enter has both a coarse schedule and a fine schedule and exit has only a coarse schedule. In the case of exit, when its coarse schedule is continually true, i.e. some process is in its critical section and remains there, then eventually exit is taken and p exits its critical section. Event enter is eventually taken if a process p is waiting continually (coarse schedule) and that infinitely often no process is in its critical section (fine schedule). The fine schedule ensures that enter occurs de-

spite the other processes going in and out of their critical sections.

The notion of schedule allows us to prove that certain events are guaranteed to occur without having to reference the other events. This is in contrast to Event-B where the only way to ensure that enter is taken is to make sure that it is eventually the only event enabled.

The next step is to formulate the liveness requirement. In UNITY logic, on which Unit-B is based, the absence of livelock (i.e. prg0) is formulated as:

$$(\exists\, p ::\ st.p = waiting) \ \leadsto\ (\exists\, p ::\ st.p = cs)\ .$$

It reads "whenever a process is waiting it eventually follows that a process, possibly a different one, will gain access to its critical section."

Although the absence of livelock is an interesting property, it is too weak to be useful; the goal of the processes is not to allow an arbitrary other process to carry on with its work; it is rather the goal of the mutual exclusion protocol to let the processes go about their business unhindered, independently from each other. This means that it is more important for the purpose of each process not to be left to wait forever than any other property that has to do with the competing processes. Therefore, we choose individual progress as the central property to be proved. Its UNITY formulation is:

$$st.p = waiting \ \leadsto\ st.p = cs \qquad \text{(prg1)}$$

The free variable p is implicitly universally quantified over the whole formula.

In the process of proving (prg1), we will discover that another progress property is required. This is because the only way that every process can safely have a turn in their critical section is for no process to linger in theirs forever. We formulate it as (prg2) and prove it first:

$$true \ \leadsto\ (\forall\, p ::\ \neg st.p = cs) \qquad \text{(prg2)}$$

It reads "infinitely often, every process will be simultaneously out of their critical section." In LTL, they are stated as:

$$(\forall\, p ::\ \Box(\ st.p = waiting\ \Rightarrow\ \Diamond st.p = cs\ )) \qquad \text{(prg1}')$$
$$\Box\Diamond(\forall\, p ::\ \neg st.p = cs) \qquad \text{(prg2}')$$

The standard way of proving a liveness property in Unit-B is to use rules from UNITY logic to transform the property into something that is more easily proved using the events. The rules will be explained in more details in Section 3. For the sake of conciseness, we only sketch the intuition behind the proofs of this example.

A sketched proof of (prg2) As long as $(\forall\, p ::\ \neg st.p = cs)$ does not hold, there exists a process p is in its critical section, i.e., the coarse schedule of exit$[p]$ is true. Therefore, according to its scheduling assumption, exit$[p]$ eventually will, thus establishing $(\forall\, p ::\ \neg st.p = cs)$ in the process thanks to the mutual exclusion invariant, inv1.

A sketched proof of (prg1) Given a process p, enter$[p]$ is the only event that can establish $st.p = cs$ and it does so if $st.p = waiting$ (its coarse schedule) holds continuously and $(\forall\, q :\ p \neq q :\ \neg st.q = cs)$ (its fine schedule) is true infinitely many times. The latter is entailed by (prg2) which we already proved. The former condition is satisfied as soon as $st.p = waiting$ (the antecedent of (prg1)) is established. .

It is very important to note that the index p allows us to specify the scheduling assumptions on a process-by-process basis. We can thus assert and prove that every process will eventually acquire the lock (prg1), a property otherwise known as *starvation freedom*. This property cannot be proved in Event-B. In Event-B we can only prove the weaker property (prg0) that some arbitrary process will eventually acquire the lock.

Traditionally, scheduling assumptions fall into two categories: weak fairness and strong fairness. In this example, weak fairness (stating that if a process p is waiting and the lock is free *continually* then p eventually takes the lock) is insufficient to prove that process p eventually takes the lock. Normally, to prove this property, the enter$[p]$ event would be scheduled with strong fairness (stating that if a process p is waiting and the lock is free *infinitely often* then p eventually holds the lock). Using strong fairness, the coarse schedule (conditions required to hold continually) and the fine schedule (conditions required to become true infinitely often) would be wrapped in a single guard, thus intertwining the reasoning about those two aspects. By decoupling these orthogonal considerations, we can reason about process p waiting separately from the lock becoming free. Moreover, during refinement, this decoupling will allow us to trade freely between the coarse and the fine schedules. The distinction between coarse and fine schedules and their relation to other scheduling assumptions are explained further in Section 3 and Section 4.4.

The combination of the progress preserving refinement calculus with the novel notions of coarse and fine schedules makes it possible in Unit-B to introduce liveness properties at any stage of a development process. Reasoning about both safety and liveness can be done at the relevant abstractions. As a consequence, not only do we use liveness requirements to rule out any design decision that would be too conservative, but we also use them to guide us to the right design decisions. As a result, liveness properties, in particular progress properties, drive the development process.

1.2 Contribution

This paper features a formal semantics for Unit-B models and their properties, alongside an example of application of Unit-B to a non- trivial control problem. The semantics is formulated in computation calculus [7]. We use it to formally prove the soundness of the rules for reasoning about temporal properties and refinement relationships in Unit-B.

In the past, Unit-B has been used to design a mutual exclusion algorithm [11] and a signal controller for a train station [13]. This paper is an extended version of the latter. In addition to the contributions of [13], we (1) strengthen the separation between the formal semantics and the proof obligations; (2) present a new rule permitting the reuse of progress properties without reproving them; (3) formulate the refinement rules so as to allow the refinement of events to be justified using only one rule; (4) elaborate the individual refinement steps of the example with the design concerns that guide it and the specific proof obligations; (5) expand the example with two refinement steps leading to the specification of a controller and (6) illustrate the use of inductive proofs of liveness in the example.

1.3 Structure

The rest of the paper is organised as follows. In Section 2, we review Dijkstra's computation calculus [7] which we used to formulate our semantics and design our proofs. We follow with a description of the Unit-B method in Section 3. We demonstrate the method and its refinement rules by developing a signal control system in Section 4. We summarise our work in Section 5 including discussion about related work and future work.

2 Background: Computation Calculus

In this section, we give a brief introduction to computation calculus, based on [7]. This will be the basis for defining the semantics of Unit-B models, defining the semantics of temporal properties (both safety and liveness) and formulating the proof of soundness of the Unit-B refinement rules in Section 3.

- In Section 2.1, we introduce the notion of computation predicates which can be manipulated algebraically. We use them in Section 3 to characterize the execution of Unit-B models as well as their properties.
- In Section 2.2, we introduce state predicates, a special case of computation predicates which we use in Section 3 to formalize Unit-B invariants, progress and safety properties, events' guards and schedules.
- In Section 2.3, we introduce atomic computation predicates, a special case of computation predicates which we use in Section 3 to formalize the meaning of the events' actions.

Let $\mathscr{S}$ be the *state space*: a non-empty set of "states". Let $\mathscr{C}$ be the *computation space*: a set of non-empty (finite or infinite) sequences of states henceforth referred to as "computations".

2.1 Computation Predicates

Definition 1 (Computation Predicates) *The set of computation predicates CPred is defined as follows:*

$$CPred \quad = \quad \mathscr{C} \to \mathbb{B}\,, \tag{1}$$

i.e., functions from computations to Booleans.

The standard Boolean operators of the predicate calculus are lifted, i.e., extended to apply to *CPred*. For example, assuming $s,t \in CPred$ and $\tau \in \mathscr{C}$, we have,[1]

$$(s \Rightarrow t).\tau \quad \equiv \quad (s.\tau \Rightarrow t.\tau) \tag{2}$$

$$(\forall i ::\ s.i).\tau \quad \equiv \quad (\forall i ::\ s.i.\tau)\,. \tag{3}$$

The everywhere-operator quantifies universally over all computations, i.e.,

$$[\,s\,] \quad \equiv \quad (\forall \tau ::\ s.\tau)\,. \tag{4}$$

Whenever there are no risks of ambiguity, we shall use $s = t$ as a shorthand for $[\,s \equiv t\,]$ for computation predicates s,t.

Postulate 1 *CPred is a predicate algebra.*

A consequence of Postulate 1 is that *CPred* satisfies all postulates for the predicate calculus as defined in [5]. In particular, **true** (maps all computations to TRUE) and **false** (maps all computations to FALSE) are the "top" and the "bottom" elements of the complete Boolean lattice with the order $[\,\_ \Rightarrow \_\,]$ specified by these postulates. The lattice operations are denoted by various Boolean operators including $\wedge, \vee, \neg, \Rightarrow$, etc.

The predicate algebra is extended with sequential composition as follows.

Definition 2 (Sequential Composition)

$$\begin{array}{lll}(s;t).\tau & \equiv & (\#\tau = \infty \wedge s.\tau) \ \vee \\ & & (\exists n :\ n < \#\tau :\ s.(\tau \uparrow n{+}1) \wedge t.(\tau \downarrow n))\end{array} \tag{5}$$

where $\#, \uparrow$ *and* $\downarrow$ *denote sequence operations* 'length', 'take' *and* 'drop', *respectively.*

[1] In this paper, we use $f.x$ to denote the result of applying a function f to argument x. Function application is left-associative, so $f.x.y$ is the same as $(f.x).y$.

Intuitively, the sequential composition of s and t can be understood as a program specification that requires s to be run first and then t to be run as soon as s terminates, if it does. More specifically, a computation τ satisfies $s;t$ if either it is an infinite computation satisfying s, or τ can be broken into a finite prefix $\tau \uparrow n{+}1$ and a suffix $\tau \downarrow n$ sharing state $\tau.n$ such that the prefix satisfies s and the suffix satisfies t.

In the course of reasoning using computation calculus, we make use of the distinction between infinite ("eternal") and finite computations. Two constants $\mathbf{E}, \mathbf{F} \in \mathit{CPred}$ have been defined for this purpose.

Definition 3 (Eternal and Finite Computations) *For any predicate s,*

$$\mathbf{E} \;=\; \mathbf{true};\mathbf{false} \tag{6}$$
$$\mathbf{F} \;=\; \neg\mathbf{E} \tag{7}$$
$$s \textit{ is eternal} \;\equiv\; [\, s \;\Rightarrow\; \mathbf{E}\,] \tag{8}$$
$$s \textit{ is finite} \;\equiv\; [\, s \;\Rightarrow\; \mathbf{F}\,] \tag{9}$$

An important property related to $\mathbf{E}$ (from [7]) is that for any predicate s, we have

$$s;\mathbf{false} \;=\; s \wedge \mathbf{E}\,. \tag{10}$$

Given $\mathbf{F}$, the temporal "eventually" operator (i.e., $\Diamond$) can be formulated as $\mathbf{F};s$. The "always" operator $\mathbf{G}$ is defined as the dual of the "eventually" operator.

Definition 4 (Always Operator) *For any predicate s,*

$$\mathbf{G}s \;=\; \neg(\mathbf{F};\neg s)\,. \tag{11}$$

Important properties of $\mathbf{G}$ are that it is *strengthening* (12), *monotonic* (13), and it *distributes over conjunction* (14). For any predicates s and t, we have:

$$[\,\mathbf{G}s \;\Rightarrow\; s\,], \tag{12}$$
$$[\,s \Rightarrow t\,] \;\Rightarrow\; [\,\mathbf{G}s \Rightarrow \mathbf{G}t\,], \tag{13}$$
$$\mathbf{G}(s \wedge t) \;=\; \mathbf{G}s \wedge \mathbf{G}t\,. \tag{14}$$

A useful technique that is frequently applied is to strip all the outer $\mathbf{G}$ in some proofs, as illustrated in the following example. For any predicates s, t, and u, we have

$$[\,s \wedge t \;\Rightarrow\; u\,] \;\Rightarrow\; [\,\mathbf{G}s \;\Rightarrow\; (\mathbf{G}t \Rightarrow \mathbf{G}u)\,]\,. \tag{15}$$

The proof of (15) is as follows.

$$\begin{array}{ll} & [\,\mathbf{G}s \;\Rightarrow\; (\mathbf{G}t \Rightarrow \mathbf{G}u)\,] \\ = & \quad \{\ \text{shunting}\ \} \\ & [\,\mathbf{G}s \wedge \mathbf{G}t \;\Rightarrow\; \mathbf{G}u\,] \\ = & \quad \{\ \mathbf{G}\text{ distributes through }\wedge\ (14)\ \} \\ & [\,\mathbf{G}(s \wedge t) \;\Rightarrow\; \mathbf{G}u\,] \\ \Leftarrow & \quad \{\ \text{monotonicity (13)}\ \} \\ & [\,s \wedge t \;\Rightarrow\; u\,] \end{array}$$

According to (15), whenever we need to prove a formula of the form $[\,\mathbf{G}s \;\Rightarrow\; (\mathbf{G}t \Rightarrow \mathbf{G}u)\,]$, we can reformulate it to strip the outer $\mathbf{G}$'s and manipulate s,t and u on their own to simplify the proof.

Definition 5 (Persistence) *For any predicate s,*

$$s \textit{ is persistent} \;\equiv\; s = \mathbf{G}s\,. \tag{16}$$

A persistent predicate describes some repetitive mosaic. If a persistent s can be used to describe a computation τ, s can also be used to describe every suffix of τ. We borrow the following facts related to the notion of persistence from [7]. For all predicates s, t, and persistent u, we have

$$\mathbf{G}s \text{ is persistent, and} \tag{17}$$
$$[\,u \;\Rightarrow\; (s;t \;\equiv\; s;(t \wedge u))\,]\,. \tag{18}$$

Consider some computation predicate r where $\mathbf{G}r$ holds, (17) ensures that $\mathbf{G}r$ is persistent, and (18) (together with (12)) enables us to insert r after any sub-computation in a series of sequential compositions. This is particularly useful when r is an invariant, i.e., $r = p;\mathbf{true}$, where p is a *state predicate* as defined in the subsequent.

2.2 State Predicates

A constant $\mathbb{1}$ is defined as the (left- and right-) neutral element for sequential composition.

Definition 6 (Constant $\mathbb{1}$) *For any computation τ,*

$$\mathbb{1}.\tau \;\equiv\; \#\tau = 1 \tag{19}$$

An important property of $\mathbb{1}$ is that it is finite, i.e.,

$$[\,\mathbb{1} \Rightarrow \mathbf{F}\,]\,. \tag{20}$$

In fact, $\mathbb{1}$ is the characteristic predicate of the state space. Moreover, we choose not to distinguish between a single state and the singleton computation consisting of that state, which allows us to identify predicates of one state with the predicates that hold only for singleton computations. Let us denote the set of state predicates by *SPred*.

Definition 7 (State Predicate) *For any predicate p,*

$$p \in \mathit{SPred} \;\equiv\; [\,p \Rightarrow \mathbb{1}\,]\,. \tag{21}$$

A consequence of this definition is that *SPred* is also a complete Boolean lattice with the order $[\,\_ \Rightarrow \_\,]$, with $\mathbb{1}$ and **false** being the "top" and "bottom" elements. It inherits all the lattice operators that it is closed under: conjunction, disjunction, and existential quantification. The other lattice operations, i.e., negation and universal quantification, are defined by restricting the corresponding operators on *CPred* to state predicates. We only use state predicate negation in this paper.

Definition 8 (State predicate negation $\sim$) *For any state predicate p,*

$$\sim p \quad = \quad \neg p \wedge \mathbb{1} \,. \tag{22}$$

For a state predicate p, the set of computations with the initial state satisfying p is captured by $p;\mathbf{true}$: the weakest such predicate. A special notation $\bullet : \mathit{SPred} \to \mathit{CPred}$ is introduced to denote this predicate.

Definition 9 (Initially Operator $\bullet$) *For any state predicate p,*

$$\bullet p \quad = \quad p;\mathbf{true} \,. \tag{23}$$

This entails the validity of the following rule, which we will use anonymously in the rest of the paper: for p, q two *state predicates*,

$$p;q \quad = \quad p \wedge q \,. \tag{24}$$

Another common rule related to state predicate is the *state restriction* rule allowing to trade $\wedge$ and $\bullet$ for ; and vice versa. Given any predicate s and any state predicate p, we have

$$p;s \quad = \quad s \wedge \bullet p \,. \tag{25}$$

2.3 Atomic Actions

An important operator in LTL is the "next-time operator". This is captured in computation calculus by the notion of atomic computations: computations of length 2. A constant $\mathbf{X} \in \mathit{CPred}$ is defined for this purpose.

Definition 10 (Atomic Actions) *For any computation τ and predicate a,*

$$\mathbf{X}.\tau \quad \equiv \quad \#\tau = 2 \tag{26}$$

$$a \textit{ is an atomic action} \quad \equiv \quad [\, a \Rightarrow \mathbf{X} \,] \tag{27}$$

Given the above definition, the "next" operator can be expressed as $\mathbf{X};s$ for arbitrary computation s. An important property for $\mathbf{X}$ is that it is finite, i.e.,

$$[\, \mathbf{X} \Rightarrow \mathbf{F} \,] \,. \tag{28}$$

3 The Unit-B Method

This section presents our contribution: the Unit-B method. It is inspired by Event-B [2] and UNITY [4].

Similar to Event-B, Unit-B is aimed at the design of software systems by stepwise refinement, where each step is verified via the application of correctness preserving refinement rules. It differs from Event-B by its capability for reasoning about progress properties and by its refinement order which preserves liveness properties. It also differs from UNITY by unifying the notions of programs and specifications, allowing stepwise refinement of programs from abstract models.

- In Section 3.1, we briefly review the syntax of Unit-B models (which has been informally introduced earlier in Section 1.1.2).
- In Section 3.2, we describe the semantics of a Unit-B model M. We characterize the set of executions of M by a computation predicate $ex.\mathsf{M}$ which is the conjunction of a safety and a liveness component. We provide proof rules for invariant preservation and *unless properties*. We also show the that the proof rules are sound with respect to the semantics.
- In Section 3.3, we provide proof rules for progress properties and prove their soundness.
- In Section 3.4, we provide refinement rules and prove their soundness with respect to the semantics.

A more extensive discussion of the soundness of the proof rules of Unit-B is presented in [11].

3.1 Syntax

Similar to Event-B, a Unit-B system is modelled by a transition system, where the state space is captured by variables v and the transitions are modelled by guarded events. Furthermore, Unit-B has additional assumptions on how the events should be scheduled. Using an Event-B-similar syntax, a Unit-B event has the following form:

$$\begin{array}{l} \mathsf{e}\,[i] \mathrel{\widehat{=}} \ \mathsf{during}\ c.i.v\ \ \mathsf{upon}\ f.i.v \\ \qquad\quad \mathsf{when}\ g.i.v\ \mathsf{then}\ s.i.v.v'\ \ \mathsf{end} \end{array} \tag{29}$$

where i are the event's *indices*, g is the event's *guard*, c is the event's *coarse schedule*, f is the event's *fine schedule*, and s is the event's *action* changing state variables v. The action is usually made up of several *assignments*, either deterministic ($:=$) or non-deterministic ($:|$ or $:\in$). An event e with indices i stands for multiple events. Each corresponds to several non-indexed events $\mathsf{e}.i$, one for each possible value of the indices i. Here g, c, f are state predicates. An event e is said to be enabled when its guard g holds. The scheduling assumption of the event is specified by c and f as follows: if c *holds continually* and f *becomes true infinitely often* then event e is carried out infinitely often. An event without any scheduling assumption will have its coarse schedule c equal to **false**. An event having only the coarse schedule c will have the fine schedule to be $\mathbb{1}$. Vice versa, an event having only the fine schedule f will have the coarse schedule to be $\mathbb{1}$.

In addition to the variables and the events, a model has an initialisation state predicate *init* constraining the initial value of the state variables. All computations of a model start from a state satisfying the initialisation and are such that, at every step, either one of its enabled events occurs or the state is unchanged, and each computation satisfies the scheduling assumptions of all events.

3.2 Semantics

In the following, we use computation calculus to give the formal semantics of Unit-B models. Let M be a Unit-B model containing a set of events of the form (29) and an initialisation predicate *init*. Since the action of the event can be described by a before-after predicate $s.i.v.v'$, it corresponds to an atomic action

$$\mathbf{S}.i \quad = \quad (\forall x :: \bullet(x = v) \;\Rightarrow\; \mathbf{X}; s.i.x.v) \;. \tag{30}$$

In the above, the quantified variable x is introduced to capture the before value of v, hence allows us to relate the pre-state and the post-state using the state predicate $s.i.x.v$ applied to the post-state (as indicated by $\mathbf{X};\_$). Given that an event $\mathsf{e}.i$ can only be carried out when it is enabled, we formulate the effect of each event execution as follows:

$$act.(\mathsf{e}.i) \quad = \quad g.i \;;\; \mathbf{S}.i \;. \tag{31}$$

The semantics of M is given by a computation predicate $ex.\mathsf{M}$ which is a conjunction of a "safety part" $saf.\mathsf{M}$ and a "liveness part" $live.\mathsf{M}$ (both to be defined later), i.e.,

$$[\, ex.\mathsf{M} \quad \equiv \quad saf.\mathsf{M} \;\wedge\; live.\mathsf{M} \,] \;. \tag{32}$$

Definition 11 *A property s is satisfied by M, denoted $\mathsf{M} \models s$, if the property is implied by $ex.\mathsf{M}$.*

$$\mathsf{M} \;\models\; s \quad \textit{if and only if} \quad [\, ex.\mathsf{M} \;\Rightarrow\; s \,] \;. \tag{33}$$

We use $\mathsf{M} \vdash s$ to denote that $\mathsf{M} \models s$ *is provable*.

Properties of Unit-B models are captured by two types of properties: *safety* and *progress* (liveness).

3.2.1 Safety

Below, we define the general form of one step of execution of model M, i.e., $step.\mathsf{M}$, and the *safety* constraints $saf.\mathsf{M}$ on its complete computations.

$$[\, step.\mathsf{M} \quad \equiv \quad (\exists \mathsf{e}, i : \mathsf{e}.i \in \mathsf{M} : act.(\mathsf{e}.i)) \vee \mathsf{Skip} \,] \tag{34}$$

$$[\, saf.\mathsf{M} \quad \equiv \quad \bullet init \wedge \mathbf{G}\,(step.\mathsf{M}\,;\, \mathbf{true}) \,] \tag{35}$$

where Skip is a special unscheduled event that is a part of every model. Its guard is true and its effect is to leave all the variables of model unchanged. Since the variables of the current model may be only one of the components of the state space — the other components being the variables of the models that may refine the current one — Skip makes no commitment about the final value of those components; that is to say that they will be changed non-deterministically without constraints.

Safety properties of the model are captured by *invariance* properties (also called *invariants*) and by *unless* properties.

Invariance properties An invariant $I.v$ is a state-property that holds at every reachable state of the model. If $I.v$ is an invariant of M, in all executions of M, $I.v$ holds forever:

$$[\, ex.\mathsf{M} \quad \Rightarrow \quad \mathbf{G}\,\bullet I \,] \;. \tag{36}$$

In particular, we rely solely on the safety part of the model to prove invariance properties, i.e., we prove $[\, saf.\mathsf{M} \Rightarrow \mathbf{G}\,\bullet I \,]$. This leads to the well-known invariance principle.

$$\frac{\begin{array}{ll} & \vdash \; init.v \;\Rightarrow\; I.v \\ & \vdash \; I.v \wedge g.i.v \wedge s.i.v.v' \;\Rightarrow\; I.v' \quad \text{(for all event } \mathsf{e}.i\text{)} \\ \mathsf{M} & \vdash \; \mathbf{G}\,\bullet I \end{array}}{} \tag{INV}$$

Invariance properties are important for reasoning about the correctness of the models since they give an (over-)approximation of the set of reachable states. This makes it possible to use invariance properties as additional assumptions in proofs for other properties (often as a consequence of applying (17) and (18)). For example, we can propagate a state predicate I to the middle of a sequential composition $s;t$ as follows: under the assumption that I holds forever, i.e., $\mathbf{G}\,\bullet I$, either because it is an invariant or for other reasons, for any predicates s and t, we have

$$s;t \;=\; s;I;t \tag{37}$$

The proof of (37) is as follows.

$$\begin{array}{ll} & s;t \\ = & \quad \{\; \mathbf{G}\,\bullet I \text{ (persistent) with persistence rule (18)} \;\} \\ & s;(t \wedge \mathbf{G}\,\bullet I) \\ = & \quad \{\; \mathbf{G} \text{ is strengthening (12)} \;\} \\ & s;(t \wedge \bullet I \wedge \mathbf{G}\,\bullet I) \\ = & \quad \{\; \mathbf{G}\,\bullet I \text{ (persistent) with persistence rule (18)} \;\} \\ & s;(t \wedge \bullet I) \\ = & \quad \{\; \text{state restriction (25)} \;\} \\ & s;I;t \end{array}$$

In the subsequent, we assume that model M has an invariant $I.v$.

Unless properties The other important class of safety properties is defined by the *unless* operator **un**.

Definition 12 (un operator) *For any state predicates p, q,*

$$[\, (p \;\mathbf{un}\; q) \quad \equiv \quad \mathbf{G}\,(\bullet p \;\Rightarrow\; (\mathbf{G}\,\bullet p)\,;(\mathbb{1} \vee \mathbf{X})\,;\bullet q) \,] \tag{38}$$

Informally, $p\,\mathbf{un}\,q$ is a safety property stating that if condition p holds then it will hold continuously unless q becomes true. The formula $(\mathbb{1} \vee \mathbf{X})$ is used in (38) to allow the last state where p holds and the state where q first holds to either be the same state or to immediately follow one another.

The following theorem is used for proving that a Unit-B model satisfies an unless property.

Theorem 1 (Unless rule) *Consider a model* M *with invariant* I *and an unless property* $p.v\,\mathbf{un}\,q.v$. *We have*

$$[\, ex.\mathsf{M} \;\Rightarrow\; p\,\mathbf{un}\,q \,]$$

if for every event e *and index value* i *with* $\mathsf{e}.i \in \mathsf{M}$,

$$[\, (I \wedge p \wedge {\sim}q);act.(\mathsf{e}.i) \;\Rightarrow\; \mathbf{X};(p \vee q) \,]\,. \tag{39}$$

Proof (Sketch) Condition (39) ensures that every event $\mathsf{e}.i$ of M either maintains p or establishes q. By induction, we can see that the only way for p to become false after a state where it was true is that either q becomes true or that it was already true. The full proof can be found in [11, Section 2.0.1] □

It follows from Theorem 1 that the following proof rule can be used to prove unless properties.

$$\frac{\begin{array}{lll} & p.v \wedge \neg q.v \wedge I.v \wedge & \\ \vdash & g.i.v \wedge s.i.v.v' & \\ \Rightarrow & p.v' \vee q.v' & \text{(for all event } \mathsf{e}.i\text{)} \end{array}}{\mathsf{M} \;\vdash\; p\ \mathbf{un}\ q} \tag{UN}$$

The antecedent of (UN) has the interesting peculiarity that it does not include either the fine or the coarse schedule of event e.

3.2.2 Liveness

For each event of the form (29), its schedule $sched.(\mathsf{e}.i)$ is formulated as follows, where c and f are the event's coarse and fine schedule, respectively:

$$\left[\, sched.(\mathsf{e}.i) \;\equiv\; \mathbf{G}\left(\begin{array}{l} \mathbf{G}\bullet c.i \wedge \mathbf{GF};\bullet f.i \\ \Rightarrow \\ \mathbf{F};f.i;act.(\mathsf{e}.i);\mathbf{true} \end{array}\right) \,\right]\,. \tag{SCH}$$

Intuitively, (SCH) states that if the coarse schedule c holds *continually*, i.e., $\mathbf{G}\bullet c$ and the fine schedule f becomes true *infinitely often*, i.e., $\mathbf{GF};\bullet f$, then eventually $\mathsf{e}.i$ occurs at a point where f holds, i.e., $\mathbf{F};f;act.(\mathsf{e}.i)$. To ensure that the event $\mathsf{e}.i$ only occurs when its guard $g.i$ holds, we require the following *feasibility* condition:

$$I.v \wedge c.i.v \wedge f.i.v \;\Rightarrow\; g.i.v \tag{SCH\_FIS}$$

In absence of this condition, the (coarse or fine) schedule may be continuously in contradiction with the guard: while the scheduling constraint (SCH) states that all valid computation will include occurrences of the event, the safety constraint (35) states that, under the same conditions, the event will not happen. It follows that no traces satisfy the two constraints and the system cannot be implemented.

Our coarse and fine schedules are a generalisation of the standard weak-fairness and strong-fairness assumptions. The standard *weak-fairness* assumption for event $\mathsf{e}.i$ (stating that if $\mathsf{e}.i$ is enabled continually then eventually it will be taken) can be formulated by using $c = g$ and $f = \mathbb{1}$. Similarly, the standard *strong-fairness* assumption for $\mathsf{e}.i$ (stating that if $\mathsf{e}.i$ is enabled infinitely often then eventually it will be taken) can be formulated by using $c = \mathbb{1}$ and $f = g$.

$$\begin{array}{lll} [\,\mathrm{wf}.(\mathsf{e}.i) & \equiv & \mathbf{G}\,(\mathbf{G}\bullet g.i \;\Rightarrow\; \mathbf{F};act.(\mathsf{e}.i);\mathbf{true})\,] \\ [\,\mathrm{sf}.(\mathsf{e}.i) & \equiv & \mathbf{G}\,(\mathbf{GF};\bullet g.i \;\Rightarrow\; \mathbf{F};act.(\mathsf{e}.i);\mathbf{true})\,] \end{array}$$

Instead of categorizing Unit-B events between weakly fair and strongly fair, our generalization allows us to have a little of both in every event. Strong fairness is often a nice abstraction of scheduling magic happening under the hood but it is necessary to refine it away in order to implement it. Our generalization facilitates this by making the transition between strong fairness to weak fairness smoother.

In Section 4.4.1, we provide a methodological comparison between weak and strong fairness on one hand and coarse and fine schedules on the other hand in the context of the main example. Furthermore, in Section 4.4 we discuss the heuristics justifying the choice of coarse and fine schedules of events.

The liveness part of the model is the conjunction of the schedules for its events, i.e.,

$$[\, live.\mathsf{M} \;\equiv\; (\forall\, \mathsf{e}, i :\, \mathsf{e}.i \in \mathsf{M} :\, sched.(\mathsf{e}.i))\,] \tag{40}$$

The summary of the Unit-B modelling notation is showed in Figure 3.

$$\mathsf{e}\,[i] \mathrel{\widehat{=}} \begin{array}{l} \mathsf{during}\ c.i.v\ \ \mathsf{upon}\ f.i.v \\ \mathsf{when}\ g.i.v\ \mathsf{then}\ s.i.v.v'\ \ \mathsf{end} \end{array}$$

$$\begin{array}{lll} [\, ex.\mathsf{M} & \equiv & saf.\mathsf{M} \wedge live.\mathsf{M}\,] \\ [\, saf.\mathsf{M} & \equiv & \bullet init \wedge \mathbf{G}\,(step.\mathsf{M};\mathbf{true})\,] \\ [\, step.\mathsf{M} & \equiv & (\exists\, \mathsf{e}, i :\, \mathsf{e}.i \in \mathsf{M} :\, act.(\mathsf{e}.i)) \vee \mathsf{Skip}\,] \\ [\, act.(\mathsf{e}.i) & \equiv & g.i.;\mathbf{S}.i\,] \\ [\, \mathbf{S}.i.t & \equiv & (\forall x ::\, \bullet(x = v) \;\Rightarrow\; \mathbf{X};s.i.x.v)\,] \\ [\, live.\mathsf{M} & \equiv & (\forall\, \mathsf{e}, i :\, \mathsf{e}.i \in \mathsf{M} :\, sched.(\mathsf{e}.i))\,] \\ [\, sched.(\mathsf{e}.i) & \equiv & \mathbf{G}\,(\mathbf{G}\bullet c.i \wedge \mathbf{GF};\bullet f.i \;\Rightarrow\; \mathbf{F};f.i;act.(\mathsf{e}.i);\mathbf{true})\,] \\ \mathsf{M} \models s & \text{iff} & [\, ex.\mathsf{M} \;\Rightarrow\; s\,] \end{array}$$

Fig. 3 Summary of Unit-B

3.3 Progress Properties

Progress properties are of the form $p \leadsto q$, where $\leadsto$ is the leads-to operator. They state that every state satisfying predicate p is eventually followed by by a state satisfying q.

Definition 13 ($\leadsto$ operator) *For any state predicates p, q,*

$$[\,(p \leadsto q) \;\equiv\; \mathbf{G}\,(\bullet p \Rightarrow \mathbf{F};\bullet q)\,] \tag{41}$$

In the case where p and q contain free variables, i.e., variables not belonging to the state space, $p \leadsto q$ is understood implicitly as

$$(\forall x ::\; p \leadsto q)$$

with x the tuple of all the free variables appearing in either p or q. The same principle is applied to unless, transient and falsifies properties, the last two are introduced later in this section.

A special kind of progress properties is captured by the *transient operator*. Transient property $\mathbf{tr}\,p$ states that whenever predicate p holds, it is eventually falsified. Transient properties are especially useful for creating a bridge between leads- to properties and the events that effect them. That bridge is completed by the **falsifies** operator which we introduce later in this section.

Definition 14 (tr operator) *For any state predicate p,*

$$\mathit{tr}\,p \;=\; p \leadsto \sim p \;=\; \mathbb{1} \leadsto \sim p \;=\; \mathbf{G}\mathbf{F};\bullet \sim p \tag{42}$$

The properties of $\leadsto$ and **tr** that we will use in this paper are as follows. For any state predicates p, q, and r, we have:

$$[\,\mathbf{G}\,\bullet(p \Rightarrow q) \;\Rightarrow\; (p \leadsto q)\,] \qquad \text{(Implication)}$$
$$[\,(p \leadsto q) \wedge (q \leadsto r) \;\Rightarrow\; (p \leadsto r)\,] \qquad \text{(Transitivity)}$$
$$[\,(p \leadsto q) \;\equiv\; (p \wedge \sim q \leadsto q)\,] \qquad \text{(Split-Off-Skip)}$$
$$[\,(p \;\mathbf{un}\; q) \wedge (\mathbf{tr}\,p \wedge \sim q) \;\Rightarrow\; (p \leadsto q)\,] \qquad \text{(Ensure)}$$
$$\left[\begin{array}{l} (p \wedge v = M \;\leadsto\; (p \wedge v < M) \vee q) \\ \Rightarrow\; (p \leadsto q) \end{array}\right] \qquad \text{(Induction)}$$
$$\left[\begin{array}{l} (p \leadsto q) \wedge (r \;\mathbf{un}\; b) \\ \Rightarrow\; (p \wedge r \leadsto (q \wedge r) \vee b) \end{array}\right] \qquad \text{(PSP)}$$

Above, in the induction rule, M is a free variable and v is the variant, an expression involving some state variables. The name of the (PSP) rule stands for *Progress, Safety, Progress*. Except for (Split-Off-Skip), the above rules are taken from [4].

We prove progress properties by relating them to the events of the model with **falsifies** properties. We can establish $\mathbf{tr}\,p$ by choosing an event e of the model and proving e **falsifies** p, i.e., if p holds continually e is eventually taken and whenever e is executed in a state where p holds it falsifies p.

Definition 15 (falsifies operator) *For any state predicate p and any event as follows.:*

$$\begin{array}{l} \mathsf{e}\,[i] \mathrel{\widehat{=}} \text{during } c.i.v \text{ upon } f.i.v \\ \qquad \text{when } g.i.v \text{ then } s.i.v.v' \text{ end} \end{array}$$

Event e *with (actual) index* i falsifies *property p (denoted as* e.i ***falsifies*** *p) if under condition p,* e.i *negates p in one step* (NEG)*, the coarse schedule c is enabled* (C_EN)*, and the fine schedule f is eventually enabled* (F_EN)*.*

$$[\,\mathsf{e}.i \;\textbf{\textit{falsifies}}\; p \;\equiv\; (\text{NEG}) \wedge (\text{C\_EN}) \wedge (\text{F\_EN})\,]\,, \tag{43}$$

where

$$\mathbf{G}\;\; ((p \wedge c \wedge f);\; act.(\mathsf{e}.i)\;;\; true \;\Rightarrow\; \mathbf{X};\bullet \sim p)\;, \qquad \text{(NEG)}$$
$$\mathbf{G}\;\; \bullet(p \;\Rightarrow\; c)\;, \qquad \text{(C\_EN)}$$
$$p \wedge c \;\;\leadsto\;\; f\;. \qquad \text{(F\_EN)}$$

Property e.i **falsifies** p states that, when state predicate p holds, if it is not falsified by events other than e.i, e.i will eventually occur and falsify p.

Given the definition of **falsifies** , we have the following proof rule (taking into account the invariant $I.v$).

$$\begin{array}{ll} & \vdash\; I.v \wedge p.v \;\wedge\; c.i.v \wedge f.i.v \wedge s.i.v.v' \;\Rightarrow\; \neg p.v' \\ & \vdash\; I.v \wedge p.v \Rightarrow c.v \\ \mathsf{M} & \vdash\; p.v \wedge c.i.v \;\leadsto\; f.i.v \\ \hline \mathsf{M} & \vdash\; \mathsf{e}.i \;\textbf{falsifies}\; p.v \end{array} \tag{FLS}$$

The **falsifies** properties are the main tool for linking the model and the progress properties in Unit-B. The attractiveness of such properties is that we can *implement* them using a single event. In the case of events without a fine schedule (i.e., f is $\mathbb{1}$), which is the most common one, the last condition (F_EN) becomes trivial and can be omitted.

Theorem 2 (Transient rule) *Consider state predicate p and a model* M *contains event* e*.*

$$\begin{array}{l} \mathsf{e}\,[i] \mathrel{\widehat{=}} \text{during } c.i.v \text{ upon } f.i.v \\ \qquad \text{when } g.i.v \text{ then } s.i.v.v' \text{ end} \end{array}$$

Given an (actual) index i, we have

$$[\,ex.\mathsf{M} \;\Rightarrow\; \mathit{tr}\,p\,] \;\; \textit{if} \;\; [\,ex.\mathsf{M} \;\Rightarrow\; \mathsf{e}.i \;\textbf{\textit{falsifies}}\; p\,]\,.$$

Proof Unfolding the definitions of **tr** and **falsifies** , we prove $\mathbf{G}\mathbf{F};\bullet \sim p$ under the assumptions (NEG), (C_EN) and (F_EN). Moreover, since e is an event in M, we have $[\,ex.\mathsf{M} \Rightarrow sched.(\mathsf{e}.i)\,]$. Therefore we have $sched.(\mathsf{e}.i)$ as an additional assumption.

Dropping the outer **G** in the goal and in the assumptions (similar to (15)), our goal becomes $\mathbf{F};\bullet \sim p$. Additionally, since $[\,\neg s \Rightarrow s \;\equiv\; s\,]$ for any computation predicate s, we discharge our obligation by strengthening $\mathbf{F}\;;\bullet \sim p$ to its negation, $\mathbf{G}\,\bullet p$.

$$
\begin{array}{ll}
& \mathbf{F};\bullet\sim p \\
\Leftarrow & \quad \{\ [\,\mathbf{F};\mathbf{X}\Rightarrow\mathbf{F}\,]\text{, aiming for (NEG)}\ \} \\
& \mathbf{F};\mathbf{X};\bullet\sim p \\
\Leftarrow & \quad \{\ \text{(NEG)}\ \} \\
& \mathbf{F};(p\wedge c\wedge f);act.(\mathsf{e}.i);\mathbf{true} \\
\Leftarrow & \quad \{\ \text{property of } \mathbf{G}\ (37)\ \} \\
& \mathbf{F};f;act.(\mathsf{e}.i);\mathbf{true}\ \wedge\ \mathbf{G}\bullet c\ \wedge\ \mathbf{G}\bullet p \\
\Leftarrow & \quad \{\ sched.(\mathsf{e}.i)\text{ and definition (SCH)}\ \} \\
& \mathbf{GF};\bullet f\ \wedge\ \mathbf{G}\bullet c\ \wedge\ \mathbf{G}\bullet p \\
= & \quad \{\ \mathbf{G}\text{ distributes through }\wedge\ (14)\} \\
& \mathbf{G}\,(\ \mathbf{F};\bullet f\ \wedge\ \bullet(c\wedge p)\ ) \\
= & \quad \{\ \text{(F\_EN)}\ \} \\
& \mathbf{G}\bullet(c\wedge p) \\
= & \quad \{\ \text{(C\_EN)}\ \} \\
& \mathbf{G}\bullet p
\end{array}
$$

□

Theorem 2 corresponds to the following proof rule.

$$\frac{\mathsf{M}\ \vdash\ \mathsf{e}.i\ \textbf{falsifies}\ p}{\mathsf{M}\ \vdash\ \textbf{tr}\ p} \qquad \text{(TRS)}$$

3.4 Refinement

In this section, we develop rules for refining Unit-B models such that safety and liveness properties are preserved. Consider models M and N. Refinement, denoted by $\mathsf{M}\sqsubseteq\mathsf{N}$ is defined by:

$$\mathsf{M}\sqsubseteq\mathsf{N}\ \equiv\ [\,ex.\mathsf{M}\ \Leftarrow\ ex.\mathsf{N}\,]\,. \qquad \text{(REF)}$$

We call M the abstract model and N the concrete model. As a result of this definition, any property of M is also satisfied by N. Similarly to Event-B, refinement is considered in Unit-B on a per event basis. Each abstract event e_a is *refined* by a concrete event e_c.

$$\mathsf{e}_a\,[i]\ \widehat{=}\ \begin{array}{l}\textsf{during}\ c_a.i.v\ \ \textsf{upon}\ f_a.i.v\\ \textsf{when}\ g_a.i.v\ \textsf{then}\ s.i.v.v'\ \ \textsf{end}\end{array} \qquad (44)$$

$$\mathsf{e}_c\,[j]\ \widehat{=}\ \begin{array}{l}\textsf{during}\ c_c.j.v\ \ \textsf{upon}\ f_c.j.v\\ \textsf{when}\ g_c.j.v\ \textsf{then}\ s.j.v.v'\ \ \textsf{end}\end{array} \qquad (45)$$

We say that e_c refines e_a if

$$\left(\forall j::\ \left(\begin{array}{l}\exists i::[\,ex.\mathsf{N}\ \Rightarrow\\ \quad (act.(\mathsf{e}_c.j)\Rightarrow act.(\mathsf{e}_a.i))\,]\end{array}\right)\right) \qquad \text{(EVT\_SAFE)}$$

$$\left(\forall i::\ \left(\begin{array}{l}\exists j::[\,ex.\mathsf{N}\ \Rightarrow\\ \quad (sched.(\mathsf{e}_c.j)\Rightarrow sched.(\mathsf{e}_a.i))\,]\end{array}\right)\right) \qquad \text{(EVT\_LIVE)}$$

The proof that N refines M (i.e., (REF)) given conditions (EVT_SAFE) and (EVT_LIVE) is left out. A special case of event refinement is when the concrete event e_c is a new event. In this case, we prove that e_c is the refinement of the special Skip event which is unscheduled and does not change any variables of the abstract model.

Condition (EVT_SAFE) leads to similar proof obligations in Event-B such as *guard strengthening* and *simulation*. We focus here on expanding the condition (EVT_LIVE).

We consider two cases for event refinement: (1) the abstract and concrete events have the same indices, and (2) the indices are removed from the concrete event.

Theorem 3 (Retaining Events' Indices) *Consider events* e_a *and* e_c *as follows.*

$$\mathsf{e}_a\,[i]\ \widehat{=}\ \begin{array}{l}\textsf{during}\ c_a.i.v\ \ \textsf{upon}\ f_a.i.v\\ \textsf{when}\ g_a.i.v\ \textsf{then}\ s.i.v.v'\ \ \textsf{end}\end{array} \qquad (46)$$

$$\mathsf{e}_c\,[i]\ \widehat{=}\ \begin{array}{l}\textsf{during}\ c_c.i.v\ \ \textsf{upon}\ f_c.i.v\\ \textsf{when}\ g_c.i.v\ \textsf{then}\ s.i.v.v'\ \ \textsf{end}\end{array} \qquad (47)$$

Assume (EVT_SAFE) *have been proved for* e_a *and* e_c*, i.e.,*

$$act.(\mathsf{e}_c.i)\ \Rightarrow\ act.(\mathsf{e}_a.i)\,. \qquad (48)$$

Given

$$c_a\wedge f_a\ \rightsquigarrow\ c_c \qquad \text{(C\_FLW)}$$
$$c_c\ \ \textbf{\textit{un}}\ \ c_a \qquad \text{(C\_STB)}$$
$$c_a\wedge f_a\ \rightsquigarrow\ f_c \qquad \text{(F\_FLW)}$$
$$\mathbf{G}\bullet(c_c\wedge f_c\ \Rightarrow\ f_a) \qquad \text{(F\_STR)}$$

then

$$sched.(\mathsf{e}_c.i)\ \Rightarrow\ sched.(\mathsf{e}_a.i) \qquad (49)$$

Proof We first prove that the left-hand side of $sched.(\mathsf{e}_a.i)$, i.e., $\mathbf{G}\bullet c_a\wedge\mathbf{GF};\bullet f_a$ eventually leads to the left-hand side of $sched.(\mathsf{e}_c.i)$, i.e., $\mathbf{G}\bullet c_c\wedge\mathbf{GF};\bullet f_c$.

$$\mathbf{G}\,(\mathbf{G}\bullet c_a\wedge\mathbf{GF};\bullet f_a\ \ \Rightarrow\ \ \mathbf{F};(\mathbf{G}\bullet c_c\wedge\mathbf{GF};\bullet f_c)) \qquad (50)$$

Dropping the outer G from (50), we start the proof from $\mathbf{GF};\bullet f_a$ with assumption $\mathbf{G}\bullet c_a$.

$$
\begin{array}{ll}
& \mathbf{GF};\bullet f_a \\
\Rightarrow & \quad \{\ \mathbf{G}\bullet c_a\ \} \\
& \mathbf{GF};\bullet(c_a \wedge f_a) \\
\Rightarrow & \quad \{\ \text{(C\_FLW) and (F\_FLW)}\ \} \\
& \mathbf{GF};\bullet c_c \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ \text{(C\_STB) and definition of } \mathbf{un}\ (38)\ \} \\
& \mathbf{GF};(\mathbf{G}\bullet c_c);(\mathbb{1}\vee\mathbf{X});\bullet\neg c_a \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ \mathbf{G}\bullet c_a\ \} \\
& \mathbf{GF};(\mathbf{G}\bullet c_c);(\mathbb{1}\vee\mathbf{X});(\bullet\neg c_a\wedge\bullet c_a) \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ \text{contradiction}\ \} \\
& \mathbf{GF};(\mathbf{G}\bullet c_c);(\mathbb{1}\vee\mathbf{X});\mathbf{false} \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ (\mathbb{1}\vee\mathbf{X});\mathbf{false}=\mathbf{false}\ \} \\
& \mathbf{GF};(\mathbf{G}\bullet c_c);\mathbf{false} \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ \text{property of eternal computation (10)}\ \} \\
& \mathbf{GF};(\mathbf{G}\bullet c_c)\wedge\mathbf{E} \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ \text{weakening}\ \} \\
& \mathbf{GF};(\mathbf{G}\bullet c_c) \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ \mathbf{G}\ \text{is strengthening (12)}\ \} \\
& \mathbf{F};(\mathbf{G}\bullet c_c) \ \wedge\ \mathbf{GF};\bullet f_c \\
\Rightarrow & \quad \{\ \mathbf{GF};\bullet f_c\ \text{is persistent (17) and persistence rule (18)}\ \} \\
& \mathbf{F};(\mathbf{G}\bullet c_c \ \wedge\ \mathbf{GF};\bullet f_c)
\end{array}
$$

Finally, the proof of (49) is as follows. Expanding the definition of *sched*, we prove

$$\mathbf{G}(\mathbf{G}\bullet c_a\wedge\mathbf{GF};\bullet f_a \ \Rightarrow\ \mathbf{F};f_a;act.(\mathsf{e_a}.i)) \tag{51}$$

under the assumptions

$$\mathbf{G}(\mathbf{G}\bullet c_c\wedge\mathbf{GF};\bullet f_c \ \Rightarrow\ \mathbf{F};f_c;act.(\mathsf{e_c}.i)) \tag{52}$$

First, notice that we drop the outer $\mathbf{G}$ from (51), (52). and start the proof with the left-hand side of (51).

$$
\begin{array}{ll}
& \mathbf{G}\bullet c_a\wedge\mathbf{GF};\bullet f_a \\
\Rightarrow & \quad \{\ (50)\ \} \\
& \mathbf{F};(\mathbf{G}\bullet c_c\wedge\mathbf{GF};\bullet f_c)) \\
\Rightarrow & \quad \{\ (52)\ \} \\
& \mathbf{F};(\mathbf{G}\bullet c_c\wedge\mathbf{F};f_c;act.(\mathsf{e_c}.i)) \\
\Rightarrow & \quad \{\ \text{invariant property (37), and } \mathbf{F};\mathbf{F}=\mathbf{F}\} \\
& \mathbf{F};(c_c\wedge f_c);act.(\mathsf{e_c}.i) \\
\Rightarrow & \quad \{\ \text{(F\_STR)}\ \} \\
& \mathbf{F};f_a;act.(\mathsf{e_c}.i) \\
\Rightarrow & \quad \{\ (48)\ \} \\
& \mathbf{F};f_a;act.(\mathsf{e_a}.i)
\end{array}
$$

Theorem 3 leads to the following proof rule.

$$
\frac{\begin{array}{ll}
\mathsf{N} & \vdash\ c_a\wedge f_a \rightsquigarrow c_c \\
\mathsf{N} & \vdash\ c_c\ \mathbf{un}\ \sim c_a \\
\mathsf{N} & \vdash\ c_a\wedge f_a \rightsquigarrow f_c \\
 & \vdash\ I_c\wedge c_c\wedge f_c \Rightarrow f_a
\end{array}}{\mathsf{N}\ \vdash\ sched.(\mathsf{e_c}.i)\ \Rightarrow\ sched.(\mathsf{e_a}.i)} \tag{53}
$$

The following corollaries are direct consequences of Theorem 3, hence concerning events $\mathsf{e_a}$ and $\mathsf{e_c}$ as in (46) and (47). They illustrate different ways of refining event scheduling information: *weakening the coarse schedule*, *replacing the coarse schedule*, *strengthening the fine schedule*, and *removing the fine schedule*.

Corollary 1 (Coarse schedule weakening) *Given $f_c = f_a$, we have*

$$sched.(\mathsf{e_c}.t)\Rightarrow sched.(\mathsf{e}.t)$$

if

$$\mathbf{G}\bullet(c_a\Rightarrow c_c)\ . \tag{54}$$

Proof (Sketch) Given $f_c = f_a$, conditions (F_FLW) and (F_STR) of Theorem 3 are trivial. Conditions (C_FLW) and (C_STB) are direct consequences of (54).

Corollary 2 (Coarse schedule replacement) *Given $f_c = f_a$, we have*

$$sched.(\mathsf{e_c}.t)\Rightarrow sched.(\mathsf{e}.t)$$

if

$$c_a\wedge f_a \rightsquigarrow c_c \tag{C\_FLW}$$
$$c_c\ \mathbf{un}\ \sim c_a\ . \tag{C\_STB}$$

Proof (Sketch) Given $f_c = f_a$, conditions (F_FLW) and (F_STR) of Theorem 3 are trivial.

Corollary 3 (Fine schedule strengthening) *Given $c_c = c_a$, we have*

$$sched.(\mathsf{e_c}.i)\Rightarrow sched.(\mathsf{e_a}.i)$$

if

$$c_a\wedge f_a \rightsquigarrow f_c\ ,\ and \tag{F\_FLW}$$
$$\mathbf{G}\bullet(f_c\Rightarrow f_a)\ . \tag{F\_STR}$$

Proof (Sketch) Given $c_c = c_a$, conditions (C_FLW) and (C_STB) of Theorem 3 are trivial.

Corollary 4 (Fine schedule removal) *Given* $c_c = c_a$ *and* $f_c = \mathbb{1}$, *we have*

$$sched.(\mathsf{e_c}.i) \Rightarrow sched.(\mathsf{e_a}.i)$$

if

$$\mathbf{G} \bullet (c_a \Rightarrow f_a) \ . \tag{55}$$

Proof (Sketch) Given $c_c = c_a$, conditions (C_FLW) and (C_STB) of Theorem 3 are trivial. Given $f_c = \mathbb{1}$, condition (F_FLW) is trivial and condition (F_STR) is a direct consequent of (55).

A special case of event refinement allows to remove event indices as illustrated by the following theorem.

Theorem 4 (Events' indices removal) *Consider* $\mathsf{e_a}$ *as follows*

$$\mathsf{e_a}\ [i,j] \mathrel{\widehat{=}} \begin{array}{l} \textsf{during } i = E.j.v \wedge c.(i,j).v \ \textsf{ upon } f.(i,j).v \\ \textsf{when } g.(i,j).v \textsf{ then } s.(i,j).v.v' \ \textsf{ end} \end{array} \tag{56}$$

The following event $\mathsf{e_c}$ *is a refinement of* $\mathsf{e_a}$, *i.e., satisfying* (EVT_SAFE) *and* (EVT_LIVE).

$$\mathsf{e_c}\ [j] \mathrel{\widehat{=}} \begin{array}{l} \textsf{during } c.(E.j.v,\ j).v \ \textsf{ upon } f.(E.j.v,\ j).v \\ \textsf{when } g.(E.j.v,\ j).v \textsf{ then } s.(E.j.v,\ j).v.v' \ \textsf{ end} \end{array} \tag{57}$$

Proof For (EVT_SAFE), we use $E.j.v$ as the witness for the removing indices i, which leads to the proof obligation:

$$act(\mathsf{e_c}.j) \ \Rightarrow \ act(\mathsf{e_a}.(E.j.v, j)) \ .$$

For (EVT_LIVE), we note that in the case where $i \neq E.j.v$, the coarse schedule of $\mathsf{e_a}$ is **false**, hence $\mathsf{e_a}.i$ is unscheduled, hence (EVT_LIVE) is satisfied. Therefore, $sched.(\mathsf{e_c}.j) = sched(\mathsf{e_a}.(E.j.v, j))$ holds.

4 Example: A Signal Control System

We illustrate our method by applying it to design a system controlling trains at a station [12]. We first present some informal requirements of the system.

4.1 Requirements

The network at the station contains an *entry block*, several *platform blocks* and an *exiting block*, as seen in Figure 4. Trains arrive on the network at the entry block, then can move into one of the platform blocks before moving to the exiting block and leaving the network. In order to control the trains at the station, signals are positioned at the end of the entry block and each platform block. The train drivers are assumed to obey the signals. The signals are supposed to change from green to red automatically when a train passes by.

The most important properties of the system are that (1) there should be no collision between trains (SAF 1), and (2) each train in the network eventually leaves (FUN 2).

SAF 1 There is at most one train on each block

FUN 2 Each train in the network eventually leaves

ENV 3 The tracks are arranged according to Figure 4

FUN 4 Every train enters only through the *entry block*, then proceed to a *platform block* and move on to the *exit block* from where they leave the station.

EQP 5 A light signal is positioned after the entrance block and after each of the platforms.

ENV 6 Train drivers obey the light signals, i.e. when the signal is green, they advance and they stop when the signal is red.

Refinement strategy Our development consists of an initial model and five refinement steps. We summarize our refinement strategy for developing the signal control system as follows.

Init. model We abstractly model the trains in the network, focusing on FUN 2.

1st Ref. We introduce the topology of the network ENV 3 and FUN 4.

2nd Ref. We strengthen the model of the system, focusing on SAF 1.

3rd Ref. We introduce the signals and derive a specification for the controller that manages these signals EQP 5 and ENV 6.

4th Ref. We refine the controller's specification, in particular, scheduling the trains passing the station in a first-in-first-out manner.

5th Ref. We refine further the controller's specification so that it can be implemented in some programming language.

Notation Well-definedness [19] is an important issue when dealing with partial functions. However, when trying to make formulas well-defined, some overhead often has to be introduced which can make said formulas bulky. For example, if we need to express

$$f.x \leq g.y \ ,$$

with f, g two partial functions, the formula is only meaningful in the case where $x \in \mathrm{dom}.f \wedge y \in \mathrm{dom}.g$ and the formula above is therefore not necessarily well defined. This new formula

$$x \in \mathrm{dom}.f \ \wedge \ y \in \mathrm{dom}.g \ \wedge \ f.x \leq g.y$$

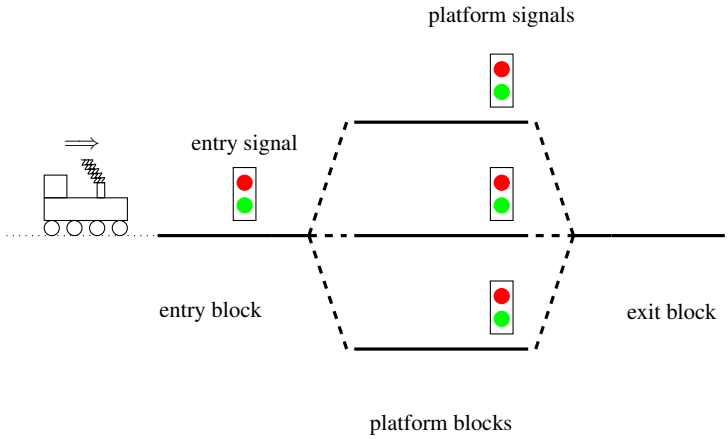

Fig. 4 A signal control system

is well-defined and is often a suitable substitute for $f.x \leq g.y$ but it is much longer and the subject matter, the ordering of f and g, constitute only a small fraction of the formula: the attention of the reader is mostly drawn to the technicality of well-definedness.

As a shorthand, we will use a new notation defined as $\langle P \rangle \triangleq \mathscr{D}(P) \wedge P$. In the previous example, we can express the property as

$$\langle f.x \leq g.y \rangle$$

which is false if $f.x$ or $g.y$ is ill-defined and has the expected truth value otherwise. It might be also handy to have a shorthand for $x \in \mathrm{dom}.f \wedge y \in \mathrm{dom}.g \;\Rightarrow\; f.x \leq g.y$ — which is true if $f.x$ or $g.y$ is ill-defined and has the normal truth value otherwise — but we won't need it in this paper and therefore refrain from defining a shorthand for it.

Logic In Sect. 3, we conducted the proofs of soundness of the refinement rules and the inference rules of temporal properties using computation calculus. In the example, we will conduct our reasoning using these inference rules and predicate calculus without reference to computation calculus. The purpose is to use the rules as a clear interface between the semantics of Unit-B and the reasoning about Unit-B models.

Proof Format The equational proof format has been used to advantage already in Section 2 and Section 3. There we use this format in rewriting an expression with value preserving rules or various order preserving rules (e.g. $\Rightarrow$, $\leq$), from an initial expression to a final expression. The style of manipulations has an algebraic flavour. In this section, we use equational proof format to manipulate sequents instead of normal expressions. While an expression has a value, a sequent is provable or not. The usual way of relating two sequents is the inference rule:

$$\frac{\Gamma, \alpha \vdash \phi}{\Gamma \vdash \psi}$$

When building a formal proof with them, the format becomes quickly unwieldy and unreadable. Instead, we use $\sqsubseteq$ to relate two sequents in equational proofs with the understanding that $\Gamma \vdash \psi \;\sqsubseteq\; \Gamma, \alpha \vdash \phi$ stands for the inference above, i.e. $\Gamma \vdash \psi$ has a proof if $\Gamma, \alpha \vdash \phi$ has a proof.

Sometimes, inferences rules have more than one premise. In such cases, in our calculations, either we keep the most important one as the main thread of reasoning and refer to the other ones in the hint of the step, or we list the ones we kept one above the other. The subsequent steps can apply to any one of them.

Naming Convention We adopt the following convention in naming the properties appearing in the subsequent development to indicate the type of the property (e.g., invariance, unless, or progress), the level of refinement, and the sequent number of the property. For example, inv0_1 is the 1st invariant of the initial model, while un2_1 and prg2_1 are the 1st unless property and the 1st progress property of the 2nd refinement, respectively.

Refinement Strategy The refinement strategy for our development is as follows.

Initial Model focuses on specifying and reasoning about the main progress requirement FUN 2.

First refinement introduces the topology of the train station ENV 3 and the movement of the trains through the station FUN 4.

Second Refinement incorporates the safety requirements of the system to prevent train collisions SAF 1.

Third Refinement adds the light signals to the model EQP 5 and the assumption that the train drivers always obey these light signals ENV 6.

Fourth Refinement realises a software controller for the signals in the station. In particular, the scheduling of the train passing through the station is performed using a queue.

Fifth Refinement simplifies the software controller by removing the index of the corresponding event.

4.2 Initial Model M_0— Arriving and Departing

In this initial model M_0, we use a carrier set TRN to denote the set of trains and a variable st (short for station) to denote the set of trains currently inside the station.

$$\text{variables}: \ st \qquad \text{invariants}: \ \text{inv0\_1}: \ st \subseteq TRN$$

Initially st is assigned the empty set $\varnothing$. At this abstract level, we have two events to model a train arriving at the station and a train leaving the station as follows:

$$\text{arrive}\ [t] \mathrel{\widehat{=}} \text{when}\ t \in TRN\ \text{then}\ st := st \cup \{t\}\ \text{end}$$
$$\text{depart}\ [t] \mathrel{\widehat{=}} \text{when}\ t \in TRN\ \text{then}\ st := st \setminus \{t\}\ \text{end}$$

Requirement FUN 2 can be specified as a progress property (with t implicitly quantified universally over the whole property):

$$t \in st \ \leadsto\ \neg t \in st\ . \qquad \text{(prg0\_1)}$$

We attempt to use event depart to implement prg0_1 as follows.

$$\begin{array}{ll} & M_0 \ \vdash\ t \in st \leadsto \neg t \in st \\ \sqsubseteq & \{\ \text{Transient definition (42)}\ \} \\ & M_0 \ \vdash\ \textbf{tr}\ t \in st \\ \sqsubseteq & \{\ \text{Transient rule (TRS) with depart.}t\ \} \\ & M_0 \ \vdash\ \text{depart.}t \ \ \textbf{falsifies}\ \ t \in st \\ \sqsubseteq & \{\ \text{Falsifies rule (FLS) with: (C\_EN\_1) and (NEG\_1)}\ \} \\ & \text{true} \end{array}$$

with:

$$t \in st \ \Rightarrow\ \textbf{false} \qquad \text{(C\_EN\_1)}$$
$$t \in st \wedge \textbf{false} \wedge st' = st \setminus \{t\} \ \Rightarrow\ \neg t \in st' \qquad \text{(NEG\_1)}$$

The proof obligation (NEG_1) is trivial. However, (C_EN_1) cannot be proved because the coarse schedule of depart is **false** (since depart is current unscheduled). We can remedy this situation by adding a coarse schedule to depart, which becomes as follows:

$$\text{depart}\ [t] \mathrel{\widehat{=}} \text{during}\ t \in st \quad \text{when}\ t \in TRN\ \text{then}\ st := st \setminus \{t\}\ \text{end}$$

The updated proof obligations are:

$$t \in st \ \Rightarrow\ t \in st \qquad \text{(C\_EN\_1')}$$
$$t \in st \ \wedge\ t \in TRN \ \wedge\ st' = st \setminus \{t\} \ \Rightarrow\ \neg t \in st'\ . \qquad \text{(NEG\_1')}$$

The proof obligations (C_EN_1') and (NEG_1') can be easily discharged.

Note that event depart has different guard and coarse schedule. It is our intention to design depart with a weak guard and a strong coarse schedule that allow us to prove system properties (e.g., invariance and progress properties). This gives more flexibility in strengthening events' guards and weakening schedules as needed during the course of refinement.

Since event arrive will not affect the reasoning about progress properties (it is always unscheduled), we are going to omit its refinement in the subsequent presentation.

4.3 First Refinement M_1— The Topology

In this refinement M_1, we first introduce the topology of the network in terms of blocks (ENV 3). We introduce a carrier set $BLK = \{Entry\} \cup PLF \cup \{Exit\}$ denoting the entry block, the platform blocks and the exit block, respectively. A new variable loc is added to denote the location of trains in the network, constrained by this invariant:

$$loc \in st \rightarrow BLK. \qquad \text{(inv1\_1)}$$

To capture FUN 4, we formulate the following safety properties:

$$\neg t \in st \ \ \textbf{un}\ \ \langle loc.t = Entry \rangle \qquad \text{(un1\_1)}$$
$$\langle loc.t = Entry \rangle \ \ \textbf{un}\ \ \langle loc.t \in PLF \rangle \qquad \text{(un1\_2)}$$
$$\langle loc.t \in PLF \rangle \ \ \textbf{un}\ \ \langle loc.t = Exit \rangle \qquad \text{(un1\_3)}$$
$$\langle loc.t = Exit \rangle \ \ \textbf{un}\ \ \neg t \in st \qquad \text{(un1\_4)}$$

They can be summarized in Figure 5.

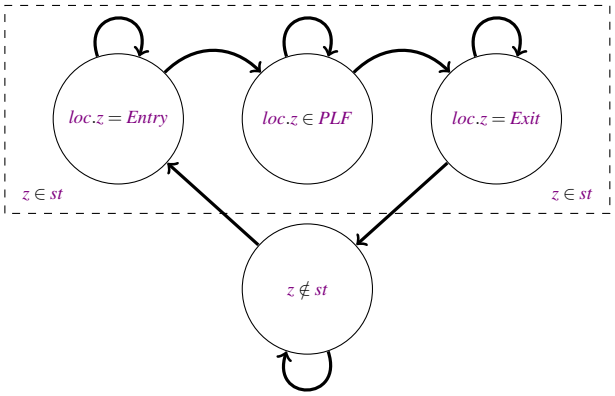

Fig. 5 State transitions for trains

We use (UN) to prove that they hold. In particular, for un1_2 and un1_3, we need to strengthen the guard of depart. Subsequently, in order to make sure that the schedule is stronger than the guard (condition (SCH_FIS)), we need to strengthen the coarse schedule accordingly. An assignment for loc is added for the maintenance of inv1_1.

depart $[t]$
during
$t \in st \wedge loc.t = Exit$
when
$t \in st \wedge loc.t = Exit$
then
$st := st \setminus \{t\}$
$loc := \{t\} \lhd loc$
end

In order to prove the refinement of depart, we apply Corollary 2 (coarse schedule replacing). In particular, conditions (C_FLW) and (C_STB) require us to prove the following properties:

$$t \in st \leadsto \langle loc.t = Exit \rangle \quad \text{(prg1_1)}$$
$$\langle loc.t = Exit \rangle \ \textbf{un}\ \neg t \in st \quad \text{(un1_4)}$$

From now on, we focus on reasoning about progress properties, e.g., prg1_1, omitting the reasoning about unless properties, e.g., un1_4. The proofs of these unless properties can be done using (UN) and will be omitted here.

In order to satisfy prg1_1, we first transform it into a transient property.

$$\begin{array}{ll} & t \in st \leadsto \langle loc.t = Exit \rangle \\ = & \quad \{\ \text{(Split-Off-Skip)}\ \} \\ & \langle \neg\, loc.t = Exit \rangle \leadsto \langle loc.t = Exit \rangle \\ \sqsubseteq & \quad \{\ \text{(Transitivity)}\ \} \\ & \langle \neg\, loc.t = Exit \rangle \leadsto \langle loc.t \in PLF \rangle \\ & \langle loc.t \in PLF \rangle \leadsto \langle loc.t = Exit \rangle \\ \sqsubseteq & \quad \left\{ \begin{array}{l} \text{(Split-Off-Skip) and with (prg1_2)} \\ \text{(see below) on first leads-to property} \end{array} \right\} \\ & \langle loc.t \in PLF \rangle \leadsto \langle loc.t = Exit \rangle \\ \sqsubseteq & \quad \{\ \text{(Ensure) with (un1_3)}\ \} \\ & \textbf{tr}\ \langle loc.t \in PLF \rangle \end{array}$$

with the new property:

$$\langle loc.t = Entry \rangle \leadsto \langle loc.t \in PLF \rangle \quad \text{(prg1_2)}$$

We implement the resulting transient property, i.e., **tr** $\langle loc.t \in PLF \rangle$ using a new event moveout. We leave the coarse and the fine schedule as some unknown c? and f?, and see how to derive them from resulting proof obligations.

moveout $[t] \mathrel{\widehat{=}}$ during c? upon f?
when $t \in st \wedge$ then $loc.t := Exit$ end
$loc.t \in PLF$

The proof that moveout implements the transient property is as follows.

$$\begin{array}{ll} & M_1 \vdash \textbf{tr}\ \langle loc.t \in PLF \rangle \\ \sqsubseteq & \quad \{\ \text{Transient rule (TRS) with moveout.}t\ \} \\ & M_1 \vdash \text{moveout.}t\ \textbf{falsifies}\ \langle loc.t \in PLF \rangle \\ = & \quad \{\ \text{Falsifies rule (FLS) with (C_EN_2) and (NEG_2)}\ \} \\ & M_1 \vdash \langle loc.t \in PLF \rangle \wedge c?\ \leadsto f? \end{array}$$

where:

$$\langle loc.t \in PLF \rangle \ \Rightarrow\ c? \quad \text{(C_EN_2)}$$

$$\begin{array}{ll} & \langle loc.t \in PLF \rangle \wedge c? \wedge f? \wedge \\ & loc' = loc \lhd\!\!+ \{t \mapsto Exit\}^2 \wedge st' = st \\ \Rightarrow & \neg \langle loc'.t \in PLF \rangle \end{array} \quad \text{(NEG_2)}$$

We design the coarse schedule c? and the fine schedule f? such that the goal, i.e., $\langle loc.t \in PLF \rangle \wedge c?\ \leadsto f?$, and conditions (C_EN_2) and (NEG_2) can be discharged trivially. One such design is to have c? being $t \in st \wedge loc.t \in PLF$ and f? being true. This gives us the following design for moveout:

moveout $[t]$
during
$t \in st \wedge loc.t \in PLF$
when
$t \in st \wedge loc.t \in PLF$
then
$loc.t := Exit$
end

The updated conditions (C_EN_2) and (NEG_2) is as follows and can be discharged easily.

$$\langle loc.t \in PLF \rangle \ \Rightarrow\ t \in st \wedge loc.t \in PLF \quad \text{(C_EN_2)}$$

$$\begin{array}{ll} & \langle loc.t \in PLF \rangle \wedge \\ & t \in st \wedge loc.t \in PLF \wedge \text{true} \wedge \\ & loc' = loc \lhd\!\!+ \{t \mapsto Exit\} \wedge st' = st \\ \Rightarrow & \neg \langle loc'.t \in PLF \rangle \end{array} \quad \text{(NEG_2)}$$

The remaining progress property, i.e., prg1_2, can be implemented in a similar fashion. We first transform prg1_2 into a transient property and implement it by the following new event movein.

[2] $loc \lhd\!\!+ \{t \mapsto Exit\}$ denotes a relation equal to loc excepts for the entry for t which is mapped to $Exit$.

movein $[t]$
during
$t \in st \wedge loc.t = Entry$
when
$t \in st \wedge loc.t = Entry$
then
$loc.t :\in PLF$
end

Applying the unless rule (UN), we can verify that the new events, i.e., movein and moveout, satisfy safety constraints, such as un1_1, un1_2, un1_3, and un1_4. It should be noted that those safety requirements guided our design of the new events along side the progress properties that they are meant to satisfy.

4.4 Second Refinement M_2— Preventing Collisions

In this refinement, M_2, we incorporate the safety requirement stating that there are no collisions between trains within the network, i.e., SAF 1. This is captured by a new invariant about loc:

$$(\forall t_1, t_2 : t_1, t_2 \in st \wedge loc.t_1 = loc.t_2 : t_1 = t_2). \qquad \text{(inv2\_1)}$$

The guard of event moveout needs to be strengthened with the fact that the exit block is free (i.e., $\neg Exit \in \text{ran}\, loc$), to maintain inv2_1. Due to the feasibility condition (SCH_FIS) for Unit-B events (requiring the schedules to be stronger than the guard), we need to strengthen the schedules accordingly. In particular, we add a fine schedule to moveout:

moveout $[t]$
during
$t \in st \wedge loc.t \in PLF$
upon
$\neg Exit \in \text{ran}\, loc$
when
$t \in st \ \wedge \ loc.t \in PLF \ \wedge \ \neg Exit \in \text{ran}\, loc$
then
$loc.t := Exit$
end

The scheduling information for moveout states that for any train t, if t stays in a platform for infinitely long and the exit block becomes free infinitely often, then t will eventually move out of the platform.

Heuristics. So far, all the scheduling information was encoded in coarse schedules. It is a general principle that coarse schedules should be used over fine schedules whenever possible. This is because, contrary to fine schedules, each coarse schedule can be manipulated in separation from each other.

For example, if during refinement we want to replace the coarse schedule $a_0 \wedge a_1 \wedge p$ with $c_0 \wedge c_1 \wedge p$, we can meet the proof obligation by proving

$$a_0 \wedge a_1 \wedge p \rightsquigarrow c_0 \wedge c_1 \wedge p \ ,$$
$$c_0 \wedge c_1 \wedge p \ \mathbf{un} \ a_0 \wedge a_1 \wedge p \ .$$

Since the two schedules involved can be arbitrarily complex, it is often more convenient to break down the proof obligation into the following smaller ones:

$$a_0 \rightsquigarrow c_0$$
$$a_1 \rightsquigarrow c_1$$
$$c_0 \ \mathbf{un} \ \neg a_0$$
$$c_1 \ \mathbf{un} \ \neg a_1$$

Instead of seeing the replacement of the coarse schedule as one operation, we can see it as the replacement of a_0 with c_0 and a_1 with c_1. This is especially convenient because the set of concrete schedules is rarely manipulated all at once.

In comparison, when refining fine schedules, the proof obligation ($a_0 \wedge a_1 \wedge p \rightsquigarrow c_0 \wedge c_1 \wedge p$) has to be dealt with as a whole. The comparison between coarse and fine schedule is similar when proving liveness properties. This is why we should keep the fine schedules as small as possible.

One situation favors fine schedules. Contentions happen when many events have to occur and the occurrence of one falsifies the schedules of the others. A mutual exclusion protocol is a good example of contention. The *enter_critical_section* event of each process has to occur when the process is waiting but no other process is in its critical section. When two processes, say p_0 and p_1 are waiting, if p_0 first enters its critical section, it falsifies p_1's schedule. If it were a coarse schedule, this means that there is no guarantee that the other process will ever be granted access to its critical section. The situation can be fixed by making part of the events' schedule coarse and the other part fine. The fact that p_1 is waiting is stable and no other process will falsify this. It can therefore safely be made into the coarse schedule. The other part, the condition that no other process be in their critical region, should be made into the fine schedule. It is not stable but as long as it becomes true infinitely often, p_1 will be granted access to its critical section.

As a rule of thumb, most schedules should be made into coarse schedules. When liveness properties cannot be proved, coarse schedules should be selectively made into fine schedules.
(end of *heuristics*)

In order to prove the refinement of moveout, we apply Corollary 3 (fine schedule strengthening), which requires to prove the following progress property. The abstract event moveout has no fine schedules, it is assumed to be true. Condition (F_STR) is trivial since the abstract fine schedule is

true. Condition (F_FLW) leads to the following property:

$$\langle loc.t \in PLF \rangle \;\leadsto\; \neg Exit \in \mathrm{ran}.loc\ , \qquad \text{(prg2\_1)}$$

which can be strengthened to

$$\mathrm{true} \;\leadsto\; \neg Exit \in \mathrm{ran}.loc \qquad \text{(prg2\_2)}$$

We satisfy prg2_2 (which is a transient property) by applying the transient rule (TRS) using event depart with the index denoting the train at the *Exit* location, i.e., $loc^{-1}.Exit$. Intuitively, the train at the exit block will eventually depart, hence the exit block becomes free.

$$\begin{array}{ll} & M_2 \vdash \textbf{tr}\ Exit \in \mathrm{ran}.loc \\ \sqsubseteq & \quad \{\ \text{Transient rule (TRS) with depart.}((loc^{-1}).Exit)\ \} \\ & M_2 \vdash \text{depart.}((loc^{-1}).Exit)\ \ \textbf{falsifies}\ \ Exit \in \mathrm{ran}.loc \\ = & \quad \{\ \text{Falsifies rule (FLS) with (C\_EN\_3) and (NEG\_3)}\ \} \\ & \mathrm{true} \end{array}$$

where:

$$\begin{array}{ll} & Exit \in \mathrm{ran}.loc \\ \Rightarrow & \\ & (loc^{-1}).Exit \in st \\ & \wedge\ loc.(\ (loc^{-1}).Exit\ ) = Exit \end{array} \qquad \text{(C\_EN\_3)}$$

$$\begin{array}{ll} & Exit \in \mathrm{ran}.loc \\ & \wedge\ (loc^{-1}).Exit \in st \\ & \wedge\ loc.(\ (loc^{-1}).Exit\ ) = Exit \\ & \wedge\ \mathrm{true} \\ & \wedge\ loc' = \{\ (loc^{-1}).Exit\ \} \lhd loc \\ & \wedge\ st' = st \setminus \{\ (loc^{-1}).Exit\ \} \\ \Rightarrow & \neg\ Exit \in \mathrm{ran}.loc' \end{array} \qquad \text{(NEG\_3)}$$

The proofs of conditions (C_EN_3) and (NEG_3) are straightforward and will be left out.

Finally we strengthen the guard of movein and subsequently strengthen its coarse schedule. We apply Corollary 2 (coarse schedule replacing) movein. The detailed proof is omitted here.

movein $[t]$
during
$t \in st \wedge loc.t = Entry \wedge \neg PLF \subseteq \mathrm{ran}.loc$
when
$t \in st \wedge loc.t = Entry \wedge \neg PLF \subseteq \mathrm{ran}.loc$
then
$loc.t :\in PLF \setminus \mathrm{ran}.loc$
end

4.4.1 Comparison between Coarse/Fine Schedules and Weak/Strong Fairness

Event moveout has both a coarse and a fine schedule. The alternative, using only weak or strong fairness, would complicate the proofs and make refinement of the system more difficult.

On the one hand, weak-fairness requires for the exit block to remain free continuously in order for trains to move out. This assumption is not met by the current system: if, infinitely often, another train than t located at a different platform moves on to the exit block before t does, t's weak-fairness allows for t to stay where it is forever. In other words, the weak- fairness assumption for moveout will be *too weak*; it does not guarantee that a train inside the station will eventually exit. An attempt to prove the refinement with the weakly-fair moveout event using Corollary 2 will lead to the following unprovable (C_STB) condition.

$$\begin{array}{ll} & \langle loc.t \in PLF \ \wedge\ \neg Exit \in \mathrm{ran}.loc \rangle \\ \textbf{un} & \neg(t \in st \ \wedge\ loc.t \in PLF) \end{array} \qquad (58)$$

The event that fails to satisfy (58) is moveout for train other than the current t.

On the other hand, strong-fairness would allow a train to access the exit block if it is present on the platform intermittently. This assumption is more flexible than we need since it allows behaviours where a train hops on and off the platform infinitely often while waiting for its turn at the exit block. The price of that flexibility is to entangle properties of the exit block with properties of trains: indeed, we would need not only to prove that the train will be on its platform and that the exit block will become free but that both happen simultaneously infinitely often. More formally, while we can prove that the strongly-fair moveout event refines the abstract moveout event, future refinement of moveout will be more difficult due to the stronger scheduling assumption. We choose to relinquish this flexibility and are therefore capable of structuring our proof better: on one hand, the train stays on its platform as long as necessary; independently, the exit block becomes free infinitely many times. This (choosing a weaker scheduling assumption) is similar to choosing a weaker guard such that safety properties are satisfied: it is minimalistic and gives more flexibility for later refinements.

The relationship between our schedules (coarse/fine) and fairness assumptions (weak/strong) can be illustrated as follows. Consider the following events with identical actions. The guard of these events are also the same as $c \wedge f$ for some

predicates c, f.

$e_{wf} \mathrel{\widehat{=}}$ during $c \wedge f$
when $c \wedge f$ then ... end

$e_{sch} \mathrel{\widehat{=}}$ during c upon f
when $c \wedge f$ then ... end

$e_{sf} \mathrel{\widehat{=}}$ upon $c \wedge f$
when $c \wedge f$ then ... end

Event e_{wf} is scheduled with weakly fairness, event e_{sf} is scheduled with strongly fairness. Event e_{sch}'s scheduling is split between a coarse schedule c and a fine schedule f. Consider the strength of their scheduling assumption, we have the following relationship:

$$sched.e_{wf} \Leftarrow sched.e_{sch} \Leftarrow sched.e_{sf} .$$

In fact, using coarse and fine schedules, we can specify a finer-grained spectrum of scheduling assumptions (compared to fairness assumptions) with the minimum being the weak-fairness assumption and the maximum being the strong-fairness assumption, as can be seen in Figure 6.

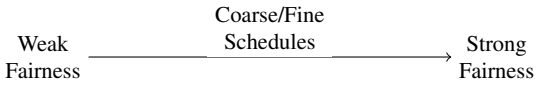

Fig. 6 The spectrum of scheduling assumptions

4.5 Third Refinement — The Actuators

In this refinement M_3, we focus on requirements EQP 5 and ENV 6 which describe the light signals in the station and state the assumption that the train drivers always obey these light signals. So far, moveout and movein which model the behaviour of individual trains and their driver, state that the trains move when it is safe to. However, the drivers often cannot judge for themselves whether it is safe to proceed because they cannot see all the dangers. This is why we adopt the convention of the light signals: wherever it may be dangerous to proceed, a light signal is located and can only be green if it is safe for the train it is addressed to advance. This allows us to change the guard and schedules of the train events to only refer to information that the drivers have access to.

We continue our development by modelling the signals associated with different blocks within the network. Variable g_sgn is introduced to denote the set of platform for which the light signal is green. We focus the rest of this section on the control of the signals regulating the departure from the platforms. In particular, invariants inv3_2 and inv3_3 state that if a platform signal is *green* then the exit block is free and the other platform signals are *red*. Invariant inv3_4 states that the signal is green only for occupied platforms.

invariants :
inv3_1 : $g\_sgn \subseteq PLF$
inv3_2 : $Exit \in \text{ran}\, loc \Rightarrow g\_sgn = \varnothing$
inv3_3 : $(\forall p, q : p, q \in g\_sgn : p = q)$
inv3_4 : $g\_sgn \subseteq \text{ran}\, loc$

We refine the moveout event to use the platform signal as follows.

moveout $[t]$
during
$t \in st \wedge loc.t \in PLF$
$\wedge$ $\boxed{loc.t \in g\_sgn}$
upon
~~$\neg Exit \in \text{ran}\, loc$~~
when
$t \in st \wedge loc.t \in PLF$
$\wedge$ $\boxed{loc.t \in g\_sgn}$
then
$loc.t := Exit$
$g\_sgn := g\_sgn \setminus \{loc.t\}$
end

The refinement of moveout is justified by applying Theorem 3 which requires us to prove the following:

$$\langle loc.t \in PLF \rangle \rightsquigarrow \langle loc.t \in PLF \cap g\_sgn \rangle \quad \text{(C\_FLW\_3)}$$
$$\langle loc.t \in PLF \cap g\_sgn \rangle \ \textbf{un}\ \neg \langle loc.t \in PLF \rangle \quad \text{(C\_STB\_3)}$$
$$\langle loc.t \in PLF \rangle \wedge \neg Exit \in \text{ran}\, loc \rightsquigarrow \text{true} \quad \text{(F\_FLW\_3)}$$
$$\langle loc.t \in PLF \cap g\_sgn \rangle \wedge \text{true} \Rightarrow \neg Exit \in \text{ran}\, loc \quad \text{(F\_STR\_3)}$$

(F_FLW_3) follows directly from the implication rule (Implication); (F_STR_3) follows from inv3_2; (C_STB_3) requires a number of simple proofs which will be left out. To discharge (C_FLW_3), we apply the ensure-rule (Ensure).

$$\langle loc.t \in PLF \rangle \ \textbf{un}\ \langle loc.t \in PLF \cap g\_sgn \rangle \quad \text{(un3\_1)}$$
$$\textbf{tr}\ \langle loc.t \in PLF \setminus g\_sgn \rangle \quad \text{(prg3\_1)}$$

Ignoring safety property un3_1, we focus on prg3_1. In order to have an event to falsify $\langle loc.t \in PLF \setminus g\_sgn \rangle$, we need the event to add $loc.t$ to g_sgn, i.e., to turn to green the signal of the platform where t is. This is clearly a task of the controller rather than a model of the trains. Therefore, we introduce a controller event, ctrl_platform, indexed with the platforms. Our first design for ctrl_platform is as follows.

ctrl_platform $[p]$
during
$p \in \text{ran}\, loc \wedge p \in PLF \setminus g\_sgn$
begin
$g\_sgn := g\_sgn \cup \{p\}$
end

The proof that ctrl_platform implements prg3_1 is as follows.

$$
\begin{array}{ll}
 & M_3 \vdash \textbf{tr}\ \langle loc.t \in PLF \setminus g\_sgn \rangle \\
\sqsubseteq & \quad \{\ \text{Transient rule (TRS) with ctrl\_platform.}(loc.t)\ \} \\
 & M_3 \vdash \text{ctrl\_platform.}(loc.t)\ \textbf{falsifies} \\
 & \qquad \langle loc.t \in PLF \setminus g\_sgn \rangle \\
= & \quad \{\ \text{Falsifies rule (FLS) with (C\_EN\_4) and (NEG\_4)}\ \} \\
 & \text{true}
\end{array}
$$

where:

$$
\begin{array}{ll}
 & \langle loc.t \in PLF \setminus g\_sgn \rangle \\
\Rightarrow & \\
 & \boxed{loc.t \in \text{ran}.loc} \qquad \text{// coarse schedule} \qquad \text{(C\_EN\_4)} \\
 & \wedge\ \boxed{loc.t \in PLF \setminus g\_sgn}
\end{array}
$$

$$
\begin{array}{ll}
 & \langle loc.t \in PLF \setminus g\_sgn \rangle \\
 & \wedge\ loc.t \in \text{ran}.loc \\
 & \wedge\ loc.t \in PLF \setminus g\_sgn \\
 & \wedge\ \boxed{g\_sgn' = g\_sgn \cup \{loc.t\}} \qquad \text{// action} \qquad \text{(NEG\_4)} \\
 & \wedge\ \boxed{loc' = loc} \\
\Rightarrow & \\
 & \neg \langle loc'.t \in PLF \setminus g\_sgn' \rangle
\end{array}
$$

Notice that we choose the coarse schedule of ctrl_platform so as to simplify the proof of (C_EN_4). Furthermore, we choose the assignment to g_sgn in such a way as to satisfy (NEG_4).

The next step is to prove that ctrl_platform satisfies the safety properties (i.e. unless properties and invariants) of the current refinement. In order to prove inv3_2 and inv3_3, we need to strengthen the guard to:

$$p \in PLF \ \wedge\ p \in \text{ran}.loc \ \wedge\ \neg Exit \in \text{ran}.loc \ \wedge\ g\_sgn = \varnothing\ .$$

Due to feasibility condition (SCH_FIS), we need to strengthen the schedules of ctrl_platform accordingly. Since strengthening the current coarse schedule will invalidate the current proof of (C_EN_4), we introduce the following new fine schedule for ctrl_platform:

$$\neg Exit \in \text{ran}.loc \ \wedge\ g\_sgn = \varnothing\ .$$

Event ctrl_platform is finalized as:

ctrl_platform [p]
during
$p \in \text{ran}.loc \wedge p \in PLF \setminus g\_sgn$
upon
$\boxed{\neg Exit \in \text{ran}.loc \wedge g\_sgn = \varnothing}$
when
$\boxed{\begin{array}{l} p \in PLF \ \wedge\ p \in \text{ran}.loc \\ \wedge\ \neg Exit \in \text{ran}.loc \ \wedge\ g\_sgn = \varnothing \end{array}}$
then
$g\_sgn := g\_sgn \cup \{p\}$
end

Event part	Formula	PO
coarse schedule	prg3_1	(C_EN_4)
action	prg3_1	(NEG_4)
guard	inv3_2 and inv3_3	invariance
fine schedule	guard	(SCH_FIS)

Table 1 Proof obligations and formulas justifying the design of ctrl_platform

Table 1 summarizes the design choices behind event ctrl_platform.

It is interesting to see that, in this refinement, we effectively shift the fine schedule from moveout (an environment event) to ctrl_platform (a controller event). The model remains abstract when it comes to specify the order in which the trains gain access to the exit block but specific enough to maintain liveness.

Event ctrl_platform is a specification for the computer to control the platform signals satisfying both safety and liveness properties of the overall system. In particular, the scheduling information states that if (1) a platform is occupied and the platform signal is *red* infinitely long and (2) the exit block is unoccupied and the other platform signals are all *red* infinitely often, then the system should eventually turn this platform signal to *green*. The refinement of event movein and how the entry signal is controlled is similar and omitted for the rest of the paper.

The new fine schedule changes the earlier proof of prg3_1 in two ways. First, the fine schedule gets in the antecedent of (NEG_4), which does not invalidate the proof of (NEG_4). Second, we get an additional leads-to property to prove corresponding to (F_EN).

$$
\begin{array}{lll}
\langle loc.t \in PLF \setminus g\_sgn \rangle & \rightsquigarrow & \neg Exit \in \text{ran}.loc \qquad \text{(F\_EN\_4)} \\
 & & \wedge\ g\_sgn = \varnothing
\end{array}
$$

We start the proof of (F_EN_4) by weakening its right-hand side to true (consequence of transitivity (Transitivity) and implication (Implication) rules) and we keep refining it until we can implement it simply.

$$
\begin{array}{ll}
 & \text{true} \ \rightsquigarrow\ \neg Exit \in \text{ran}.loc \ \wedge\ g\_sgn = \varnothing \\
\sqsubseteq & \quad \{\ \text{(Transitivity)}\ \} \\
 & \text{true} \ \rightsquigarrow\ g\_sgn = \varnothing \\
 & g\_sgn = \varnothing \ \rightsquigarrow\ \neg Exit \in \text{ran}.loc \ \wedge\ g\_sgn = \varnothing \\
\sqsubseteq & \left\{ \begin{array}{l} \text{(PSP) on second with } p := \text{true}\ , \\ q := \neg Exit \in \text{ran}.loc\ ,\ r := g\_sgn = \varnothing\ , \\ b := \neg Exit \in \text{ran}.loc \ \wedge\ g\_sgn = \varnothing \end{array} \right\} \\
 & \text{true} \ \rightsquigarrow\ g\_sgn = \varnothing \qquad \text{(prg3\_2)} \\
 & \text{true} \ \rightsquigarrow\ \neg Exit \in \text{ran}.loc \qquad \text{(prg3\_3)} \\
 & g\_sgn = \varnothing \ \textbf{un}\ \neg Exit \in \text{ran}.loc \ \wedge\ g\_sgn = \varnothing \qquad \text{(un3\_2)}
\end{array}
$$

We leave out the detailed proof for prg3_2 and un3_2: prg3_2 is a transient property which can be implemented by moveout event, un3_2 is a trivial safety property.

Property prg3_3 is identical to prg2_2 in the second refinement M_2. However, we cannot directly reuse prg2_2 since this could lead to circular reasoning. As explained below, additional precautions are required to avoid the problem.

4.5.1 Reusing Progress Properties

Property prg2_2, which states that the exit block becomes free infinitely often, turns out to be a key abstraction in M_2. In M_3 and later refinements, it will be very important that the exit block be available infinitely many times so that, even if a given train misses the first opportunity to move away from its platform, it still is certain to get a turn eventually. As a result, we would like to reuse prg2_2 in the reasoning about M_3 and subsequent refinements.

Earlier, we have proved that M_2 satisfies prg2_2, i.e.,

$$[\, ex.M_2 \;\Rightarrow\; \mathsf{prg2\_2} \,] \;. \tag{59}$$

To ensure that M_3 refines M_2, we prove

$$[\, ex.M_3 \;\Rightarrow\; ex.M_2 \,] \tag{60}$$

Succeed in proving (60), together with (59), will ensure that M_3 also satisfies prg2_2. However, we cannot directly reuse prg2_2 during the proof of (60): *our reasoning would be circular*.

In order to avoid circular reasoning, for each model, we keep a binary relation representing the dependency between progress properties of interest and the events that contribute to their implementation. Consider the initial model M_0, since the property prg0_1 is eventually implemented by depart, the dependency relation can be

$$\{\mathsf{prg0\_1} \mapsto (M_0.)\mathsf{depart}\}.$$

In the first refinement, consider the dependency for prg0_1, it is dependent on event $(M_1.)$depart (which is the refinement of the abstract event $(M_0.)$depart) and events moveout, movein (which are used for proving the refinement of $(M_0.)$depart by $(M_1.)$depart).

$$\begin{aligned}&\{\mathsf{prg0\_1} \mapsto (M_1.)\mathsf{depart},\\ &\;\mathsf{prg0\_1} \mapsto (M_1.)\mathsf{moveout},\\ &\;\mathsf{prg0\_1} \mapsto (M_1.)\mathsf{movein}\} \;.\end{aligned}$$

More formally, the dependency relation denotes the relationship between the scheduling assumptions of the events and the property these events implement. Consider a model M and its dependency relation:

$$\{P_1 \mapsto (M.)e_1, P_1 \mapsto (M.)e_2, P_2 \mapsto (M.)e_3\} \;.$$

The above relation encodes the following conditions:

$$[\, saf.M \wedge sched.e_1 \wedge sched.e_2 \;\Rightarrow\; P_1 \,] \tag{61}$$

$$[\, saf.M \wedge sched.e_3 \;\Rightarrow\; P_2 \,] \tag{62}$$

For a given model, we only need to consider the dependency relation for the progress properties that we want subsequent refinements to reuse. By default, progress properties are not reused.

In summary, the dependency relation summarizes the proofs (accumulated through refinement) of progress properties. We use it in order to avoid any circular reasoning linked to the reuse of progress properties. A progress property can be reused only after its effecting events have been refined. Naturally, during the proof of refinements of these dependent events, the progress property cannot be used. This means that if, in a refinement of M in the previous example, we replace the coarse schedule of e_1, we cannot use P_1 in the proof of (C_FLW).

Coming back to the train station example, we are interested in reusing prg2_2 which is introduced in M_2. Since prg2_2 is implemented by depart, the dependency relation for M_2 is as follows:

$$\{\mathsf{prg2\_2} \mapsto (M_2.)\mathsf{depart}\} \;.$$

Since depart is unchanged in M_3, its refinement is trivial. Therefore, we are free to make use of prg2_2 in all the proofs of liveness in M_3. It also follows that, the dependency for M_3 is the same as that of M_2, i.e.,

$$\{\mathsf{prg2\_2} \mapsto (M_3.)\mathsf{depart}\} \;.$$

In fact, property prg2_2 is also reused in future refinements.

4.6 Fourth Refinement — The Controller

In this refinement M_4, we focus on realising the software controller. At the end of the previous refinement, the controller is entirely specified by ctrl_platform which has a fine schedule. Although the fine schedule is a useful specification mechanism, we argue that it is not readily implementable. While it is easy to produce a correct (if not efficient) implementation for an event that has only a coarse schedule — a program that tests infinitely often (every second, every minutes or every year) the schedule and execute the event when its guard is true would be a correct implementation — it is not so straightforward for a fine schedule. Repeatedly testing a fine schedule will be incorrect in a situation where the fine schedule becomes true and false infinitely many times and that the program just happens to test infinitely many times only when it is false. This naive scheduler will fail to detect that the event has to be executed.

To make our controller more deterministic, we now proceed to refining ctrl_platform's fine schedule away. For that

purpose, we introduce three new variables (qe, hd, tl) to model a queue. Variable qe is an injective function from platform to an interval of integers where hd is the index of the first element of the queue and tl is the index just after the last element of the queue. This entails that the queue is empty when $hd = tl$.

As a convention, platforms are pulled at hd and inserted at tl. Below, $[hd, tl)$ is an integer interval that includes hd but excludes tl and $(qe^{-1})[\ \{hd\}\ ]$ is the application of the inverse of qe to the set $\{hd\}$. The latter has the particularity that, when hd is not in the range of qe, the expression evaluates to the empty set rather than being undefined.

invariants :
inv4_1 : $qe \in PLF \rightarrowtail [hd, tl)$
inv4_2 : $g\_sgn \subseteq (qe^{-1})[\ \{hd\}\ ]$
inv4_3 : $\mathrm{dom}.qe = PLF \cap \mathrm{ran}.loc$

In order to maintain inv4_3, we let movein increase tl and insert the platforms in the queue as they become occupied and moveout increase hd and remove the platforms as they become free.

movein [t]
during
...
when
...
then
...
$tl := tl + 1$
$qe.(loc'.t) := tl$
$loc.t :\in PLF \setminus \mathrm{ran}.loc$
end

moveout [t]
during
...
when
$loc.t \in g\_sgn$
then
...
$hd := hd + 1$
$qe := qe \rhd \{hd\}$
$loc.t := Exit$
$g\_sgn := g\_sgn \setminus \{loc.t\}$
end

With the queue, we can schedule the controller deterministically: to turn green the light signal of a platform, that platform has to be the head of the queue. Event ctrl_platform is refined as follows.

ctrl_platform [p]
during
~~$p \in PLF \cap \mathrm{ran}.loc$~~
$p \in \mathrm{dom}.qe$
$\wedge\ qe.p = hd$
$\wedge\ \neg Exit \in \mathrm{ran}.loc$
$\wedge\ \neg p \in g\_sgn$
upon
~~$\neg Exit \in \mathrm{ran}.loc \wedge g\_sgn = \varnothing$~~
when
$p \in PLF \wedge p \in \mathrm{ran}.loc$
$\wedge\ \neg Exit \in \mathrm{ran}.loc \wedge g\_sgn = \varnothing$
then
$g\_sgn := g\_sgn \cup \{p\}$
end

We apply Theorem 3 to prove the refinement of ctrl_platform. Omitting the trivial obligations related to (F_FLW) and (F_STR), we focus on the obligations for replacing $p \in PLF \cap \mathrm{ran}.loc$ (which is equivalent to $p \in \mathrm{dom}.qe$) with $\langle qe.p = hd \rangle \wedge \neg Exit \in \mathrm{ran}.loc$ for the coarse schedule of ctrl_platform (i.e., conditions (C_FLW) and (C_STB)).

$$p \in \mathrm{dom}.qe \quad \leadsto \quad \langle qe.p = hd \rangle \wedge \neg Exit \in \mathrm{ran}.loc \tag{C_FLW_5}$$

$$\langle qe.p = hd \rangle \wedge \neg Exit \in \mathrm{ran}.loc \quad \textbf{un} \quad \neg p \in \mathrm{dom}.qe \tag{C_STB_5}$$

So far in our development, progress properties can be separated in two different groups:

1. Those that are satisfied in a single step. These properties are proved by transforming them into transient properties (for example, making use of rules such as ensure-rule (Ensure)). Each transient property is implement by an individual event using a combination of transient rule (TRS) and falsifies rule (FLS).
2. Those that are satisfied in some pre-determined number of steps. These properties are proved by breaking them down into several properties that can be satisfied in a single step using transitivity (Transitivity).

Property (C_FLW_5) does not fit neither categories so far. In fact, the number of steps to satisfy (C_FLW_5) depends on the position of the platform p within the queue qe. As a result, in order to prove (C_FLW_5) we apply the induction rule (Induction).

$$p \in \mathrm{dom}.qe \quad \leadsto \quad \langle qe.p = hd \rangle \wedge \neg Exit \in \mathrm{ran}.loc$$

$\sqsubseteq$ { Induction rule (Induction)}

$$\langle qe.p - hd = M \rangle \leadsto \langle qe.p - hd < M \rangle \vee \langle qe.p = hd \wedge \neg Exit \in \mathrm{ran}.loc \rangle \tag{prg4_1}$$

$\sqsubseteq$ { (PSP) with $p := \langle qe.p - hd = M \rangle$, $q := \neg\langle qe.p - hd = M \rangle$, $r := \langle qe.p - hd \leq M \rangle$, $b := \langle qe.p = hd \rangle \wedge \neg Exit \in \mathrm{ran}.loc$ }

$$\langle qe.p - hd = M \rangle \quad \leadsto \quad \neg\langle qe.p - hd = M \rangle \tag{prg4_2}$$

$$\langle qe.p - hd \leq M \rangle \quad \textbf{un} \quad \langle qe.p = hd \rangle \wedge \neg Exit \in \mathrm{ran}.loc \tag{un4_1}$$

We focus on the development for prg4_2. It basically says that, eventually, either the value of $qe.p - hd$ changes or p is no longer in the queue. While this is exactly what moveout does, we cannot prove that moveout falsifies $\langle qe.p - hd = M \rangle$. This is because it could take as many as three steps to do that. For example, let us assume that there is a train at the $Exit$ block, i.e., $Exit \in \mathrm{ran}.loc$, all the platform signals are red, hence $g\_sgn = \varnothing$. In order to falsify $\langle qe.p - hd = M \rangle$, the following steps have to happen:

1. Event depart frees the *Exit* block,
2. Event ctrl_platform turns the platform signal to green for some platform,
3. Event moveout moves to the exit block the train located at the platform for which the signal is green.

As a result, we use (Transitivity) to split prg4_2 into three different properties.

$$\langle qe.p - hd = M\rangle \;\leadsto\; \langle qe.p - hd = M\rangle \wedge \neg Exit \in \mathrm{ran}\, loc \qquad \text{(prg4\_3)}$$

$$\langle qe.p - hd = M\rangle \wedge \neg Exit \in \mathrm{ran}\, loc \;\leadsto\; \langle qe.p - hd = M\rangle \wedge \neg g\_sgn \subseteq \varnothing \wedge \neg Exit \in \mathrm{ran}\, loc \qquad \text{(prg4\_4)}$$

$$\langle qe.p - hd = M\rangle \wedge \neg g\_sgn \subseteq \varnothing \wedge \neg Exit \in \mathrm{ran}\, loc \;\leadsto\; \neg\langle qe.p - hd = M\rangle \qquad \text{(prg4\_5)}$$

Subsequently, our intention is to implement prg4_3 with depart, prg4_4 with ctrl_platform, and prg4_5 with moveout, according to our informal reasoning before. The detail proofs are left out.

4.7 Refinement 5 — Removal of the Event Indices

At the end of the fourth refinement, the controller event ctrl_platform is indexed with the platform p whose signal is going to be turned green. However, since p is determined as the head of the queue qe, we can remove the index of ctrl_platform. The final version of ctrl_platform is as follows.

ctrl_platform
  during
    $hd < tl$
    $\wedge\ \neg Exit \in \mathrm{ran}\, loc$
    $\wedge\ \neg(qe^{-1}).hd \in g\_sgn$
  when
    $\neg Exit \in \mathrm{ran}\, loc \wedge g\_sgn = \varnothing$
  then
    $g\_sgn := g\_sgn \cup \{(qe^{-1}).hd\}$
  end

The refinement of ctrl_platform can be justified trivially using Theorem 4, with $(qe^{-1}).hd$ as the witness for the removed index p.

The first-in-first-out policy may appear too rigid because it does not allow trains to stand still at a platform for a while when they are ahead of schedule. We chose to adhere to it because of its simplicity which is a correct choice since the ability for trains to linger is not one of the stated requirements. It would, however, make for an interesting model, one which is outside the scope of this paper.

4.8 Summary

Our development from M_0 to M_5 is driven by both safety and progress concerns. In particular, we choose on purpose to take into account liveness requirement FUN 2 at M_0. As a result, the need to prove and maintain progress properties justifies a number of design decisions within our development. We summarize the key features and techniques of Unit-B that have illustrated throughout our case study.

M_0 We introduce the basis of modelling using scheduled events and application of transient rule (TRS) to prove simple progress properties.

M_1 We illustrate how to refine scheduled events, and applications of transitivity rule (Transitivity) and ensure rule (Ensure) to prove progress properties.

M_2 We discuss the difference between coarse/fine schedules and weak/strong fairness.

M_3 We illustrate how progress properties that have been proved in earlier abstract models can be reused through refinement.

M_4 We compare different strategies for implementing progress properties: single step (ensure and transient rules), pre-determined number of steps (transitivity rule), arbitrary finite number of steps (induction rule).

M_5 We illustrate how events can be made more concrete by removing indices.

5 Conclusion

In this paper, we presented Unit-B, a formal method inspired by Event-B and UNITY. Our method allows systems to be developed gradually via refinement and support reasoning about both safety and liveness properties. An important feature of Unit-B is the notion of coarse and fine schedules for events. Standard weak and strong fairness assumptions can be expressed using these event schedules. We proposed and prove the soundness of refinement rules to manipulate the coarse and fine schedules so that liveness properties are preserved automatically (i.e., without the need to reprove them). We illustrated Unit-B by developing a signal control system.

A key observation in Unit-B is the role of event scheduling regarding liveness properties being similar to the role of guards regarding safety properties. Guards prevent events from occurring in unsafe states so that safety properties will not be violated; similarly, schedules ensure the occurrence of events in order to satisfy liveness properties.

Another key aspect of Unit-B is the role of progress properties during refinement: the obligation to prove new progress properties in the application of refinement rules motivates the introduction of new events and suggests the

refinement of old events. In short, the progress considerations guide the refinement of the system.

Related work Unit-B and Event-B differ mainly in the scheduling assumptions. In Event-B, event executions are assumed to satisfy a *minimal progress* condition: as long as there are some enabled events, one of them will be executed non-deterministically. Given this assumption, certain liveness properties can be proved for Event-B models such as *progress* and *persistence* [10]. The minimum progress assumption is often too weak to prove the required set of liveness properties. Furthermore, the liveness properties that can be proved using minimal progress have to be reproved in later refinements to ascertain that they still hold.

TLA+ [17] is another well-known formal method based on refinement supporting reasoning about liveness properties. The execution of a TLA+ model is also captured as a formula with safety and liveness sub-formulae expressed in the Temporal Logic of Actions (TLA) [16]. Actions in TLA+ (events in Unit-B) can be scheduled with weak or strong fairness. Refinement in TLA+ is based on the WF2 and SF2 rules [16]. Rule WF2 allows a weakly fair event to be refined by another weakly fair event and Rule SF2 allows a strongly fair event to be refined by another strongly fair event. The refinement rule for Unit-B is more general than the combination of WF2 and SF2: during refinement, we can trade freely between the weakly fair component (i.e., the coarse schedule) and the strongly fair component (i.e., the fine schedule). Moreover, liveness properties in TLA+ are considered to be *unimportant* [17, Chapter 8]. In our opinion, developing systems satisfying liveness properties is as important as ensuring that the systems satisfy safety properties. We argue that the liveness properties should be considered from the early stages of the design. Indeed, addressing liveness properties as an after thought in the design process will often lead to complicated proofs, since the model is not designed with proofs of liveness properties in mind.

See [18] for a review of the temporal logic framework developed by Manna and Pnueli. The authors use fair transition systems for the semantics of concurrent or reactive programs and temporal logic for specifying system properties. Rules are provided for proving response properties (called progress properties in this paper) that rely on just or compassionate transitions (equivalent to the weak and strong fairness scheduling policies) of the system for their validity. Although the Manna-Pnueli framework does not have a progress preserving refinement calculus as in Unit-B, it does have rules for data abstraction and compositional reasoning.

The idea of combining different formal methods to reason about liveness properties is also explored by other researchers. In [20], the authors combine Event-B and TLA+ for proving liveness properties in population protocols. While refinement has been used in their development, liveness properties are not preserved: progress properties have to be reproved at each level of refinement.

Future work Currently, we only consider superposition refinement in Unit-B where variables are retained during refinement. More generally, variables can be removed and replaced by other variables during refinement (data refinement). We are working on extending Unit-B to provide rules for data refinement.

Another important technique for coping with the difficulties in developing complex systems is composition / decomposition and is already a part of methods such as Event-B and UNITY. We intend to investigate on how this technique can be added to Unit-B, in particular, the role of event scheduling during composition / decomposition.

Tool support is currently under construction under the name *Literate Unit-B*. The goal is to integrate seamlessly the activities of modelling, proving and documenting. We do so by making equational proofs first class citizens in models, by taking LaTeX source files at the input of the tool and allowing arbitrary interleaving of model and proof elements. We use the Z3 SMT solver [21] to discharge the proofs obligations and to validate the proof steps.

The goal is to allow the user to formulate formal proofs in a clear manner and integrate them in the documentation of the models, letting the tool verify that every step of reasoning is sound or suggest where a lemma would be needed to justify a step. Such a tool is needed for the Unit-B method to be practical. This tool substantially reduces the burden of validity checking therefore allowing developers to focus on the software design.

As is the case with Rodin [3], the tool for Event-B, a large percentage of proof obligations can be discharged automatically, freeing the user from the need to check many simple facts. This leaves him with the job of proving only the hardest obligations. It is useful then to be able to design and present the proof of these hard theorems using a format that is both readable by humans and amenable to formal reasoning by humans. We believe the equational format [9] exhibits these properties since it permits the user to focus on one line of reasoning at a time.

Acknowledgment

We would like to thank the anonymous reviewers for their constructive comments which helped improve the paper significantly. The first and last authors gratefully acknowledge a Discovery Grant from NSERC (National Science and Engineering Research Council).

References

1. Abrial, J.R.: The B-book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Abrial, J.R.: Modeling in Event-B - System and Software Engineering. Cambridge University Press (2010)
3. Abrial, J.R., Butler, M., Hallerstede, S., Hoang, T., Mehta, F., Voisin, L.: Rodin: an open toolset for modelling and reasoning in Event-B. STTT **12**(6), 447–466 (2010)
4. Chandy, M., Misra, J.: Parallel program design - a foundation. Addison-Wesley (1989)
5. Dijkstra, E., Scholten, C.: Predicate Calculus and Program Semantics. Springer-Verlag New York, Inc., New York, NY, USA (1990)
6. Dijkstra, E.W.: The notational conventions I adopted, and why (2000). URL http://www.cs.utexas.edu/users/EWD/ewd13xx/EWD1300.PDF. Circulated privately
7. Dijkstra, R.: Computation calculus: Bridging a formalization gap. Mathematics of Program Construction (1998). URL http://www.springerlink.com/index/QH44VD66GVE0U3LU.pdf
8. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: B.W. Boehm, D. Garlan, J. Kramer (eds.) ICSE' 99, pp. 411–420. ACM, Los Angeles, CA, USA (1999). URL http://portal.acm.org/citation.cfm?id=302405.302672
9. Gries, D., Schneider, F.B.: A Logical Approach to Discrete Math. Springer (1993)
10. Hoang, T., Abrial, J.R.: Reasoning about liveness properties in Event-B. In: S. Qin, Z. Qiu (eds.) ICFEM, *LNCS*, vol. 6991, pp. 456–471. Springer-Verlag (2011)
11. Hudon, S.: A progress preserving refinement. Master's thesis, ETH Zurich (2011)
12. Hudon, S., Hoang, T.: Development of control systems guided by models of their environment. ENTCS **280**, 57–68 (2011)
13. Hudon, S., Hoang, T.S.: Systems design guided by progress concerns. In: E.B. Johnsen, L. Petre (eds.) Integrated Formal Methods, *Lecture Notes in Computer Science*, vol. 7940, pp. 16–30. Springer-Verlag, Turku, Finland (2013). http://dx.doi.org/10.1007/978-3-642-38613-8_2
14. Jones, C.B.: Systematic software development using VDM. Prentice Hall International Series in Computer Science. Prentice Hall (1986)
15. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. **3**(2), 125–143 (1977)
16. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
17. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
18. Manna, Z., Pnueli, A.: Temporal verification of reactive systems: Response. In: Z. Manna, D. Peled (eds.) Time for Verification, Essays in Memory of Amir Pnueli, *Lecture Notes in Computer Science*, vol. 6200, pp. 279–361. Springer, Berlin (2010). URL http://dx.doi.org/10.1007/978-3-642-13754-9_13
19. Mehta, F.D.: Proofs for the working engineer. Ph.D. thesis, ETH, Zurich (2008). URL http://dx.doi.org/10.3929/ethz-a-005635243
20. Méry, D., Poppleton, M.: Formal modelling and verification of population protocols. In: E.B. Johnsen, L. Petre (eds.) IFM, *Lecture Notes in Computer Science*, vol. 7940, pp. 208–222. Springer (2013)
21. de Moura, L.M., Bjørner, N.: Z3: an efficient SMT solver. In: C.R. Ramakrishnan, J. Rehof (eds.) TACAS 2008, *Lecture Notes in Computer Science*, vol. 4963, pp. 337–340. Springer, Budapest, Hungary (2008). URL http://dx.doi.org/10.1007/978-3-540-78800-3_24
22. Spivey, J.M.: Z Notation - a reference manual (2. ed.). Prentice Hall International Series in Computer Science. Prentice Hall (1992)