Noname manuscript No.
(will be inserted by the editor)

Progress Concerns as Design Guidelines

Simon Hudon · Thai Son Hoang

Wednesday 31st July, 2013

Abstract We present Unit-B, a formal method inspired by Event-B and UNITY, for designing systems via step-wise refinement preserving both safety and liveness properties. In particular, we introduce the notion of coarse- and fine-schedules for events, a generalisation of weak- and strong-fairness assumptions. We propose proof rules for reasoning about progress properties related to the schedules. Furthermore, we develop techniques for refining systems by adapting event schedules such that liveness properties are preserved. We illustrate our approach by an example to show that Unit-B developments can be guided by both safety and liveness requirements.

Keywords: progress properties, refinement, fairness, scheduling, Unit-B.

1 Introduction

Developing systems satisfying their desirable properties is a non-trivial task. Formal methods have been seen as a 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 in a hierarchical development.

System properties are often put into two classes: *safety* and *liveness* [**?**]. A safety property ensures that undesirable behaviours will never happen during the 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 properties. Although safety properties are often considered the most important ones, we argue that having *live* systems is also important. A system that is safe but not live is useless. For example, consider an elevator system that does not move. Such an elevator system is safe (nobody could ever get hurt), yet useless.

Department of Computer Science, York University, Toronto, Canada
E-mail: simon@cse.yorku.ca · Institute of Information Security, ETH-Zurich, Switzerland
E-mail: htson@inf.ethz.ch

According to a survey [?], liveness properties (in terms of *existence* and *progress*) amount to 45% of the overall system properties.

In most refinement-based development methods such as (B, Event-B, VDM, Z) the focus is on preserving safety properties. A possible problem for such safety-oriented methods is that when applying them to design a system, we can make the design so safe that it becomes unusable. It is hence our aim to design a refinement framework preserving both safety and liveness properties.

Some modelling methods such as UNITY [?], include the capability of reasoning about liveness properties. In UNITY, there is a clear distinction between specifications (temporal properties) and programs (transition systems). Refinement in UNITY involves transforming specifications according to the UNITY logic. At the end of the refinement process, one obtains several temporal properties which then can be implemented by some program fragments according to well-defined rules. As a result, programs (transition systems) in UNITY are not part of the design, they are the output of the refinement process. A disadvantage of this approach is that the transformation of temporal properties can make the choice of refinements hard to understand. In order to overcome this limitation, we unified the notion of specification and that of program, making smoother the transition from one to the other.

In this paper, we present a formal method, namely Unit-B [?], inspired by UNITY [?] and Event-B [?]. We borrow the ideas of system development from Event-B, in which a series of models is constructed and linked by refinement relationships. The temporal logic that we use to specify and to reason about progress properties is based on UNITY.

The main attraction of our method is that it incorporates the reasoning about safety and liveness properties within a single refinement framework. Furthermore, our approach features the novel notions of coarse- and fine-schedules, a generalisation of the standard weak- and strong-fairness assumptions. They allow us (1) to reason about the satisfiability of progress properties by a given model, and (2) to refine a given model while preserving liveness properties. This makes it possible in Unit-B to introduce liveness properties at any stage of the development process. Subsequently, not only does it rule out any design that would be too conservative, but it also justifies design decisions. As a result, liveness properties, in particular progress properties, act as a design guideline for developing systems.

We give a semantics for Unit-B models and their properties using computation calculus [?]. This enables us to formally prove the rules for reasoning about properties and refinement relationship in Unit-B.

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

2 Background: Computation Calculus

This section gives a brief introduction to computation calculus, based on [**?**]. Let $\mathcal{S}$ be the state space: a non-empty set of "states". Let $\mathcal{C}$ be the computation space: a set of non-empty (finite or infinite) sequences of states ("computations"). The set of computation predicates $CPred$ is defined as follows.

Definition 1 (Computation Predicate) $CPred = \mathcal{C} \rightarrow \mathbb{B}$*, 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 $\tau \in \mathcal{C}$, we have,[1]

$$(s \Rightarrow t).\tau \;\;\equiv\;\; (s.\tau \Rightarrow t.\tau) \quad (1) \qquad\qquad \langle \forall\, i ::\; s.i \rangle.\tau \;\;\equiv\;\; \langle \forall\, i ::\; s.i.\tau \rangle\;. \quad (2)$$

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

$$[\; s \;] \;\;\equiv\;\; \langle \forall\, \tau ::\; s.\tau \rangle \qquad (3)$$

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 [**?**]. In particular, $\mathbf{true}$ (maps all computations to TRUE) and $\mathbf{false}$ (maps all computations to FALSE) are the "top" and the "bottom" elements of the complete boolean lattice with the order $[\; \_ \Rightarrow \_ \;]$ specifying 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)

$$(s;t).\tau \;\;\equiv\;\; (\#\tau = \infty \wedge s.\tau) \;\vee\; \langle \exists\, n \colon\; n < \#\tau \colon\; s.(\tau \uparrow n{+}1) \wedge t.(\tau \downarrow n) \rangle \quad (4)$$

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

Intuitively, a computation τ satisfies $s\,;t$ if either it is an infinite computation satisfying s, or there is a finite prefix of τ (i.e. $\tau \uparrow n{+}1$) satisfying s and the corresponding suffix $\tau \downarrow n$ (which overlaps with the prefix on one state) satisfying t.

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

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

$$\mathbf{E} = \mathbf{true}; \mathbf{false} \quad (5) \qquad\qquad s \textit{ is eternal} \;\equiv\; [\; s \Rightarrow \mathbf{E} \;] \quad (7)$$

$$\mathbf{F} = \neg\mathbf{E} \quad (6) \qquad\qquad s \textit{ is finite} \;\equiv\; [\; s \Rightarrow \mathbf{F} \;] \quad (8)$$

[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 $(f.x).y$.

Given $\mathbf{F}$ the temporal "eventually" operator (i.e., $\diamondsuit$) 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) $\mathbf{G}\, s = \neg(\mathbf{F}; \neg s)$, *for any predicate* s

Important properties of $\mathbf{G}$ are that it is strengthening and monotonic. For any predicates s and t, we have:

$$[\, \mathbf{G}\, s \Rightarrow s \,]\,, \tag{9}$$

$$[\, s \Rightarrow t \,] \;\;\Rightarrow\;\; [\, \mathbf{G}\, s \Rightarrow \mathbf{G}\, t \,]\,, \tag{10}$$

$$[\, \mathbf{G}\,(s \Rightarrow t) \;\;\Rightarrow\;\; (\mathbf{G}\, s \Rightarrow \mathbf{G}\, t) \,]\,. \tag{11}$$

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

Definition 5 (Constant $\mathbb{1}$) *For any computation* τ, $\mathbb{1}.\tau \;\;\equiv\;\; \#\tau = 1$

State Predicates 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 6 (State Predicate) *For any predicate* p, $p \in SPred \;\;\equiv\;\; [\, p \Rightarrow \mathbb{1} \,]$.

A consequence of this definition is that $SPred$ is also a complete boolean lattice with the order $[\, \_ \Rightarrow \_ \,]$, with $\mathbb{1}$ and $\mathbf{false}$ 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 7 (State predicate negation $\sim$) *For any state predicate* p, $\sim p = \neg p \wedge \mathbb{1}$.

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 $\bullet : SPred \to CPred$ is introduced to denote this predicate.

Definition 8 (Initially Operator) *For any state predicate* p, $\bullet p = p\,;\mathbf{true}$

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*, $p\,; q \;=\; p \wedge q$.

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 $\mathbf{X} \in CPred$ is defined for this purpose.

Definition 9 (Atomic Actions) *For any computation* τ *and predicate* a,

$$\mathbf{X}.\tau \;\;\equiv\;\; \#\tau = 2 \tag{12}$$

$$a \textit{ is an atomic action} \;\;\equiv\;\; [\, a \Rightarrow \mathbf{X} \,] \tag{13}$$

Given the above definition, the "next" operator can be expressed as $\mathbf{X}\,; s$ for arbitrary computation s.

3 The Unit-B Method

This section presents our contribution: the Unit-B method which is inspired by Event-B and UNITY. Similar to Event-B, Unit-B is aimed at the design of software systems by stepwise refinement. It differs from Event-B by the capability of reasoning about progress properties and its refinement-order which preserves liveness properties. It also differs from UNITY by unifying the notions of programs and specifications, allowing refinement of programs.

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:

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any}\ t\ \textbf{where}\ g.t.v\ \textbf{during}\ c.t.v\ \ \textbf{upon}\ f.t.v\ \ \textbf{then}\ s.t.v.v'\ \ \textbf{end}\ \ , \qquad (14)$$

where t are the parameters, g is the *guard*, c is the *coarse-schedule*, f is the *fine-schedule*, and s is the *action* changing state variables v. The action is usually made up of several *assignments*, either deterministic ($:=$) or non-deterministic ($:|$). An event e with parameters t stands for multiple events. Each corresponds to several non-parameterised events $\mathsf{e}.t$, one for each possible value of the parameter t. Here g, c, f are state predicates. An event is said to be enabled when the guard g holds. The scheduling assumption of the event is represented by c and f as follows: if c holds for infinitely long and f holds infinitely often then the event 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.

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

3.2 Semantics

We are going to use computation calculus to give the semantics of Unit-B models. Let $\mathbf{M}$ be a Unit-B model containing a set of events of the form (14) and an initialisation predicate init. Since the action of the event can be described by a before-after predicate $s.t.v.v'$, it corresponds to an atomic action $\mathbf{S}.t \ = \ \langle \forall\, e ::\ \bullet(e = v) \ \Rightarrow\ \mathbf{X}\,;s.t.e.v \rangle$. Given that an event $\mathsf{e}.t$ can only be carried out when it is enabled, the effect of each event execution can therefore be formulated as follows: $act.(\mathsf{e}.t) = g.t\,;\,\mathbf{S}.t$. A special constant SKIP is used to denote the atomic action that does not change the state.

Definition 10 (Constant SKIP) $\text{SKIP}.\tau \quad \equiv \quad \#\tau = 2 \ \wedge\ \tau.0 = \tau.1$, *for all traces* τ *(*$\tau.0$, $\tau.1$ *denotes the first two elements of* τ*).*

The semantics of $\mathbf{M}$ is given by a computation predicate $ex.\mathbf{M}$ which is a conjunction of a "safety part" $\mathit{saf}.\mathbf{M}$ and a "liveness part" $\mathit{live}.\mathbf{M}$, i.e.,

$$[\ ex.\mathbf{M} \ \equiv\ \mathit{saf}.\mathbf{M} \wedge \mathit{live}.\mathbf{M}\ ]\ . \tag{15}$$

A property represented by a formula s is satisfied by $\mathbf{M}$, if

$$[\ ex.\mathbf{M} \Rightarrow s\ ]\ . \tag{16}$$

3.2.1 Safety

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

$$[\ \mathit{step}.\mathbf{M} \ \equiv\ \langle \exists\, \mathsf{e}, t\colon\ \mathsf{e}.t \in \mathbf{M}\colon\ act.(\mathsf{e}.t)\rangle \vee \text{SKIP}\ ] \tag{17}$$

$$[\ \mathit{saf}.\mathbf{M} \ \equiv\ \bullet\,\mathsf{init} \wedge \mathbf{G}\,(\mathit{step}.\mathbf{M}\,;\,\mathbf{true})\ ] \tag{18}$$

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

An invariant $I.v$ is a state-properties that hold at every reachable state of the model. In order to prove that $I.v$ is an invariant of $\mathbf{M}$, we prove that $[\ ex.\mathbf{M} \Rightarrow \mathbf{G}\ \bullet I\ ]$. In particular, we rely solely on the safety part of the model to prove invariance properties, i.e., we prove $[\ \mathit{saf}.\mathbf{M} \Rightarrow \mathbf{G}\ \bullet I\ ]$. This leads to the well-known invariance principle.

$$\begin{array}{l} \quad [\ \mathsf{init} \Rightarrow I\ ]\ \wedge\ [\ \langle \forall\, \mathsf{e}, t\colon\ \mathsf{e}.t \in \mathbf{M}\colon\ I\,;\,act.(\mathsf{e}.t) \ \Rightarrow\ \mathbf{X};I\rangle\ ] \\ \Rightarrow \\ \quad [\ \mathit{saf}.\mathbf{M} \Rightarrow \mathbf{G}\ \bullet I\ ] \end{array} \tag{INV}$$

Invariance properties are important for reasoning about the correctness of the models since they limit the set of reachable states. In particular, invariance properties can be used as additional assumptions in proofs for progress properties.

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

Definition 11 (un operator) *For all state predicates* p *and* q*,*

$$[\ (p\ \textbf{\textit{un}}\ q) \ \equiv\ \mathbf{G}\,(\bullet p \ \Rightarrow\ (\mathbf{G}\ \bullet p)\,;\,(\mathbb{1} \vee \mathbf{X})\,;\,\bullet q)\ ] \tag{19}$$

Informally, p **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 (19) 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 (Proving an un-property) *Consider a machine* $\mathbf{M}$ *and property* p ***un*** q. *If*

$$\langle \forall\, \mathsf{e}, t \colon\ \mathsf{e}.t \in \mathbf{M} \colon\ \mathbf{G}(\,(p \wedge {\sim} q);act.(\mathsf{e}.t);\mathbf{true}\ \Rightarrow\ \mathbf{X};(p \vee q);\mathbf{true}\,)\rangle \tag{20}$$

then $[\ ex.\mathbf{M}\ \Rightarrow\ p\,\textbf{\textit{un}}\,q\ ]$

Proof (Sketch) Condition (20) ensures that every event of $\mathbf{M}$ 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.

3.2.2 Liveness

For each event of the form (14), its schedule $sched.(\mathsf{e}.t)$ is formulated as follows, where c and f are the event's coarse- and fine-schedule, respectively.

$$[\ sched.(\mathsf{e}.t)\quad \equiv\quad \mathbf{G}\,(\mathbf{G}\ \bullet c\ \wedge\ \mathbf{G}\,\mathbf{F}\,;\bullet f\ \Rightarrow\ \mathbf{F}\,;f\,;act.(\mathsf{e}.t)\,;\mathbf{true})\ ]\ . \tag{21}$$

To ensure that the event $\mathsf{e}.t$ only occurs when it is enabled, we require the following *feasibility* condition:

$$[\ ex.\mathbf{M}\ \Rightarrow\ \mathbf{G}\ \bullet(c \wedge f\ \Rightarrow\ g)\ ] \tag{SCH-FIS}$$

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

$$[\ \mathrm{wf}.(\mathsf{e}.t) \equiv \mathbf{G}\,(\mathbf{G}\ \bullet g\ \Rightarrow\ \mathbf{F};act.(\mathsf{e}.t);\mathbf{true})\ ] \tag{22}$$

$$[\ \mathrm{sf}.(\mathsf{e}.t) \equiv \mathbf{G}\,(\mathbf{G}\,\mathbf{F};\bullet g\ \Rightarrow\ \mathbf{F};act.(\mathsf{e}.t);\mathbf{true})\ ] \tag{23}$$

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

$$[\ live.\mathbf{M}\quad \equiv\quad \langle \forall\, \mathsf{e}, t \colon\ \mathsf{e}.t \in \mathbf{M} \colon\ sched.(\mathsf{e}.t)\rangle\ ] \tag{24}$$

3.3 Progress Properties

Progress properties are of the form $p \leadsto q$, where $\leadsto$ is the leads-to operator.

Definition 12 ($\leadsto$ operator) *For all state predicates* p *and* q,

$$[\ (p \leadsto q)\quad \equiv\quad \mathbf{G}\,(\bullet p\ \Rightarrow\ \mathbf{F}\ \bullet q)\ ] \tag{25}$$

In this paper, properties and theorems are often written without explicit quantifications: these are universally quantified over all values of the free variables occurring in them.

Important properties of $\leadsto$ are as follows. For state predicates p, q, and r, we have:

$$[\ (p \Rightarrow q) \quad \Rightarrow \quad (p \leadsto q)\ ] \qquad \text{(Implication)}$$

$$[\ (p \leadsto q) \land (q \leadsto r) \quad \Rightarrow \quad (p \leadsto r)\ ] \qquad \text{(Transitivity)}$$

$$[\ (p \leadsto q) \quad \equiv \quad (p \land {\sim} q \quad \leadsto \quad q)\ ] \qquad \text{(Split-Off-Skip)}$$

The main tool for reasoning about progress properties in Unit-B is the *transient operator* **tr**.

Definition 13 (tr operator) *For all state predicate p, $[\ \textbf{\textit{tr}}\, p \quad \equiv \quad \mathbf{G}\,\mathbf{F}\,;\bullet{\sim}p\ ]$.*

$\textbf{tr}\, p$ states that state predicate p is infinitely often false. The relationship between **tr** and $\leadsto$ is as follows:

$$p \ \leadsto\ {\sim}p \quad = \quad \mathbb{1} \ \leadsto\ {\sim}p \quad = \quad \textbf{tr}\, p\ . \qquad (26)$$

The attractiveness of properties such as $\textbf{tr}\, p$ is that we can *implement* these using a single event as follows.

Theorem 2 (Implementing tr) *Consider a Unit-B model $\mathbf{M}$ and a transient property $\textbf{\textit{tr}}\, p$. We have $[\ ex.\mathbf{M} \Rightarrow \textbf{\textit{tr}}\, p\ ]$, if there exists an event*

$$\mathsf{e} \mathrel{\widehat{=}} \ \textbf{any}\ t\ \textbf{where}\ g.t.v\ \textbf{during}\ c.t.v\ \ \textbf{upon}\ f.t.v\ \ \textbf{then}\ s.t.v.v'\ \ \textbf{end}\ \ ,$$

that is to say $ex.\mathbf{M}$ entails:

$$\mathbf{G}\,(\mathbf{G}\ \bullet c \land \mathbf{G}\,\mathbf{F}\,;\bullet f \ \Rightarrow\ \mathbf{F}\,;f\,;act.(\mathsf{e}.t))\ , \qquad \text{(LIVE)}$$

and parameter t such that $\mathsf{e}.t \in \mathbf{M}$ and $ex.\mathbf{M}$ entails each of the conditions below:

$$\mathbf{G}\ \bullet(p \Rightarrow c)\ , \qquad \text{(SCH)}$$

$$c \leadsto f\ , \qquad \text{(PRG)}$$

$$\mathbf{G}\ (\ (p \land c \land f)\,;act.(\mathsf{e}.t)\,;\mathbf{true} \ \ \Rightarrow\ \ \mathbf{X}\,;\bullet{\sim}p\ )\ . \qquad \text{(NEG)}$$

Proof In this case, $\mathbf{G}$ acts as an everywhere operator which allows us to prove $\mathbf{F};\bullet \sim p$ instead of $\mathbf{G}\,\mathbf{F};\bullet \sim p$. Additionally, since $[\ \neg s \Rightarrow s \quad \equiv \quad s\ ]$ for any computation predicate s, we discharge our proof obligation by strengthening $\mathbf{F}\ ; \bullet \sim p$ to its negation, $\mathbf{G}\ \bullet p$.

$$\begin{array}{ll}
 & \mathbf{F} ; \bullet\!\sim\! p \\
\Leftarrow & \quad \{\ [\ \mathbf{F} ; \mathbf{X} \Rightarrow \mathbf{F}\ ], \text{aiming for (NEG)}\ \} \\
 & \mathbf{F} ; \mathbf{X} ; \bullet\!\sim\! p \\
\Leftarrow & \quad \{\ \text{(NEG)}\ \} \\
 & \mathbf{F} ; (p \wedge c \wedge f) ; act ; \mathbf{true} \\
\Leftarrow & \quad \{\ \text{computation calculus}\ \} \\
 & \mathbf{F};f;act;\mathbf{true}\ \wedge\ \mathbf{G}\ \bullet c\ \wedge\ \mathbf{G}\ \bullet p \\
\Leftarrow & \quad \{\ \text{(LIVE); } \mathbf{G}\ \text{ is conjunctive}\ \} \\
 & \mathbf{G}\,\mathbf{F};\bullet f\ \wedge\ \mathbf{G}\ \bullet c\ \wedge\ \mathbf{G}\ \bullet p \\
= & \quad \{\ \text{(PRG)}\ \} \\
 & \mathbf{G}\ \bullet c\ \wedge\ \mathbf{G}\ \bullet p \\
= & \quad \{\ \mathbf{G}\ \text{ is conjunctive; (SCH)}\ \} \\
 & \mathbf{G}\ \bullet p
\end{array}$$

(Due to space restriction, for the rest of this paper, we only present sketch of proofs of theorems. Detailed proofs are available in [**?**]).

Condition (SCH) is an invariance property. Condition (PRG) is a progress property. Condition (NEG) states that event e.t establish $\sim\! p$ in one step. In practice, often we design c such that it is the same as p and f is $\mathbb{1}$ (i.e., omitting f); as a result, conditions (SCH) and (PRG) are trivial. Condition (NEG) can take into account any invariance property I and can be stated as $[\ (I \wedge p \wedge c \wedge f)\ ; act.(\mathrm{e}.t)\ \Rightarrow\ \mathbf{X};\sim\! p\ ]$.

In general, progress properties can be proved using the following *ensure-rule*. The rule relies on proving an unless property and a transient property.

Theorem 3 (The ensure-rule) *For all state predicates p and q,*

$$[\ (p\, \boldsymbol{un}\, q) \wedge (\boldsymbol{tr}\, p \wedge \sim q) \quad \Rightarrow \quad (p \leadsto q)\ ] \tag{27}$$

Proof (Sketch) $p\,\mathbf{un}\,q$ ensures that if p holds then it will hold for infinitely long or eventually q holds. If q holds eventually then we have $p \leadsto q$. Otherwise, if p holds for infinitely long and $\sim\! q$ also hold for infinitely long, we have a contradiction, since $\mathbf{tr}\, p \wedge \sim q$ ensures that eventually $p \wedge \sim q$ will be falsified. As a result, if p holds for infinitely long then eventually q has to hold.

3.4 Refinement

In this section, we develop rules for refining Unit-B models such that safety and liveness properties are preserved. Consider a machine $\mathbf{M}$ and a machine $\mathbf{N}$, $\mathbf{N}$ refines $\mathbf{M}$ if

$$[\ ex.\mathbf{N} \Rightarrow ex.\mathbf{M}\ ]\ . \tag{REF}$$

As a result of this definition, any property of $\mathbf{M}$ is also satisfied by $\mathbf{N}$. Similarly to Event-B, refinement is considered in Unit-B on a per event basis. Consider an abstract event $\mathsf{e}.t$ belong to $\mathbf{M}$ and a concrete event $\mathsf{f}.t$ belong to $\mathbf{N}$ as follows.

$$\mathsf{e} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } g.t.v \textbf{ during } c.t.v \ \textbf{ upon } f.t.v \ \textbf{ then } s.t.v.v' \ \textbf{ end} \tag{28}$$

$$\mathsf{f} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } h.t.v \textbf{ during } d.t.v \ \textbf{ upon } e.t.v \ \textbf{ then } r.t.v.v' \ \textbf{ end} \tag{29}$$

We have $\mathsf{f}.t$ is a refinement of $\mathsf{e}.t$ if

$$[\ ex.\mathbf{N} \ \Rightarrow \ (act.(f.t) \Rightarrow act.(e.t))\ ]\text{ , and} \tag{EVT-SAF}$$

$$[\ ex.\mathbf{N} \ \Rightarrow \ (sched.(f.t) \Rightarrow sched.(e.t))\ ] \tag{EVT-LIVE}$$

A similar rule is presented for the initialisation. The proof that $\mathbf{N}$ refines $\mathbf{M}$ (i.e., (REF)) given conditions such as (EVT-SAF) and (EVT-LIVE) is left out. A special case of event refinement is when the concrete event f is a new event. In this case, f is proved to be a refinement of a special SKIP event which is unscheduled and does not change any abstract variables.

Condition (EVT-SAF) leads to similar proof obligations in Event-B such as *guard strengthening* and *simulation*. We focus here on expanding the condition (EVT-LIVE). The subsequent theorems are related to concrete event f (29) and abstract event e (28), under the assumption that condition (EVT-SAF) has been proved. 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*.

Theorem 4 (Weakening the coarse-schedule) *Given* $e = f$. *If*

$$[\ ex.\mathbf{N} \quad \Rightarrow \quad \mathbf{G} \bullet (c \Rightarrow d)\ ] \quad \textit{then} \quad [\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]\ .$$

Proof (Sketch) The coarse-schedule is at an anti-monotonic position within the definition of $sched$.

Theorem 5 (Replacing the coarse-schedule) *Given* $e = f$. *If*

$$[\ ex.\mathbf{N} \ \Rightarrow \ c \ \rightsquigarrow \ d\ ] \tag{30}$$

$$[\ ex.\mathbf{N} \ \Rightarrow \ d \ \textbf{\textit{un}} \ \sim c\ ]\ , \tag{31}$$

then $[\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]$

Proof (Sketch) Conditions (30) and (31) ensures that if c holds then eventually d holds and it will hold for at least as long as c As a result, if c holds for infinitely long, d also holds for infinitely long. Hence the new schedule ensures that f occurs at least on those cases where e has to occur.

Theorem 6 (Strengthening the fine-schedule) *Given* $d = c$. *If*

$$[\ ex.\mathbf{N} \ \Rightarrow \ \mathbf{G} \bullet (e \Rightarrow f)\ ]\ \textit{, and} \tag{32}$$

$$[\ ex.\mathbf{N} \ \Rightarrow \ f \ \rightsquigarrow \ e\ ] \tag{33}$$

then $[\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]$.

Proof (Sketch) We can prove $sched.(\mathsf{e}.t)$ under the assumptions $sched.(\mathsf{f}.t)$ and $ex.\mathbf{N}$ by calculating from $\mathbf{F}\,;(c \wedge f)\,; act.(\mathsf{e}.t)\,;\mathbf{true}$ (the right hand side of $sched.(\mathsf{e}.t)$) and applying one assumption after the other (in this order (32), (EVT-SAF), $sched.(\mathsf{f}.t)$, (33)) to strengthen it to $\mathbf{G}\ \bullet c \wedge \mathbf{G}\,\mathbf{F}\,; \bullet f$ (the right hand side of $sched.(\mathsf{e}.t)$).

Theorem 7 (Removing the fine-schedule) *Given* $d = c$ *and* $e = \mathbb{1}$. *If*

$$[\ ex.\mathbf{M} \quad \Rightarrow \quad \mathbf{G} \bullet (c \Rightarrow f)\ ] \tag{34}$$

then $[\ ex.\mathbf{N} \Rightarrow (sched.(\mathsf{f}.t) \Rightarrow sched.(\mathsf{e}.t))\ ]$.

Proof (Sketch) Condition (34) ensures that when c holds for infinitely long, f holds for infinitely long, hence we can remove the fine-schedule f, i.e., replaced it by $\mathbb{1}$.

4 Example: A Signal Control System

We illustrate our method by applying it to design a system controlling trains at a station [**?**]. 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 1. 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.

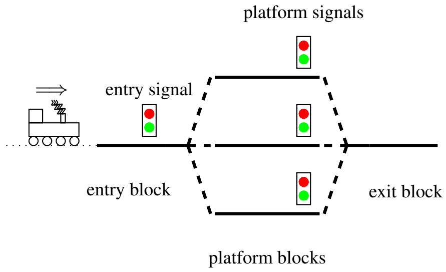

Fig. 1 A signal control system

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 (REQ 2).

SAF 1 There is at most one train on each block

REQ 2 Each train in the network eventually leaves

Refinement strategy In the initial model, we abstractly model the trains in the network, focusing on REQ 2. In the first refinement, we introduce the topology of the network. We strengthen the model of the system, focusing on SAF 1 in the second refinement. In the third refinement, we introduce the signals and derive a specification for the controller that manages these signals.

4.2 Initial Model

In this initial model, we use a carrier set TRN to denote the set of trains and a variable $trns$ to denote the set of trains currently within the network. Initially $trns$ is assigned the empty set. At this abstract level, we have two events to model a train arriving at the station and a train leaving the station as below.

$$\begin{array}{l} \mathsf{arrive} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } t \in TRN \textbf{ then } trns := trns \cup \{t\} \textbf{ end} \\ \mathsf{depart} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } t \in TRN \textbf{ then } trns := trns \setminus \{t\} \textbf{ end} \end{array}$$

The requirement REQ 2 can be specified as a progress property prg0_1: $t \in trns \leadsto t \notin trns$. According to (26), prg0_1 is equivalent to prg0_2: **tr** $t \in trns$. In order to implement this transient property, we rely on Theorem 2 and add scheduling information for event depart as follows.

$$\mathsf{depart} \mathrel{\widehat{=}} \textbf{any } t \textbf{ where } t \in TRN \textbf{ during } t \in trns \textbf{ then } trns := trns \setminus \{t\} \textbf{ end}$$

Here, we design our depart event to implement the transient property prg0_2 such that conditions (SCH) and (PRG) are trivial. For condition (NEG), it is easy to prove that depart establishes the fact $t \notin trns$ in one step.

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

4.3 First Refinement

In this refinement, we first introduce the topology of the network in terms of blocks. We introduce a carrier set BLK and three constants $Entry$, PLF, $Exit$ denoting the entry block, platform blocks and exit block, respectively. A new variable loc is introduced denoting the location of trains in the network, constrained by invariant inv1_1: $loc \in trns \to BLK$.

For event depart, we strengthen the guard to state that a train can only leave from the exit block. 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 (see Figure 2). In order to prove the refinement of depart, we apply Theorem 5 (coarse-schedule replacing). In particular we need to prove the following conditions:

$$t \in trns \leadsto t \in trns \land loc.t = Exit \qquad (\mathsf{prg1\_1})$$

$$t \in trns \land loc.t = Exit \;\; \textbf{un} \;\; {\sim}(t \in trns) \qquad (\mathsf{un1\_2})$$

From now on, we focus on reasoning about progress properties, e.g., prg1_1, omitting the reasoning about unless properties, e.g., un1_2. The readers should be convinced that using Theorem 1, these unless properties are valid for our model. We first apply (Split-Off-Skip) to obtain $t \in trns \land loc.t \neq Exit \leadsto t \in trns \land loc.t = Exit$ and then apply the transitivity property (Transitivity) of the leads-to operator to establish two progress properties prg1_3 and prg1_4 as follows.

$$t \in trns \land loc.t \neq Exit \leadsto t \in trns \land loc.t \in PLF \qquad (\mathsf{prg1\_3})$$

$$t \in trns \land loc.t \in PLF \leadsto t \in trns \land loc.t = Exit \qquad (\mathsf{prg1\_4})$$

```
depart
 any $t$ where
  $t \in trns \land loc.t = Exit$
 during
  $t \in trns \land loc.t = Exit$
 then
  $trns := trns \setminus \{t\}$
  $loc := \{t\} \lhd loc$
 end

moveout
 any $t$ where
  $t \in trns \land loc.t \in PLF$
 during
  $t \in trns \land loc.t \in PLF$
 then
  $loc.t := Exit$
 end

movein
 any $t$ where
  $t \in trns \land loc.t = Entry$
 during
  $t \in trns \land loc.t = Entry$
 then
  $loc :\mid \langle \exists\, p :\ p \in PLF :\ loc' = loc \mathbin{\lhd\!\!-} \{t \mapsto p\} \rangle$
 end
```

Fig. 2 Events of the first refinement

Consider prg1_4, we first apply the ensure-rule (Theorem 3) to establish two properties (after simplification) as follows:

$$t \in trns \land loc.t \in PLF \ \ \textbf{un} \ \ t \in trns \land loc.t = Exit \qquad \text{(un1\_5)}$$

$$\textbf{tr}\ t \in trns \land loc.t \in PLF \qquad \text{(prg1\_6)}$$

We apply Theorem 2 to implement prg1_6 by the new event moveout which has a weakly-fair scheduling (see Figure 2). The proof that moveout implements prg1_6 is straightforward and therefore is omitted.

Similarly, for prg1_3, we apply the ensure-rule and implementing the resulting transient property, i.e., **tr** $t \in trns \land loc.t = Entry$, by event movein in Figure 2.

4.4 Second Refinement

In this refinement, we incorporate the safety requirement stating that there are no collisions between trains within the network, i.e. SAF 1. This is captured by invariant inv2_1 about loc: $\langle \forall\, t_1, t_2 :\ t_1, t_2 \in trns \land loc.t_1 = loc.t_2 :\ t_1 = t_2 \rangle$.

The guard of event moveout needs to be strengthened to maintain inv2_1. As a result, we need to modify the schedule information to ensure the *feasibility* condition (SCH-FIS) for Unit-B events stating that the schedules are stronger than the guard. In particular, we add (through strengthening) a fine-schedule to moveout (see Figure 3). 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 can move out of the platform.

We want to highlight the fact that moveout has both coarse- and fine-schedules. In particular, using only either weak- or strong-fairness would be unsatisfactory. Weak-fairness requires for the exit block to be remain free continuously in order for trains to move out. This assumption is not met by the current system. Strong-fairness allows a train to leave if the train 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. 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

```
moveout
any $t$ where
  $t \in trns \land loc.t \in PLF \land$
  $Exit \notin \operatorname{ran}.loc$
during
  $t \in trns \land loc.t \in PLF$
upon
  $Exit \notin \operatorname{ran}.loc$
then
  $loc.t := Exit$
end
```

```
movein
any $t$ where
  $t \in trns \land loc.t = Entry \land \langle \exists\, p\colon\ p \in PLF\colon\ p \notin \operatorname{ran}.loc \rangle$
during
  $t \in trns \land loc.t = Entry \land \langle \exists\, p\colon\ p \in PLF\colon\ p \notin \operatorname{ran}.loc \rangle$
then
  $loc :\mid \langle \exists\, p\colon\ p \in PLF \setminus \operatorname{ran}.loc\colon\ loc' = loc \lhd\!\!+ \{t \mapsto p\} \rangle$
end
```

Fig. 3 Events of the second refinement

that the train will be on its platform and that the exit block will become free but that both happen simultaneously infinitely often.

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.

In order to prove the refinement of moveout, we apply Theorem 6 (fine-schedule strengthening), which requires to prove the following progress property (remember that when an event has no fine schedules, it is assumed to be $\mathbb{1}$).

$$\mathbb{1} \leadsto Exit \notin \operatorname{ran}.loc \qquad \text{(prg2\_3)}$$

Property prg2_3 is equivalent to transient property prg2_4: **tr** $Exit \in \operatorname{ran}.loc$. We satisfy prg2_4 by applying the transient rule (Theorem 2) using event depart where the value for the parameter t is given by $loc^{-1}.Exit$, i.e., the train at the exit block. The proofs of conditions (SCH), (PRG), and (NEG) are straight-forward.

Finally we strengthen the guard of movein and subsequently strengthen its coarse-schedule (see Figure 3). We apply Theorem 5 (coarse-schedule replacing) movein. The detailed proof is omitted here.

4.5 Third Refinement

In this refinement, we introduce the signals associated with different blocks within the network. Variable sgn is introduced to denote the value of the signals associated with different blocks. We focus on the controlling of the platform signals here. In particular, invariants inv3_2 and inv3_3 state that if a platform signal is green (GR) then the exit block is free and the other platform signals are red (RD).

inv3_1 : $sgn \in \{Entry\} \cup PLF \to COLOR$
inv3_2 : $\langle \forall\, p\colon\ p \in PLF \land sgn.p = GR\colon\ Exit \notin \operatorname{ran}.loc \rangle$
inv3_3 : $\langle \forall\, p, q\colon\ p, q \in PLF \land sgn.p = sgn.q = GR\colon\ p = q \rangle$

We refine the moveout event using the platform signal as shown in Figure 4. The refinement of moveout is justified by applying Theorem 5 (coarse-schedule replacing) and Theorem 7 (fine-schedule removing). In particular, replacing the coarse-schedule requires the following transient property

$$\textbf{tr}\ t \in trns \land loc.t \in PLF \land sgn.(loc.t) = RD\ . \qquad \text{(prg3\_5)}$$

```
moveout
 any $t$ where
  $t \in trns \land loc.t \in PLF \land$
  $sgn.(loc.t) = GR$
 during
  $t \in trns \land loc.train \in PLF \land$
  $sgn.(loc.t) = GR$
 then
  $loc.t := Exit$
  $sgn.(loc.t) := RD$
 end
```

```
ctrl_platform
 any $p$ where
  $p \in PLF \land p \in \text{ran}\,.loc \land Exit \notin \text{ran}\,.loc \land$
  $\langle \forall\, q\colon\ q \in PLF\colon\ sgn.q = RD \rangle$
 during
  $p \in PLF \land p \in \text{ran}\,.loc \land sgn.p = RD$
 upon
  $Exit \notin \text{ran}(loc) \land \langle \forall\, q\colon\ q \in PLF \land q \neq p\colon\ sgn.q = RD \rangle$
 then
  $sgn.p := GR$
 end
```

Fig. 4 Events of the third refinement

In order to satisfy prg3_5, we introduce a new event ctrl_platform for the controller to change a platform signal to green according to Theorem 2 (see Figure 4). This event ctrl_platform is a specification for the system to control the platform signals preserving both safety and liveness properties of the system. In particular, the scheduling information states that if (1) a platform is occupied and the platform signal is RD infinitely long and (2) the exit block is unoccupied and the other platform signals are all RD infinitely often, then the system should eventually set this platform signal to GR. The refinement of event movein and how the entry signal is controlled is similar and omitted.

5 Conclusion

We presented in this paper 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 refinement rules to manipulate the coarse- and fine-schedules such that liveness properties are preserved. 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 some unsafe state 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. Often, to ensure the validity of a refinement, one needs to prove some progress properties which (eventually) can be implemented (satisfied) by some scheduled events.

Related work Unit-B and Event-B differ mainly in the scheduling assumptions. In Event-B, event executions are assumed to satisfy the *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* [**?**]. However, this work does not discuss how the refinement notion can be adapted to preserve liveness properties. Moreover, the minimum progress assumption is often either too weak to prove liveness properties or, when it is not, make the proofs needlessly complicated.

TLA+[**?**] is another formal method based on refinement supporting liveness properties. The execution of a TLA+ model is also captured as a formula with safety and liveness sub-formulae. However, refinement of the liveness part in TLA+ involves calculating explicitly the fairness assumptions of the abstract and concrete models. In our opinion, this is not practical for developing realistic systems. The lack of practical rules for refining the liveness part in TLA+ might stem from the view of the author of TLA+ concerning the *unimportance of liveness* [**?**, Chapter 8]. In our opinion, liveness properties are as important as safety properties to design correct systems.

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.

Given the close relationship between Unit-B and Event-B, we are looking at extending the supporting Rodin platform [**?**] of Event-B to accomodate Unit-B. We expect to generate the corresponding proof obligations according to different refinement rules such that it can be verified using the existing provers of Rodin.