Managing LTL properties in Event-B refinement

Steve Schneider[1], Helen Treharne[1],
Heike Wehrheim[2], and David M. Williams[3]

[1] University of Surrey, England, UK
[2] University of Paderborn, Germany
[3] VU University Amsterdam

Abstract. Refinement in Event-B supports the development of systems via proof based step-wise refinement of events. This refinement approach ensures safety properties are preserved, but additional reasoning is required in order to establish liveness and fairness properties. In this paper we present results which allow a closer integration of two formal methods, Event-B and linear temporal logic. In particular we show how a class of temporal logic properties can carry through a refinement chain of machines. Refinement steps can include introduction of new events, event renaming and event splitting. We also identify a general liveness property that holds for the events of the initial system of a refinement chain. The approach will aid developers in enabling them to verify linear temporal logic properties at early stages of a development, knowing they will be preserved at later stages. We illustrate the results via a simple case study.

1 Introduction

Event-B [1] is a step-wise development method with excellent tools: Rodin platform [2] providing proof support and ProB [11] providing model checking. As Hoang and Abrial [10] clearly state the focus of verification within Event-B has been on the safety properties of a system to ensure that "something (bad) never happens". Typically, this has been done via the discharging of proof obligations. Nonetheless, the use of linear temporal logic (LTL) to specify temporal liveness properties has also been prevelant, for example in its application within the ProB tool [12]. The challenge is to identify more natural ways of integrating Event-B and LTL, so that LTL properties can be preserved by Event-B refinement, which is not currently the case in general.

Event-B describes systems in terms of *machines* with state, and *events* which are used to update the state. Events also have *guards*, which are conditions for the event to be enabled. One (abstract) machine may be refined by another (concrete) machine, using a *refinement step*. A *linking invariant* captures how the abstract and concrete states are related, and each abstract event must be refined by one or more concrete events whose state transformations match the abstract one in the sense of preserving the linking invariant. Refinement is transitive, so a sequence of refinement steps, known as a *refinement chain*, will result in a concrete machine which is a refinement of the original abstract one.

A particular feature provided by Event-B is the introduction of *new* events in a refinement step—events which do not refine any abstract event. This allows for refinements to add finer levels of granularity and concretisation as the design develops; there are many examples in [1]. These new events are invisible at the abstract level (they correspond to the abstract state not changing), and we generally need to verify that they cannot occur forever. Event-B makes use of *labels* to keep track of the status of events as a refinement chain progresses. Event-B labels are *anticipated*, *convergent* and *ordinary*. The labelling of events in Event-B form part of the core of a system description but their inclusion is primarily to support the proof of safety properties and ensuring that events cannot occur forever: convergent events must decrease a variant and anticipated events cannot increase it. In this paper all newly introduced events must be convergent or anticipated, and anticipated events must become convergent at some stage. As an initial example, consider a *Lift* machine with two events *top* and *ground*, representing movement to the top and to the ground floor. This can be refined by a machine $Lift'$ introducing two new anticipated events *openDoors* and *closeDoors*. The events *top* and *ground* are blocked when the doors are open, but enabled when the doors are closed.

Linear temporal logic provides a specification language for capturing properties of executions of systems, and is appropriate for reasoning about liveness and fairness. For example, we might verify for *Lift* that whenever *top* occurs, then eventually *ground* will occur. However, this is not guaranteed for its refinement $Lift'$: it may be that the doors open and close repeatedly forever following the *top* event, thus never reaching the next *ground* event. Alternatively it may be that the system deadlocks with the doors open, again preventing *ground* from occurring. Hence we see that LTL properties are not automatically preserved by Event-B refinement. In the first case we would require some assurance that *openDoors* and *closeDoors* cannot repeat forever without the lift moving; in the second case we require some liveness property on *closeDoors* to prevent termination with the doors open.

In this paper we present results for when temporal logic properties can be carried through Event-B refinement chains. The results generalise to events that are split—refined by several events—during a refinement chain. We also identify conditions on temporal logic properties that make them suitable for use in a refinement chain, since some properties are not preserved by Event-B refinement (for example, the property "*closeDoor* never occurs" holds for *Lift* but not for its refinement $Lift'$). The results are underpinned by our process algebra understanding of the Event-B semantics, in particular the traces, divergences and infinite traces semantics used for CSP and applied to Event-B in [15].

The paper is organised as follows: Section 2 provides the necessary Event-B refinement background and the refinement strategy we use in the paper. Section 3 introduces a running example. Section 4 defines the LTL we use. Sections 5 and 6 present and illustrate the main theoretical results. For reasons of space we do not include proofs, but they appear in the technical report available at [16]. We

put our work into the context of related work in Section 7 and our future work in Section 8.

2 Event-B

2.1 Event-B Machines

An Event-B development is defined using *machine*s. A machine M contains a vector of variables and a set of events. The *alphabet* of M, αM, is the set of events defined in M. Each event evt_i has the general form $evt_i \mathrel{\widehat{=}}$ **any** x **where** $G_i(x, v)$ **then** $v :| BA_i(v, x, v')$ **end**, where x represents the parameters of the event, the guard $G_i(x, v)$ is the condition for the event to be enabled. The body is given by $v :| BA_i(v, x, v')$ whose execution assigns to v any value v' which makes the *before-after* predicate $BA_i(v, x, v')$ true. This simplifies to $evt_i \mathrel{\widehat{=}}$ **when** $G_i(v)$ **then** $v :| BA_i(v, v')$ **end** when there are no parameters, since the guard and the *before-after* predicate does not refer to the parameters x.

Variables of a machine are initialised in an initialisation event *init* and are constrained by an invariant $I(v)$. The Event-B approach to semantics is to associate proof obligations with machines. The key proof obligation, `INV`, is that all events must preserve the invariant. There is also a proof obligation on a machine with respect to deadlock freedom which means that a guard of at least one event in M is always enabled. When this obligation holds M is *deadlock free*.

2.2 Event-B Refinement

An Event-B development is a sequence of B machines linked by a refinement relationship. In this paper we use M and M' when referring to a refinement between an *abstract* machine M and a *concrete* machine M' whereas a chain of refinements is referred to using numbered subscripts, i.e., M_0, M_i, ..., M_n, to represent the specific refinement levels.

A refinement machine can introduce new events and split existing events. We omit the treatment of merging events in this paper. New events are treated as refinements of *skip*, i.e., evt'_i does not refine an event in M. Note that when splitting events, M' has several events evt'_i refining a single event evt_i.

A machine M is considered to be refined by M' if the given *linking invariant* J on the variables between the two machines is established by their initialisation, and preserved by all events. This requirement is captured by the `INV_REF` proof obligation. Formally, we denote the refinement relation between two machines, written $M \preccurlyeq M'$, when all the following proof obligations hold: feasibility `FIS_REF`, guard strengthening `GRD_REF` and simulation `INV_REF`. Feasibility of an event is the property that, if the event is enabled (i.e., the guard is true), then there is some after-state. Guard strengthening requires that when a concrete event is enabled, then so is the abstract one. Finally, simulation ensure the occurrence of events in the concrete machine can be matched in the abstract one (including the initialization event). Further details of these proof obligations can be found in [1].

In Section 1 we introduced the three kinds of labelling of events in Event-B: *anticipated* (a), *convergent* (c) and *ordinary* (o) and noted that convergent events are those which must not execute forever whereas *anticipated* events provide a means of deferring consideration of divergence-freedom until later refinement steps. The proof obligation that deals with divergences is WFD_REF. It requires that the proposed variant V of a refinement machine satisfies the appropriate properties: that it is a natural number, that it decreases on occurrence of any convergent event, and that it does not increase on occurrence of any anticipated event. Therefore, we augment the previous refinement relation with WFD_REF such that $M \preccurlyeq_W M'$. Ordinary events can occur forever and therefore WFD_REF is not applicable for such events.

2.3 Event-B Development strategies

Event-B has a strong but flexible refinement strategy which is described in [9]. In [15] we also discussed different Event-B refinement strategies and characterised them with respect to the approaches documented by Abrial in [1] and supported by the Rodin tool. In this paper we focus on the simplest strategy, and the one most commonly used. The strategy has the following set of restrictions on a refinement chain $M_0 \preccurlyeq_W M_1 \preccurlyeq_W \ldots \preccurlyeq_W M_n$:

1. all events in M_0 are labelled ordinary. This set of events is referred to as O_0.
2. each event of M_i is refined by at least one event of M_{i+1};
3. each new event in M_i is either anticipated or convergent, where $i > 0$;
4. each event in M_{i+1} which refines an anticipated event of M_i is itself either convergent or anticipated;
5. refinements of convergent or ordinary events of M_i are ordinary in M_{i+1}.
6. no anticipated events remain in the final machine.

Figure 1 illustrates our development strategy for a vending machine, detailed in Section 3, where C_i is the set of convergent events within M_i, and O_i is the set of ordinary events within M_i.

For example, $O_0 = \{selectBiscuit, selectChoc, dispenseBiscuit, dispenseChoc\}$ and $C_0 = \emptyset$ in VM_1. In VM_2 we note that $C_1 = \{refund\}$. In VM_3 we note that $C_2 = \{refill\}$ and in VM_4 we have $C_3 = \{pay\}$. Thus we denote $C_{all} = C_1 \cup C_2 \cup C_3$.

2.4 Event-B Semantics

In this paper we define a trace of M to be either an infinite sequences of events (a,c or o), i.e., $\langle e_0, e_1, \ldots \rangle$ or a finite sequence of events, i.e., $\langle e_0, \ldots, e_{k-1} \rangle$ where the machine M deadlocks after the occurrence of the final event. Traces correspond to maximal executions of machines. Plagge and Leuschel in [14] provided a definition of an infinite or finite path π of M in terms of a sequence of events and their intermediate states. In order to distinguish notation we use u to represent a trace without the intermediate states. We need not consider the particular

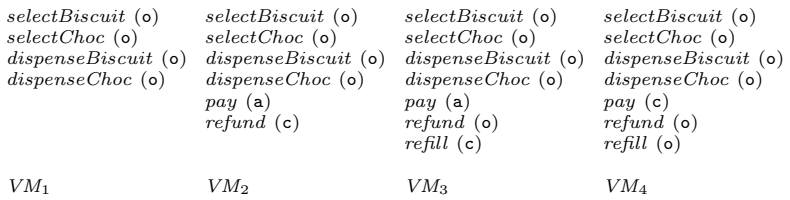

Fig. 1: Events and their annotations in the Vending Machine development

states within a trace in our reasoning which is based on infinite traces. When a machine M is deadlock free all of its traces are infinite. We use the functions of concatenation ($\frown$), projection ($\restriction$) and length ($\#$) on finite and infinite traces.

A more complex behavioural semantics for B machines was given by Schneider *et al.* in [15] based on the weakest precondition semantics of [13, 6] for action systems and CSP. In [15] there are two key results that enable us to reason about infinite sequences of convergent and ordinary events in this paper. Firstly, the following predicate captures that if an infinite trace u performs infinitely many events from C then it has infinitely many events from O, where C and O are sets of events.

Definition 1. $CA(C, O)(u) \mathrel{\widehat{=}} (\#(u \restriction C) = \infty \Rightarrow \#(u \restriction O) = \infty)$

C and O will be used to capture convergent and ordinary events through a development. For an Event-B machine M the above means that it *does not diverge on its C events*. This is precisely what we get when we prove `WFD_REF` but the above definition describes the result on traces.

The second result from [15], restated as Theorem 1, allows us to conclude that there are no infinite sequences of convergent events in the final machine of a refinement chain M_n. The function $g_{1,n}$ defines a compositional mapping for all concrete events to abstract events in terms of a function mapping f at each refinement level where $f_{i+1} : \alpha M_{i+1} \twoheadrightarrow \alpha M_i$ and $f_{i+1}(evt_{i+1}) = evt_i \Leftrightarrow evt_{i+1}$ **refines** evt_i. (Note that $g_{1,0}$ is the identity function.)

Definition 2. $g_{i,j} = f_j;\ f_{j-1};\ \ldots;\ f_i$

Theorem 1. *If* $M_0 \preccurlyeq_W M_1 \preccurlyeq_W \ldots \preccurlyeq_W M_n$ *then*

$$M_n \textbf{ sat } CA(g_{1,n}^{-1}(C_0) \cup \ldots \cup g_{i,n}^{-1}(C_i) \cup \ldots \cup C_n\ ,\ g_{1,n}^{-1}(O_0))$$

The result for our example is simply VM_4 **sat** $CA(C_{all}, O_0)$ since there is no renaming: each function mapping f_i is the identity.

3 Example

In Section 2.3 we introduced a development strategy for a vending machine. Figures 2, 3, 4 and 5 illustrate a development chain from vending machine VM_1,

machine VM_1
variables *chosen*
invariant $chosen \subseteq \{choc, biscuit\}$
events
  init $\widehat{=}$ $chosen := \{\}$
  selectBiscuit $\widehat{=}$ **status** : ordinary
    when $biscuit \notin chosen$ **then** $chosen := chosen \cup \{biscuit\}$ **end**
  selectChoc $\widehat{=}$ **status** : ordinary
    when $choc \notin chosen$ **then** $chosen := chosen \cup \{choc\}$ **end**
  dispenseBiscuit $\widehat{=}$ **status** : ordinary
    when $biscuit \in chosen$ **then** $chosen := chosen - \{biscuit\}$ **end**
  dispenseChoc $\widehat{=}$ **status** : ordinary
    when $choc \in chosen$ **then** $chosen := chosen - \{choc\}$ **end**
end

Fig. 2: VM_1

VM_2, VM_3 to VM_4; there are no anticipated events in VM_4. Note the numbers of the vending machines start from one. We introduce VM_0 in Section 6. Thus M_0 in Theorem 1 corresponds to VM_1 etc.

VM_1 is a simple machine that supports the selection and dispensing of chocolates and biscuits via four events: *selectBiscuit*, *selectChoc*, *dispenseBiscuit* and *dispenseChoc*. We abbreviate their names in the narrative to *sb*, *sc*, *db* and *dc* respectively. The first refinement step introduces VM_2 and the notion of paying and refunding. The *pay* event in VM_2 is always enabled and allows positive credit to be input. The machine allows a biscuit to be chosen if it has not already been chosen and additionally provided a payment has been made; a chocolate selection is similar. Hence the guards of all four of the original events *sb*, *sc*, *db* and *db* are strengthened. The guard of the *refund* event means that credit cannot be refunded for selected items and cannot occur forever since it is convergent. Importantly, the *refundEnabled* flag is introduced so that it is only true after a dispense and prevents infinite loops of the *pay* followed by *refund*.

VM_3 introduces the notion of stocked items and a new *refill* event. We could have chosen many different guards for the *refill* event. For example, we could have labelled it *anticipated* with a guard of *true*. Instead we have made an underspecification where the stock can be restocked when there may be no biscuits or no chocolates, and established convergence. Again the guard of the four original events have been strengthened so that they are only enabled when the appropriate stocked item is in stock. But now *db* and *dc* also capture the nondeterministic notion of running out or not of their respective items. The guard of *refund* remains unchanged. The guard of *pay* has been strengthened so that it is only enabled when there is stock but this is not strong enough to prevent it happening infinitely often, hence it remains anticipated in VM_3.

machine VM_2
variables *credit*, *chosen*, *refundEnabled*
invariant $credit \in \mathbb{N} \wedge chosen \subseteq \{choc, biscuit\} \wedge refundEnabled \in BOOL$
variant *if* $refundEnabled = FALSE$ *then* 0 *else* 1
events
init $\widehat{=}$ $credit := 0 \parallel chosen := \{\} \parallel refundEnabled := FALSE$
pay $\widehat{=}$ **status** : anticipated
any x **where** $x \in \mathbb{N}_1$
then $credit := credit + x$ **end** $\parallel$ $refundEnabled := FALSE$ **end**
selectBiscuit $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge biscuit \notin chosen \wedge credit > card(chosen)$
then $chosen := chosen \cup \{biscuit\}$ **end**
selectChoc $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge choc \notin chosen \wedge credit > card(chosen)$
then $chosen := chosen \cup \{choc\}$ **end**
dispenseBiscuit $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge biscuit \in chosen$
then $credit := credit - 1 \parallel chosen := chosen - \{biscuit\} \parallel$
$refundEnabled := TRUE$ **end**
dispenseChoc $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge choc \in chosen$
then $credit := credit - 1 \parallel chosen := chosen - \{choc\} \parallel$
$refundEnabled := TRUE$ **end**
refund $\widehat{=}$ **status** : convergent
when $credit > card(chosen) \wedge refundEnabled := TRUE$
then $credit := card(chosen) \parallel refundEnabled := FALSE$ **end**
end

Fig. 3: VM_2

The final machine, VM_4, is a straightforward data refinement which introduces the capacity of the machine. Apart from highlighting the refinement relationship between *stocked* and *chocStock* and *biscuitStock* note the strengthening of the guard of *refill* so that vending machine should only be refilled when there is no stock. Also the guard of *pay* is strengthened so that it becomes convergent.

4 LTL notation

In this paper we use the grammar for the LTL operators presented by Plagge and Leuschel [14]:

$$\phi ::= true \mid [x] \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \ U \ \phi_2$$

machine VM_3
variables *credit*, *chosen*, *refundEnabled*, *stocked*
invariant $credit \in \mathbb{N} \wedge chosen \subseteq \{choc, biscuit\} \wedge stocked \subseteq \{choc, biscuit\}$
$(choc \in chosen \Rightarrow choc \in stocked) \wedge (biscuit \in chosen \Rightarrow biscuit \in stocked)$
variant $card\{choc, biscuit\} - stocked$
events
init $\widehat{=} \ldots \parallel stocked := \{choc, biscuit\}$
pay $\widehat{=}$ **status** : anticipated
any x **where** $x \in \mathbb{N}_1 \wedge stocked \neq \emptyset$
then $credit := credit + x$ **end** $\parallel$ $refundEnabled := FALSE$ **end**
selectBiscuit $\widehat{=}$ **status** : ordinary
when $\ldots \wedge biscuit \in stocked$
then $chosen := chosen \cup \{biscuit\}$ **end**
selectChoc $\widehat{=}$ **status** : ordinary
when $\ldots \wedge choc \in stocked$
then $chosen := chosen \cup \{choc\}$ **end**
dispenseBiscuit $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge biscuit \in chosen \wedge biscuit \in stocked$
then $\ldots \parallel$ **any** x **where** $x \subseteq \{biscuit\}$ **then** $stocked := stocked - x$ **end end**
dispenseChoc $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge choc \in chosen \wedge choc \in stocked$
then $\ldots \parallel$ **any** x **where** $x \subseteq \{choc\}$ **then** $stocked := stocked - x$ **end end**
refund $\widehat{=}$ **status** : ordinary $\ldots$
refill $\widehat{=}$
status : convergent
when $choc \notin stocked \vee biscuit \notin stocked$
then $stocked := \{choc, biscuit\}$ **end**
end

Fig. 4: VM_3

A machine M satifies ϕ, denoted $M \models \phi$, if all traces of M satisfy ϕ. The definition for u to satisfy ϕ is defined by induction over ϕ as follows:

$$
\begin{array}{lll}
u \models true & & \\
u \models [x] & \Leftrightarrow & u = \langle x \rangle \frown u^1 \\
u \models \neg\phi & \Leftrightarrow & \text{it is not the case that } u \models \phi \\
u \models \phi_1 \vee \phi_2 & \Leftrightarrow & u \models \phi_1 \text{ or } u \models \phi_2 \\
u \models \phi_1 \, U \, \phi_2 & \Leftrightarrow & \exists\, k \geq 0. \forall\, i < k. u^i \models \phi_1 \text{ and } u^k \models \phi_2
\end{array}
$$

where u^n is u with the first n elements removed, i.e., $u = \langle x_0, \ldots, x_{n-1} \rangle \frown u^n$.

From these operators Plagge and Leuschel derived several additional operators, including: conjunction ($\phi_1 \wedge \phi_2$), finally (or eventually) ($F\phi$), and globally (or always) ($G\phi$), in the usual way; we also use these operators, and for explic-

machine VM_4
constants *capacity*
properties $capacity > 0$
variables *credit*, *chosen*, *refundEnabled*, *chocStock*, *biscuitStock*
invariant $credit \leq capacity \wedge chosen \subseteq \{choc, biscuit\} \wedge$
$refundEnabled \in BOOL \wedge chocStock \leq capacity \wedge biscuitStock \leq capacity \wedge$
$(choc \notin stocked \Rightarrow chocStock = 0) \wedge (choc \in stocked \Rightarrow chocStock \geq 0) \wedge$
$(biscuit \notin stocked \Rightarrow biscuitStock = 0) \wedge (biscuit \in stocked \Rightarrow biscuitStock \geq 0)$
variant $max\{(chocStock + biscuitStock) - credit), 0\}$
events
init $\widehat{=} \ldots \parallel chocStock := capacity \parallel biscuitStock := capacity$
pay $\widehat{=}$ **status** : convergent
any x **where** $x \in \mathbb{N}_1 \wedge (chocStock + biscuitStock) > credit$
then $credit := credit + x$ **end** $\parallel$ $refundEnabled := FALSE$ **end**
selectChoc $\widehat{=}$ **status** : ordinary
when $\ldots \wedge chocStock > 0$
then $chosen := chosen \cup \{choc\}$ **end**
selectBiscuit $\widehat{=}$ **status** : ordinary
when $\ldots \wedge biscuitStock > 0$
then $chosen := chosen \cup \{biscuit\}$ **end**
dispenseBiscuit $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge biscuit \in chosen \wedge biscuitStock > 0$
then $\ldots \parallel chocStock := chocStock - 1$ **end**
dispenseChoc $\widehat{=}$ **status** : ordinary
when $credit > 0 \wedge choc \in chosen \wedge chockStock > 0$
then $\ldots \parallel chocStock := chocStock - 1$ **end**
refund $\widehat{=}$ **status** : ordinary $\ldots$
refill $\widehat{=}$ **status** : ordinary
when $chocStock = 0 \wedge biscuitStock = 0$
then $chocStock := capacity \parallel biscuitStock := capacity$ **end**
end

Fig. 5: VM_4

itness we also provide direct definitions for them:

$$u \models \phi_1 \wedge \phi_2 \Leftrightarrow u \models \phi_1 \text{ and } u \models \phi_2$$
$$u \models F\phi \Leftrightarrow \exists\, i \geq 0 . u^i \models \phi$$
$$u \models G\phi \Leftrightarrow \forall\, i \geq 0 . u^i \models \phi$$

We omit atomic propositions on states since our traces are only dealing with events and not paths of states and transitions. We also omit the next operator, see Section 7. In this paper our running example uses globally, finally, or and implies.

For example, the informal specification for the *Lift* given in Section 1, that whenever *top* happens then eventually *ground* will happen, could be written as

$$G([top] \Rightarrow F[ground])$$

From our running *VM* example, the predicate $GF[selectBiscuit]$ expresses that *selectBiscuit* occurs infinitely often: at any point there is always some occurrence of *selectBiscuit* at some point in the future. We use this construction in the VM properties introduced in Section 5. For example, we have ϕ_2 given as

$$\phi_2 \quad = \quad (\neg GF[selectBiscuit]) \Rightarrow G([selectChoc] \Rightarrow F[dispenseChoc])$$

This states that provided *selectBiscuit* only occurs finitely often (i.e. eventually stops), then whenever *selectChoc* occurs then *dispenseChoc* will eventually occur.

It will also be useful to identify the events mentioned explicitly in an LTL formula ϕ. This set is called the alphabet of ϕ. This is written $\alpha(\phi)$, similar to the use of αM for the alphabet of machine M. For LTL formulae it is defined inductively as follows:

Definition 3.

$$\begin{aligned}
\alpha(true) &= \{\} \\
\alpha([x]) &= \{x\} \\
\alpha(\neg\phi) &= \alpha(\phi) \\
\alpha(\phi_1 \vee \phi_2) &= \alpha(\phi_1) \cup \alpha(\phi_2) \\
\alpha(\phi_1 \wedge \phi_2) &= \alpha(\phi_1) \cup \alpha(\phi_2) \\
\alpha(\phi_1 \ U \ \phi_2) &= \alpha(\phi_1) \ \cup \ \alpha(\phi_2) \\
\alpha(F\phi) &= \alpha(\phi) \\
\alpha(G\phi) &= \alpha(\phi)
\end{aligned}$$

For example, we have $\alpha(\phi_2) = \{selectBiscuit, selectChoc, dispenseChoc\}$ for ϕ_2 above.

5 Preserving LTL properties

In this section we provide results to demonstrate when properties are preserved by refinement chains. Firstly, we consider chains which do not contain any renaming/splitting of events in a machine. Hence, each function mapping f_i for $M_i \ldots M_n$ is the identity. The first result is a general result identifying a particular temporal property that will always hold for all refinement chains which abide by the rules of the strategy presented in Section 2.3. The second result given in Lemma 2 concerns the preservation of temporal properties that would be proposed by a specifier. We have already observed from the vending machine example that new events can be introduced during a refinement, e.g., *pay*, *refill*, etc.. We aim for such properties to hold even though new anticipated and convergent events are being introduced.

Lemma 1 states that M_n at the end of the refinement chain will always eventually perform one of the events of the initial machine M_0. In other words, M_n will perform infinitely many of the initial events. This means that the events introduced along the refinement chain cannot occur forever at the expense of the original events. In our example, $\alpha M_0 = O_0$.

Lemma 1. *If* $M_0 \preccurlyeq_W M_1 \preccurlyeq_W \ldots \preccurlyeq_W M_n$ *and* M_n *is deadlock free and* M_n *does not contain any anticipated events then* $M_n \models GF(\bigvee_{e \in \alpha M_0} [e])$

Next we provide a definition which is used in Lemma 2 below and it enables us to gain insights into the kinds of temporal properties that are appropriate to be proposed and have the potential of being preserved through a refinement chain. Definition 4 describes a maximal execution satisfying a property ϕ. The execution may include some events which do not have an impact on whether the property holds or not therefore we can restrict the maximal execution to include only those events that impact on the property.

Definition 4. *Let* β *be a set of events. Then* ϕ *is* β*-dependent if* $\alpha(\phi) \subseteq \beta$ *and* $u \models \phi \Leftrightarrow (u \upharpoonright \beta) \models \phi$.

An example of a β-dependent property is $GF(pay)$ where $\beta = \{pay\}$. If $u \models GF(pay)$ then $u \upharpoonright pay \models GF(pay)$, and vice versa. Conversely, $\neg G(pay)$ is not $\{pay\}$-dependent. For example, if $u = \langle pay, refill, pay, pay, \ldots \rangle$ then $u \models \neg G(pay)$ but $u \upharpoonright \{pay\} \not\models \neg G(pay)$.

As another example, define $\beta = \{sb, sc, db, dc\}$. Then $G(sb \vee sc \vee db \vee dc)$ is not β-dependent. This is exemplified by any trace u which contains events other than those in β. In this case $u \upharpoonright \{sb, sc, db, sc\} \models G(sb \vee sc \vee db \vee dc)$ but $u \not\models G(sb \vee sc \vee db \vee dc)$. VM_4 exhibits such traces. Observe that this property holds for VM_1 but not for VM_4: it is not preserved by refinement. Since it is not β-dependent Lemma 2 below is not applicable for this property.

Our main result for this section identifies conditions under which an LTL property ϕ will be preserved in a refinement chain. The conditions are as follows:

- by the end of the refinement chain there should be no outstanding anticipated events (and so all newly introduced events have been shown to be convergent), as given by restriction 6 of the Development Strategy of Section 2.3;
- the final machine in the refinement chain must be deadlock-free; and
- all of the events that have an effect on whether or not ϕ is true are already present in M_i (ϕ is β-dependent for some $\beta \subseteq \alpha M_i$).

These conditions are enough to ensure that ϕ is preserved through refinement chains. This means that M_i can be checked for ϕ, and we can be sure that the resulting system M_n will also satisfy it.

The lemma is formally expressed as follows:

Lemma 2. *If* $M_i \models \phi$ *and* $M_i \preccurlyeq_W \ldots \preccurlyeq_W M_n$ *and* $0 \leq i < n$ *and* M_n *is deadlock free and* M_n *does not contain any anticipated events and* ϕ *is* β*-dependent and* $\beta \subseteq \alpha M_i$ *then* $M_n \models \phi$.

5.1 Preserving Vending Machine properties

We consider the application of the above Lemmas to our running example on the refinement chain

$$VM_1 \preccurlyeq_W VM_2 \preccurlyeq_W VM_3 \preccurlyeq_W VM_4$$

In this case we obtain immediately from Lemma 1 that

$$VM_4 \models GF([selectBiscuit] \vee [selectChoc] \vee [dispenseBiscuit] \vee [dispenseChoc])$$

Any execution of VM_4 will involve infinitely many occurrences of some of these events. The newly introduced events *pay*, *refund*, *refill* cannot be performed forever without the occurrence of the original events.

We consider some further properties to illustrate the applicability of Lemma 2. Taking VM_1 to be the first machine in the refinement chain, we can consider the following temporal properties ϕ for VM_1:

$$\phi_1 = G([selectChoc] \vee [selectBiscuit] \Rightarrow F([dispenseChoc] \vee [dispenseBiscuit]))$$
$$\phi_2 = (\neg GF[selectBiscuit]) \Rightarrow G([selectChoc] \Rightarrow F[dispenseChoc])$$
$$\phi_3 = (\neg GF[selectChoc]) \Rightarrow G([selectBiscuit] \Rightarrow F[dispenseBiscuit])$$
$$\phi_4 = G([selectChoc] \Rightarrow F[dispenseChoc])$$
$$\phi_5 = G([selectBiscuit] \Rightarrow F[dispenseBiscuit])$$

We note that each of the properties are β-dependent. Next we consider whether $VM_1 \models \phi_i$ for each $i \in 1..5$. Note that in fact $VM_1 \not\models \phi_4$ and $VM_1 \not\models \phi_5$ since there is a trace for which the properties fail, e.g., in the case of ϕ_4 the $\langle sc, sb, db, sb, db, \ldots \rangle$ we could have an infinite loop of sb, db events and never reach a *dc* event. Thus Lemma 2 is not applicable to these properties.

The properties ϕ_2 and ϕ_3 are the strongest; ϕ_2 states that if you do not always have an *sb* then you will be able to choose a chocolate and for it to be dispensed, and the dual applies in ϕ_3. Once we have also established the refinement chain $VM_1 \preccurlyeq_W VM_2 \preccurlyeq_W VM_3 \preccurlyeq_W VM_4$, and that VM_4 is deadlock free we can deduce using Lemma 2 that $VM_4 \models \phi_i$ for all $i \in 1..3$. Observe however that Lemma 2 does not establish that ϕ_i holds in all refinement machines, only those with no anticipated events. For example, VM_2 and VM_3 do not satisfy ϕ_1, ϕ_2 nor ϕ_3 since *pay* is anticipated and can be executed infinitely often.

Since VM_2 introduced the event *pay* we can also introduce new temporal properties that are required to hold from VM_2 onwards. In other words, we apply Lemma 2 on the chain $VM_2 \preccurlyeq_W VM_3 \preccurlyeq_W VM_4$. The properties to consider are:

$$\phi_6 = G([pay] \Rightarrow F([dispenseBiscuit] \vee [dispenseChoc]))$$
$$\phi_7 = GF[pay]$$

The infinite behaviour of *pay* means that ϕ_6 is not satisfied in VM_2. However, $VM_2 \models \phi_7$ thus we can again apply Lemma 2, and obtain that $VM_4 \models \phi_7$ since ϕ_7 is β-dependent. This exemplifies that new temporal properties can be added to the refinement verification chain.

We note that in fact $VM_4 \models \phi_6$. Thus ϕ_6 and ϕ_7 together imply that $GF([dispenseBiscuit] \vee [dispenseChoc]))$ holds for VM_4.

6 Extending preserving LTL properties to handle splitting events

In this section we generalise the results of Section 5 in order to deal with splitting events in Event-B, which occurs when abstract events are refined by several events in the concrete machine, corresponding to a set of alternatives. Consider as a motivating example VM_0 in Figure 6. This is refined by VM_1, with linking invariant $item = card(chosen)$, *selectItem* refined by both *selectBiscuit* and *selectChoc*, and *dispenseItem* refined by both *dispenseBiscuit* and *dispenseChoc*. Splitting events also involves their renaming to allow for several concrete events to map to the same abstract one. A refinement step will therefore be associated with a renaming function h from concrete events to the abstract events that they refine. In the general case h will be many-to-one, since many concrete events may map to a single abstract event; and it will also be partial, since new events in the concrete machine will not map to any abstract event.

In general, each step in a refinement chain $M_0 \preccurlyeq M_1 \preccurlyeq \ldots \preccurlyeq M_n$ will have an event renaming function h_i corresponding to the renaming and splitting step from M_i to M_{i-1}. We define $g_{i,n}$ to be the composition of these renaming function from h_n down to h_i. Observe that $g_{i,n}$ will be undefined on any event that does not map to M_{i-1}, in other words any event that corresponds to an event introduced at some point in the refinement chain. For example, for the chain $VM_0 \preccurlyeq VM_1 \preccurlyeq \ldots \preccurlyeq VM_4$, we obtain that $g_{1,4}(selectBiscuit) = g_{1,4}(selectChoc) = selectItem$, and $g_{1,4}(dispenseBiscuit) = g_{1,4}(dispenseChoc) = dispenseItem$, and $g_{1,4}$ is not defined on the remaining events of VM_4.

Lemma 1 generalises to state that the final machine in the refinement chain must always eventually perform some event relating to an event in the initial machine.

Lemma 3. *If $M_0 \preccurlyeq M_1 \preccurlyeq \ldots \preccurlyeq M_n$ and M_n is deadlock free and M_n does not contain any anticipated events then $M_n \models GF(\bigvee_{e \in g_{1,n}^{-1}(\alpha M_0)} e)$*

Observe that if there is no renaming or splitting, then $g_{1,n}$ is the identity function on the events in αM_0, yielding Lemma 1.

We are interested in how the LTL properties of an abstract machine becomes transformed through a refinement step such as VM_0 to VM_1. For example, the property $GF[selectItem]$ for VM_0 states that from any stage that is reached, *selectItem* will eventually occur. This will translate to the property $GF([selectBiscuit] \vee [selectChoc])$ for VM_1. We now consider how LTL properties translate through a renaming function h.

For a given event renaming function h, we define $trans_h$ as the translation that maps LTL formulae by mapping abstract events to the disjunction of their corresponding concrete events, as follows:

machine VM_0
variables *item*
invariant $item \in \mathbb{N}$
events
init $\widehat{=}$ $item := 0$
selectItem $\widehat{=}$
status : ordinary
when $item \leq 2$ **then** $item := item + 1$ **end**
dispenseItem $\widehat{=}$
status : ordinary
when $item > 0$ **then** $item := item - 1$ **end**
end

Fig. 6: VM_0

Definition 5.

$$
\begin{aligned}
trans_h(true) &= true \\
trans_h([x]) &= \bigvee_{y|h(y)=x} [y] \\
trans_h(\neg\phi) &= \neg trans_h(\phi) \\
trans_h(\phi_1 \vee \phi_2) &= trans_h(\phi_1) \vee trans_h(\phi_2) \\
trans_h(\phi_1 \wedge \phi_2) &= trans_h(\phi_1) \wedge trans_h(\phi_2) \\
trans_h(\phi_1\ U\ \phi_2) &= trans_h(\phi_1)\ U\ trans_h(\phi_2) \\
trans_h(G\phi) &= G\ trans_h(\phi) \\
trans_h(F\phi) &= F\ trans_h(\phi)
\end{aligned}
$$

For example

$$
\begin{aligned}
&trans_h(G([selectItem] \Rightarrow F[dispenseItem])) \\
&= G(([selectBiscuit] \vee [selectChoc]) \Rightarrow F([dispenseBiscuit] \vee [dispenseChoc]))
\end{aligned}
$$

Lemma 2 generalises to Lemma 4 below, to state that LTL properties are carried along the refinement chain by translating them. In particular, if a property ϕ is established for M_{i-1}, then $trans_{g_{i,n}}(\phi)$ will hold for M_n:

Lemma 4. *If* $M_{i-1} \models \phi$ *and* $M_{i-1} \preccurlyeq \ldots \preccurlyeq M_n$ *and* $0 \leq i-1 < n$, M_n *is deadlock free and* M_n *does not contain any anticipated events and* ϕ *is* β*-dependent and* $\beta \subseteq \alpha M_{i-1}$ *then* $M_n \models trans_{g_{i,n}}(\phi)$

For example, from the result for VM_0 that whenever *selectItem* occurs then *dispenseItem* will eventually occur,

$$VM_0 \models G([selectItem] \Rightarrow F[dispenseItem]))$$

we obtain from Lemma 4 that

$$VM_4 \models G(\ ([selectBiscuit] \vee [selectChoc]) \\ \Rightarrow F([dispenseBiscuit] \vee [dispenseChoc])\ )$$

This states that whenever *selectBiscuit* or *selectChoc* occur, then *dispenseBiscuit* or *dispenseChoc* will eventually occur.

7 Discussion and related work

One of the few papers to discuss LTL preservation in Event-B refinement is Groslambert [8]. The LTL properties were defined in terms of predicates on system state rather than our paper's formulation in terms of the occurrence of events. His paper focused only on the introduction of new convergent events. It did not include a treatment of anticipated events but this is unsurprising since the paper was published before their inclusion in Event-B. Our results are more general in two ways. Firstly, the results support the treatment of anticipated events. Secondly, we allow more flexibility in the development methodology. A condition of Groslambert's results was that all the machines in the refinement chain needed to be deadlock free. The two main lemmas in our paper: Lemmas 2 and 4 do not require each machine in a refinement chain to be deadlock free, only the final machine. It is irrelevant if intermediate M_is deadlock as long as the deadlock is eventually refined away.

Groslambert deals with new events via stuttering and leaves them as visible events in a trace. This is why the LTL operators used by the author do not include the next operator (X). As new events may happen this may violate the X property to be checked. Plagge and Leuschel in [14] permit the use of the X operator since they treat the inclusion of new events as internal events which are not visible. Since we deal with new events as visible events we also lose the ability to reason about a temporal property using the typical X operator. Our reasoning is simpler than both Groslambert and Plagge and Leuschel since we only focus on events but this means we cannot have atomic propositions in our LTL, whereas they can.

The notion of verification of temporal properties of both classical and Event-B systems using proof obligations has been considered in many research papers. Abrial and Musat in an early paper, [3], introduced proof obligations to deal with dynamic constraints in classical B. In a more recent paper [10] Hoang and Abrial have also proposed new proof obligations for dealing with liveness properties in Event-B. They focus on three classes of properties: existence, progress and persistence, with a view to implementing them in Rodin. Bicarregui *et al.* in [5] introduced a temporal concept into events using the guard in the *when* clause and the additional labels of *within* and *next* so that the enabling conditions are captured clearly and separately. However, these concepts are not aligned with the standard Event-B labelling.

The interest of LTL preservation through refinement is wider than simply Event-B. Derrick and Smith [7] discuss the preservation of LTL properties in the

context of Z refinement but the authors extend their results to other logics such as CTL and the μ calculus. They focus on discussing the restrictions that are needed on temporal-logic properties and retrieve relations to enable the model checking of such properties. Their refinements are restricted to data refinement and do not permit the introduction of new events in the refinement steps. Our paper does permit new events to be introduced during refinement steps; the contribution is in identifying conditions for LTL properties to hold even in the context of such new events.

8 Conclusions and future work

The paper has provided foundational results that justify when temporal properties hold at the end of an Event-B refinement chain for developments which contain anticipated, convergent and ordinary events, which goes beyond that presented in [8]. The paper has also provided restrictions on the temporal properties in terms of being β-dependent which help to determine when a temporal property of interest should be introduced into the development chain.

We could extend the results to deal with merging events. The inclusion of the X LTL operator and availability will require use to look at execution paths which include state transitions (π paths). The inclusion of availability will enable us to address more advanced and useful notions of fairness in the context of temporal properties. Our notion of weak fairness will be akin to that described in Barradas and Bert in [4]. It will draw on work by Williams *et al.* [17]. We could also consider the impact on temporal property preservation in refinement chains which do not achieve convergence of all its new events by the end.

In ongoing work we are looking at event liveness via the proof obligation for strong deadlock freedom `S_NDF`. We have defined new labelling of events to so that liveness proofs are on particular events. This is analagous to proving `WFD_REF` for events that are labelled anticipated or convergent. We have recently defined the semantics of Event-B in terms of stable failures and detailed its relationship with `S_NDF`. We are currently combining these results with our work in [15] in order to provide a cohesive process algebra underpinning for Event-B.

Acknowledgments. Thanks to Thai Son Hoang and Thierry Lecomte for discussions about Event-B development strategies and the challenges of discharing liveness proofs. Thanks to Steve Wesemeyer for discussions on the example. Thanks to the reviewers for their constructive comments that helped to improve the paper.

References

1. J.-R. Abrial. *Modeling in Event-B: System and Software Engineering.* Cambridge University Press, 2010.
2. J.-R. Abrial, M. J. Butler, S. Hallerstede, T. S. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

3. J.-R. Abrial and L. Mussat. Introducing dynamic constraints in B. In *B*, volume 1393 of *LNCS*, pages 83–128. Springer, 1998.
4. H. Barradas and D. Bert. Specification and proof of liveness properties under fairness assumptions in B event systems. In *Integrated Formal Methods*, volume 2335 of *LNCS*, pages 360–379. Springer, 2002.
5. J. Bicarregui, A. Arenas, B. Aziz, P. Massonet, and C. Ponsard. Towards modelling obligations in Event-B. In *Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 181–194. Springer, 2008.
6. M. J. Butler. *A CSP approach to Action Systems*. DPhil thesis, Oxford U., 1992.
7. J. Derrick and G. Smith. Temporal-logic property preservation under Z refinement. *Formal Asp. Comput.*, 24(3):393–416, 2012.
8. J. Groslambert. Verification of LTL on B Event Systems. In *B 2007: Formal Specification and Development in B*, volume 4355 of *LNCS*, pages 109–124. Springer, 2006.
9. S. Hallerstede, M. Leuschel, and D. Plagge. Validation of formal models by refinement animation. *Science of Computer Programming*, 78(3):272 – 292, 2013.
10. T. S. Hoang and J.-R. Abrial. Reasoning about liveness properties in Event-B. In *ICFEM*, volume 6991 of *LNCS*, pages 456–471. Springer, 2011.
11. M. Leuschel and M. J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.
12. M. Leuschel, J. Falampin, F. Fritz, and D. Plagge. Automated property verification for large scale B models. In *FM*, volume 5850 of *LNCS*, pages 708–723. Springer, 2009.
13. C. Morgan. Of wp and CSP. *Beauty is our business: a birthday salute to E. W. Dijkstra*, pages 319–326, 1990.
14. D. Plagge and M. Leuschel. Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *STTT*, 12(1):9–21, 2010.
15. S. Schneider, H. Treharne, and H. Wehrheim. The behavioural semantics of Event-B refinement. *Formal Asp. Comput.*, 26(2):251–280, 2014.
16. S. Schneider, H. Treharne, H. Wehrheim, and D. Williams. Managing LTL properties in Event-B refinement. arXiv:1406:6622, June 2014.
17. D. M. Williams, J. de Ruiter, and W. Fokkink. Model checking under fairness in ProB and its application to fair exchange protocols. In *ICTAC*, volume 7521 of *LNCS*, pages 168–182. Springer, 2012.