Under consideration for publication in Formal Aspects of Computing

The original publication is available at http://dx.doi.org/10.1007/s00165-016-0376-0
In Formal Aspects of Computing, online since April 2016

Foundations for using Linear Temporal Logic in Event-B refinement

Thai Son Hoang[1] and Steve Schneider[2] and Helen Treharne[2] and David M. Williams[3]

[1]ECS, University of Southampton, UK,
[2]Department of Computer Science, University of Surrey, UK,
[3]VU University, Amsterdam

Abstract. In this paper we present a new way of reconciling Event-B refinement with linear temporal logic (LTL) properties. In particular, the results presented in this paper allow properties to be established for abstract system models, and identify conditions to ensure that the properties (suitably translated) continue to hold as those models are developed through refinement. There are several novel elements to this achievement: (1) we identify conditions that allow LTL properties to be mapped across refinement chains; (2) we provide translations of LTL predicates to reflect the introduction through refinement of new events and the renaming and splitting of existing events; (3) we do this for an extended version of LTL particularly suited to Event-B, including state predicates and enabledness of events, which can be model-checked at the abstract level. Our results are more general than any previous work in this area, covering liveness in the context of anticipated events, and relaxing constraints between adjacent refinement levels. The approach is illustrated with a case study. This enables designers to develop event based models and to consider their execution patterns so that liveness and fairness properties can be verified for Event-B systems.

Keywords: Event-B, Refinement, Linear Temporal Logic

1. Introduction

Event-B [Abr10] is a step-wise development method with excellent tools: Atelier B [Cle14] and the Rodin platform [ABH+10, Z+14] providing proof support and ProB [LB08] providing model checking. As Hoang and Abrial [HA11] 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 [LFFP09]. 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 the case in general.

Event-B describes systems in terms of *machines* with state, and *events* which are used to update the

Correspondence and offprint requests to: Steve Schneider/Helen Treharne, Department of Computing, University, of Surrey, Guildford, GU2 7XH, UK. e-mail: {S.Schneider, H.Treharne}@surrey.ac.uk

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 [Abr10]. 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, to show that concrete progress corresponds (eventually) to progress at the abstract level. 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 ensure that events cannot occur forever: convergent events must decrease a variant and anticipated events cannot increase it. In this paper, following Abrial [Abr10] 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 over entire system executions. 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. Both of these possibilities are permitted within the Event-B refinement framework. 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 fairness property on *closeDoors* to prevent termination with the doors open.

In this paper we present results to enable temporal logic properties to be carried through Event-B refinement chains. The results generalise to events that are split—refined by several events—during a refinement chain. The paper builds on [STWW14b] but it is entirely self-contained and provides significant theoretical extension to previous work. Importantly, the results presented in this paper support discharging a wider range of LTL properties through Event-B refinement than the previous work, since we can now include all the LTL operators supported by the ProB tool, i.e., support for the *next*, *proposition* and *enabled* operators is new for this paper. Note that LTL properties can be introduced at different levels of refinement. This paper's focus is much more in clarifying what refinement proofs need to be conducted during a refinement depending on the LTL properties of interest. The main new contribution is the result that allows properties that include the *enabled* ($e(t)$) operator to be preserved through a refinement chain and this requires a new proof obligation to be discharged during development.

The benefit of the new extensions is that they have facilitated the removal of an awkward restriction (β-dependency), which identified conditions on temporal logic properties to make them suitable for use in a refinement chain. Hence the LTL properties that can be specified are much more natural than in [STWW14b]. The rigour of the results is 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 [STW14].

The paper is organised as follows: Section 2 provides the necessary Event-B refinement background and one of the refinement strategies we use in the paper and was previously presented in [STWW14b]. Section 3 introduces a running example and is modified from the example in [STWW14b] to include more detail and provide the basis for new LTL properties. Section 4 introduces the infinite traces Event-B semantics required as a basis for the technical results. It is an extension of our previous work since a new definition of the refinement relation is required to support the proofs. Section 5 defines the LTL we use and it is an extended version of LTL from that presented in [STWW14b]. In particular the extended LTL contains the *next* operator, state predicates and supports the *enabled* operator. Sections 6 and 7 present and illustrate the main theoretical results. The proofs of these results are included in the Appendix. The sections have been

split into two so that it is clear what refinements proof obligations would need to be carried out in practice for a particular system in order to discharge appropriate LTL properties. The proof obligations required to be discharged for a particular system are dependent on the operators used within the LTL properties. Section 6 is significantly different from our referenced conference paper and Section 7 is entirely new. We place our work into the context of related work in Section 8 and in particular [HH13]. The paper concludes with a discussion and directions for future work in Section 9.

2. Event-B

2.1. Event-B Machines

An Event-B development is defined using *machine*s. A machine M contains a set of variables v and a set of events. The *alphabet* of M, αM, is the set of events defined in M. Each event evt has the general form $evt \mathrel{\widehat{=}}$ **any** x **where** $G_{evt}(x, v)$ **then** $v :| BA_{evt}(v, x, v')$ **end**, where x represents the parameters of the event, the guard $G_{evt}(x, v)$ is the condition for the event to be enabled. The body is given by $v :| BA_{evt}(v, x, v')$ whose execution assigns to v any value v' which makes the *before-after* predicate $BA_{evt}(v, x, v')$ true. This simplifies to $evt \mathrel{\widehat{=}}$ **when** $G_{evt}(v)$ **then** $v :| BA_{evt}(v, v')$ **end** when there are no parameters, since the guard and the *before-after* predicate does not refer to the parameters x. In the paper we also use the shorthand G_{evt} to denote the guard of the event evt when we omit the parameters for brevity. In practice, events are written using the Generalised Substitution Language, a syntactic sugar for events which encapsulate before-after relationsips in a programming style by means of abstract assignments of values to variables. For example, $v := v + x$ corresponds to the relation $v' = v + x$.

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 at least one event in M is enabled at anytime. When this obligation holds M is *deadlock free*.

2.2. Event-B Refinement

An Event-B development is a sequence of 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 machine M is considered to be refined by M' if the given *linking invariant* J on the variables between the two machines, $J(v, v')$, 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 for every event: feasibility `FIS_REF`, guard strengthening `GRD_REF` and invariant preservation `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 ensures 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 [Abr10].

Refinement is transitive, therefore for a chain of refinements $M_0 \preccurlyeq M_1 \preccurlyeq \ldots \preccurlyeq M_n$ there is a linking invariant between each refinement step such that there is a combination of linking invariants for all refinement steps:. We make explicit that there is a relationship between the states of the initial and final machine through a chain of refinements: $\mathcal{J}(v_0, v_n) = \exists\, v_1, v_2, \ldots v_{n-1}.J_1(v_0, v_1) \wedge J_2(v_1, v_2) \ldots \wedge J_n(v_{n-1}, v_n)$, and similarly between two machines M_i and M_j where $j > i$ in a refinement chain as follows: $\mathcal{J}(v_i, v_j) = \exists\, v_{i+1}, \ldots v_{j-1}.J_{i+1}(v_i, v_{i+1}) \ldots \wedge J_j(v_{j-1}, v_j)$, where M_i and M_j are adjacent $\mathcal{J}(v_i, v_j) = J(v_i, v_j)$.

A refinement machine can introduce new events and split existing events. We omit the treatment of merging events in this paper. New events evt' are treated as refinements of *skip*, i.e., evt' does not refine an event in M. Note that when splitting events, M' has several events $evt'_1 \ldots evt'_j$ refining a single event evt.

We define a function f which maps each concrete event to the abstract event that it refines. Therefore, the type of f is $f \in \alpha M' \twoheadrightarrow \alpha M$, a partial surjection from the alphabet of the refinement machine M'

selectItem (o)	*selectBiscuit* (o)	*selectBiscuit* (o)	*selectBiscuit* (o)	*selectBiscuit* (o)
	selectChoc (o)	*selectChoc* (o)	*selectChoc* (o)	*selectChoc* (o)
dispenseItem (o)	*dispenseBiscuit* (o)	*dispenseBiscuit* (o)	*dispenseBiscuit* (o)	*dispenseBiscuit* (o)
	dispenseChoc (o)	*dispenseChoc* (o)	*dispenseChoc* (o)	*dispenseChoc* (o)
		pay (a)	*pay* (a)	*pay* (c)
		refund (c)	*refund* (o)	*refund* (o)
			refill (c)	*refill* (o)
VM_0	VM_1	VM_2	VM_3	VM_4

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

$$f1 = \{sb \mapsto si, sc \mapsto si, db \mapsto di, dc \mapsto di\}$$
$$f2 = \{sb \mapsto sb, sc \mapsto sc, db \mapsto db, dc \mapsto dc\} = id$$
$$f3 = \{sb \mapsto sb, sc \mapsto sc, db \mapsto db, dc \mapsto dc, pay \mapsto pay, refund \mapsto refund\}$$
$$f4 = \{sb \mapsto sb, sc \mapsto sc, db \mapsto db, dc \mapsto dc, pay \mapsto pay, refund \mapsto refund, refill \mapsto refill\}$$
$$g_{1,4} = f_4 ; f_3 ; f_2 ; f_1 = \{sb \mapsto si, sc \mapsto si, db \mapsto di, dc \mapsto di\}$$

Fig. 2. Function mappings for the Vending Machine development

onto the alphabet of the abstract machine M. New concrete events will not map to any abstract event and are therefore not in $dom(f)$; this is why f is a partial function. In the general case f will be many-to-one, since many concrete events may map to a single abstract event. Thus, $f(evt'_1) = evt \Leftrightarrow evt'_1$ **refines** evt and similarly for any other refinement events which relate to evt. In this paper we include the **refines** annotation only when we are splitting events in a refinement, otherwise we consider all similarly named events in a refinement step to be straightforward refinements of their counterparts in the previous step.

We also provide a clear definition, Definition 2.1, to relate the mapping of concrete events to abstract events through a refinement chain. The function $g_{i,j}$ defines a mapping for concrete events to abstract events, made up of composing the function mappings f at each refinement level, i.e., f_j down to f_i, where $i \leq j$. We explicitly annotate f with the appropriate refinement level, i.e., $f_{i+1} : \alpha M_{i+1} \twoheadrightarrow \alpha M_i$ (Note that $g_{1,1} = f_1$). $g_{i,j}$ is also a partial surjective function. Note that the annotated index is the one from the concrete machine that f is mapping from. Thus for an arbitrary chain we define the compositional mapping as follows:

Definition 2.1. $g_{i,j} = f_j ; f_{j-1} ; \ldots ; f_i$

Note that $f_j ; f_{j-1} ; \ldots ; f_i = f_i \circ \ldots \circ f_{j-1} \circ f_j$. Also $g_{1,n}$ represents the compositional mapping for the entire refinement chain M_0 to M_n. Observe that $g_{i,n}$ will be undefined on any event that does not map to the first machine M_{i-1} in the sequence. In other words any event that was introduced at some point in the refinement chain does not map back to M_{i-1}.

Figure 1 illustrates a vending machine development, detailed in Section 3 that we use throughout the paper and Figure 2 identifies the functional mappings between each refinement layer within that development (using abbreviations for the *select* and *dispense* events, for example *selectBiscuit* is *sb*).

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. Alternatively a variant can be a finite set that decreases in size with convergent events, and does not increase with anticipated events. 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. Note that in Section 7 we will introduce a further stronger notion of refinement that takes liveness into account.

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

Fig. 3. VM_0

2.3. Event-B Development Strategy

Event-B has a strong but flexible refinement strategy which is described in [HLP13]. In [STW14] we also discussed different Event-B refinement strategies and characterised them with respect to the approaches documented by Abrial in [Abr10] and supported by the Rodin tool. In this paper we focus on the simplest strategy, and the one most commonly used. It is also a strategy supported by the Atelier B tool [Cle14]. 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} (i.e., events can be split);
3. each new event in M_i is either anticipated or convergent, where $i > 0$;
4. each event in M_{i+1} refines exactly one event of M_i (i.e., events cannot be merged);
5. each event in M_{i+1} which refines an anticipated event of M_i is itself either convergent or anticipated;
6. refinements of convergent or ordinary events of M_i are ordinary in M_{i+1}.
7. 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, $O_0 = \{selectItem, dispenseItem\}$ and $C_0 = \emptyset$, $O_1 = \{selectBiscuit, selectChoc, dispenseBiscuit, dispenseChoc\}$ and $C_1 = \emptyset$ in VM_1. In VM_2 we note that $C_2 = \{refund\}$. In VM_3 we note that $C_3 = \{refill\}$ and in VM_4 we have $C_4 = \{pay\}$.

3. Example

In Section 2.3 we introduced a development strategy for a vending machine. Figures 3, 4, 5, 6 and 7 illustrate a development chain from machines VM_0, VM_1, VM_2, VM_3 to VM_4; there are no anticipated events in VM_4.

VM_0 is a simple machine that supports the selection and dispensing of items via two very simple events: *selectItem* and *dispenseItem*. The *item* variable is used to control when the events are enabled. VM_1 decomposes the selection and dispensing to be specifically for chocolates and biscuits via four events: *selectBiscuit*, *selectChoc*, *dispenseBiscuit* and *dispenseChoc*. VM_1 contains a data refinement which tracks the item chosen rather simply counting the selection. The second 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 *selectBiscuit*, *selectChoc*, *dispenseBiscuit* and *dispenseChoc* 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

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

Fig. 4. VM_1

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

Fig. 5. VM_2

machine VM_3
variables $credit, chosen, refundEnabled, stocked$
invariant
$credit \in \mathbb{N} \land chosen \subseteq \{choc, biscuit\} \land card(chosen) \leq credit \land refundEnabled \in BOOL$
$\land\ stocked \subseteq \{choc, biscuit\} \land (choc \in chosen \Rightarrow choc \in stocked)$
$\land\ (biscuit \in chosen \Rightarrow biscuit \in stocked)$
variant $2 - card(stocked)$
events
init $\widehat{=} \ldots \parallel stocked := \{choc, biscuit\}$
pay $\widehat{=}$ **status** : anticipated
any x **where** $x \in \mathbb{N}_1 \land stocked \neq \emptyset$
then $credit := credit + x$ **end** $\parallel$ $refundEnabled := FALSE$ **end**
selectBiscuit $\widehat{=}$ **status** : ordinary
when $\ldots \land biscuit \in stocked$
then $chosen := chosen \cup \{biscuit\}$ **end**
selectChoc $\widehat{=}$ **status** : ordinary
when $\ldots \land choc \in stocked$
then $chosen := chosen \cup \{choc\}$ **end**
dispenseBiscuit $\widehat{=}$ **status** : ordinary
when $biscuit \in chosen \land biscuit \in stocked$
then $\ldots \parallel$ **any** x **where** $x \subseteq \{biscuit\}$ **then** $stocked := stocked - x$ **end end**
dispenseChoc $\widehat{=}$ **status** : ordinary
when $choc \in chosen \land choc \in stocked$
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 $choc \notin stocked \lor biscuit \notin stocked$
then $stocked := \{choc, biscuit\}$ **end**
end

Fig. 6. VM_3

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 events introduced in VM_1 have been strengthened so that they are only enabled when the appropriate stocked item is in stock. But now *dispenseBiscuit* and *dispenseChoc* also capture the non-deterministic 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.

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.

The example has been proved using Rodin 3.1.0 with Atelier-B 2.1.0[1]. The development uses finite set variants instead of encodings in natural numbers to facilitate proof, for example the variant in VM_3 would be $ITEM - stocked$. The proof statistics are as follows: context: 1, VM_0: 6, VM_1: 18, VM_2: 19, VM_3: 9 and VM_4: 46 proof obligations (44 automatic, 2 manual proofs). In total, there are 99 proof obligations (97 automatic).

[1] `http://www.computing.surrey.ac.uk/personal/st/H.Treharne/papers/2015/VendingMachineDevelopment_FACS2015.zip`

machine VM_4
constants *capacity*
properties $capacity > 0$
variables *credit*, *chosen*, *refundEnabled*, *chocStock*, *biscuitStock*
invariant $credit \in \mathbb{N} \wedge chosen \subseteq \{choc, biscuit\} \wedge card(chosen) \leq credit \wedge$
$refundEnabled \in BOOL \wedge chocStock \leq capacity \wedge biscuitStock \leq capacity \wedge$
$(choc \notin stocked \Rightarrow chocStock = 0) \wedge (choc \in stocked \Rightarrow chocStock > 0) \wedge$
$(biscuit \notin stocked \Rightarrow biscuitStock = 0) \wedge (biscuit \in stocked \Rightarrow biscuitStock > 0)$
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**
selectBiscuit $\widehat{=}$ **status** : ordinary
when $\ldots \wedge biscuitStock > 0$
then $chosen := chosen \cup \{biscuit\}$ **end**
selectChoc $\widehat{=}$ **status** : ordinary
when $\ldots \wedge chocStock > 0$
then $chosen := chosen \cup \{choc\}$ **end**
dispenseBiscuit $\widehat{=}$ **status** : ordinary
when $biscuit \in chosen \wedge biscuitStock > 0$
then $\ldots \parallel biscuitStock := biscuitStock - 1$ **end**
dispenseChoc $\widehat{=}$ **status** : ordinary
when $choc \in chosen \wedge chocStock > 0$
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. 7. VM_4

4. Event-B Semantics

For a particular machine M we define the ternary relation T as follows: $(s, t, s') \in T \Leftrightarrow \exists x.G_t(x, s) \wedge BA_t(s, x, s')$. This gives the relationship between the before and after state for the events within that machine where s ans s' are states and t is an event.

In this paper we define a trace (path) u of M as a finite or infinite sequence of alternating states and events (a,c or o), of the form, $\langle s_0, t_0, s_1, t_1, \ldots \rangle$ where $\forall i \geq 0.(s_i, t_i, s_{i+1}) \in T$. A path $\langle s_0, t_0, \ldots, s_{k-1}, t_{k-1}, s_k \rangle$ is finite when s_k is a deadlock state, i.e., $\neg \bigvee_{e \in \alpha M} G_e$: none of the events of the machine are enabled. When a machine M is deadlock free all of its traces are infinite.

Our definition of path is akin to Definition 2 in [PL10] where paths are also defined in terms of a ternary relation between states. Note that in [STWW14a] we did not need to consider the intermediate states as all our reasoning was based on the events of infinite traces. In this paper we need to consider intermediate states since we will be reasoning about LTL properties that require us to refer to the intermediate states.

We use the functions of projection ($\upharpoonright$), length (#) and concatenation ($\frown$) on finite and infinite sequences. Projection $u \upharpoonright E$ of a path u onto a set of events E gives the subsequence of transitions of u that are in E. This is used for example in Definition 4.1 below. Note that $u \upharpoonright \alpha M = \langle t_0, t_1, \ldots \rangle$ represents the trace of all events in the path u. The length of u, $\#(u) = \#(u \upharpoonright \alpha M)$, is defined as the number of transitions in

a path[2]. A finite sequence of state transition pairs ending with an event can be concatenated with a path, i.e., $\langle s_0, t_0, \ldots t_k \rangle \frown u$. In this paper concatenation is simply used to partition a finite or infinite sequence of events.

We additionally need to define the relationship between abstract and concrete paths since we are interested in their relationship through refinement. Thus, in the following definition we relate abstract and concrete paths. Relation R is defined as follows:

Definition 4.1. The relation R is the weakest relation such that: $\langle s_0, t_0, s_1, t_1, \ldots \rangle$ is a path of M_i and $\langle s'_0, t'_0, s'_1, t'_1, \ldots \rangle$ is a path of M_j and $i < j$ and $\langle s_0, t_0, s_1, t_1, \ldots \rangle R \langle s'_0, t'_0, s'_1, t'_1, \ldots \rangle$ iff

- $\langle s'_0, t'_0, s'_1, t'_1, \ldots \rangle \upharpoonright dom(g_{i+1,j})$ is infinite
- if $t'_0 \in dom(g_{i+1,j})$ then $t_0 = g_{i+1,j}(t'_0) \wedge \mathcal{J}(s_0, s'_0) \wedge \mathcal{J}(s_1, s'_1) \wedge \langle s_1, t_1, \ldots \rangle R \langle s'_1, t'_1, \ldots \rangle$
- if $t'_0 \notin dom(g_{i+1,j})$ then $J(s_0, s'_0) \wedge \mathcal{J}(s_0, s'_1) \wedge \langle s_0, t_0, s_1, t_1, \ldots \rangle R \langle s'_1, t'_1, \ldots \rangle$

Note that the states are matched through $\mathcal{J}$ and R must be the weakest relation in order to rule out false as a possible R. The relation is general to be between two paths and two arbitrary different refinement levels. The relation is not restricted to relating two adjacent refinement steps. This definition bears some resemblance to Definition 11 of [DJK03], though that paper is not concerned with the translation of LTL properties.

Note that the relation between two paths u_1 and u_2 (i.e., u_1 R u_2) only holds if u_2 projected onto the alphabet of the events u_1 lives in is infinite. This means that all the events in the infinite trace u_1 will correspond to something in u_2. Consider an infinite trace u_1 related to an infinite trace u_2 which only had finitely many events in u_1 that it matched. For example, if $u_1 = \langle a, a, a, a, \ldots \rangle$ and $u_2 = \langle b, c, b, c, b, c, c, c, c, c, \ldots \rangle$ where b maps to a in the mapping function g, and c is a new event. Then R does not hold between these two paths, since the infinite condition on u_2 is not met. Our theoretical results require R to be defined with respect to infinite traces since we will only need to consider infinite traces. We never need to consider finite abstract traces in our proofs, hence this definition is only required to define R on infinite traces.

For example, a path of VM_0 is

$$u_0 = \langle 0, selectItem, 1, selectItem, 2, dispenseItem, 1, selectItem, 2, \ldots \rangle$$

where the state i carries the value of $item$.

Similarly, a possible path of VM_1 is

$$u_1 = \langle \emptyset, selectBiscuit, \{biscuit\}, selectChoc, \{biscuit, choc\}, \\ dispenseChoc, \{biscuit\}, selectChoc, \{biscuit, choc\}, \ldots \rangle$$

where the state s carries the value of $chosen$.

Then we have that $u_0 R u_1$. We have that abstract states and concrete states are related by the linking invariant of VM_1, for example the fourth abstract state $item = 1$ relates to the fourth concrete state $chosen = \{biscuit\}$. We also have that the abstract and concrete events are mapped through $g_{1,1}$, for example the fourth concrete event is $selectChoc$ and the fourth abstract event is $selectItem$. Since all the states are related through the linking invariant, and all the concrete events map to the abstract events, we have that $u_0 R u_1$.

Now consider the following path of VM_2:

$$\begin{array}{lll} u_2 = \langle (0, \emptyset, FALSE), & & \\ \quad pay, & (3, \emptyset, FALSE), & \\ \quad selectBiscuit, & (3, \{biscuit\}, FALSE) & \\ \quad selectChoc, & (3, \{biscuit, choc\}, FALSE), & \\ \quad dispenseChoc, & (2, \{biscuit\}, TRUE) & \\ \quad refund, & (1, \{biscuit\}, FALSE), & \\ \quad pay, & (5, \{biscuit\}, FALSE), & \\ \quad selectChoc, & (5, \{biscuit, choc\}, FALSE), & \\ & \ldots & \rangle \end{array}$$

[2] Note that length of a path in [PL10] is defined as the number of states in a path, however for this paper our focus is on events.

Then $u_1 R u_2$: transitions for the new events *pay* and *refund* meet the third condition of Definition 4.1, and those for the events already present, e.g., *selectBiscuit*, meet the second condition. Hence it also follows that $u_0 R u_2$ through two refinement steps.

Conversely, consider the path u_2' of VM_2 consisting of an infinite sequence of *pay* transitions:

$$u_2' = \langle (0, \emptyset, FALSE), pay, (1, \emptyset, FALSE), pay, (2, \emptyset, FALSE), \ldots \rangle$$

Such a path fails the first condition of Definition 4.1, since there are not infinitely many transitions corresponding to abstract events. Hence u_2' does not relate under R to any path of VM_1 or VM_0.

Thus R only related infinite abstract paths to infinite concrete paths.

A more complex behavioural semantics for B machines was given by Schneider *et al.* in [STW14] based on the weakest precondition semantics of [Mor90, But92] for action systems and CSP. In [STW14] 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 4.2. $CA(C, O)(u) \mathrel{\widehat{=}} (\#(u \upharpoonright C) = \infty \Rightarrow \#(u \upharpoonright O) = \infty)$

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 [STW14], restated as Theorem 4.3, allows us to conclude that there are no infinite sequences of convergent events in the final machine of a refinement chain M_n.

Theorem 4.3. If $M_0 \preccurlyeq_W M_1 \preccurlyeq_W \ldots \preccurlyeq_W M_n$ then

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

M **sat** S means that predicate S holds for every infinite trace of machine M. The result for our example is

$$VM_4 \textbf{ sat } CA(\{pay, refund, refill\} \ , \ \{selectBiscuit, selectChoc, dispenseBiscuit, dispenseChoc\})$$

Thus $CA(\{pay, refund, refill\} \ , \ \{selectBiscuit, selectChoc, dispenseBiscuit, dispenseChoc\})$ holds for all traces of VM_4. If there are infinitely many *pay*, *refund* and *refill* events then there must be inifinitely many select and dispense events. In other words an execution cannot reach a point where it only does *pay*, *refund* and *refill* events from then on. It must always progress to a select or dispense event.

Note that in particular the *CA* property does not hold for the infinite trace $\langle pay, pay, pay, \ldots \rangle$ therefore that trace cannot be a trace of VM_4. Indeed, we can see that VM_4 only allows *pay* while the amount of credit is less than the amount of stock held by the machine.

5. LTL notation

In this paper we use the following grammar for the LTL operators:

$$\phi ::= true \mid [t] \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1 \ U \ \phi_2 \mid p \mid X\phi \mid e(t)$$

These operators differ from those presented in [STWW14a] since we have included the state predicate p, the next operator X and the enabledness of an event t, $e(t)$. Note that we do not provide theoretical results for $e(t)$ until Section 7 since the theoretical results require a more detailed treatment of Event-B refinement than those for the other operators. It is the same grammar as presented by Plagge and Leuschel in [PL10] and hence supported by the ProB tool.

A machine M satifies ϕ, denoted $M \models \phi$, if all paths of M starting from one its initial states satisfy ϕ. The definition for u to satisfy ϕ is defined by induction over ϕ as follows:

Definition 5.1.

$$
\begin{array}{lcl}
u \models \mathit{true} & & \\
u \models [t] & \Leftrightarrow & \#(u) \geq 1 \text{ and } u = \langle s_0, t \rangle \frown u^1 \\
u \models \neg\phi & \Leftrightarrow & \text{it is not the case that } u \models \phi \\
u \models \phi_1 \vee \phi_2 & \Leftrightarrow & u \models \phi_1 \text{ or } u \models \phi_2 \\
u \models \phi_1 \, U \phi_2 & \Leftrightarrow & \exists\, k \geq 0. \forall\, i < k. u^i \models \phi_1 \text{ and } u^k \models \phi_2 \\
u \models p & \Leftrightarrow & u = \langle s_0, \ldots \rangle \text{ and } p \text{ is true in } s_0 \\
u \models e(t) & \Leftrightarrow & u = \langle s_0, \ldots \rangle \text{ and } G_t \text{ is true in } s_0 \\
u \models X\phi & \Leftrightarrow & \#(u) \geq 1 \text{ and } u^1 \models \phi
\end{array}
$$

where u^n is u with the first n state-transition pairs removed, i.e., $u = \langle s_0, t_0, \ldots, s_{n-1}, t_{n-1} \rangle \frown u^n$. The until operator is a strong until since there is a finite k for which $u^k \models \phi_2$. Note that "G_t is true in s_0" is written as $G_t(s_0)$ in the proofs in this paper.

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. From their definition of $F\phi = \mathit{true} U \phi$ and U from above it follows that $F([a]\,U\,[b]) = F([b])$.

We also use these additional operators, and for explicitness we also provide direct definitions for them:

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

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 $G([\mathit{top}] \Rightarrow F[\mathit{ground}])$. Similarly, for our running example given in Section 3, we can state that whenever *selectItem* happens then eventually *dispenseItem* will happen is captured using the following LTL property. We reference it for use later in the paper.

$$\phi_A = G([\mathit{selectItem}] \Rightarrow F[\mathit{dispenseItem}]) \tag{1}$$

From our running *VM* example, in [STWW14b] we predominantly discussed LTL properties of the form the predicate $GF[\mathit{selectBiscuit}]$ which expresses that *selectBiscuit* occurs infinitely often: at any point there is always some occurrence of *selectBiscuit* at some point in the future. We used this construction to express properties such as:

$$\phi_B = (\neg GF[\mathit{selectBiscuit}]) \Rightarrow G([\mathit{selectChoc}] \Rightarrow F[\mathit{dispenseChoc}]) \tag{2}$$

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

In [STWW14b] we also discussed how LTL properties of an abstract machine become transformed through a refinement step. For example, one LTL property of VM_0 is $GF[\mathit{selectItem}]$ which states that from any state that is reached, *selectItem* will eventually occur. This translates to the property $GF([\mathit{selectBiscuit}] \vee [\mathit{selectChoc}])$ for VM_1 and the property remains unchanged through the remaining refinement steps.

Appropriate properties for VM_0 in this paper are:

$$\phi_C = G(\mathit{item} = 2 \Rightarrow (X(\mathit{item} = 1))) \tag{3}$$

$$\phi_D = G(\mathit{item} = 0 \Rightarrow (X(\mathit{item} = 1))) \tag{4}$$

Notice that ϕ_C and ϕ_D use the next and proposition operator which are new to this paper's results. In Section 6 we will discuss how these are transformed and preserved through refinement. Property ϕ_C states that when you select the maximum amount of items then the amount of selected items will be decreased in the next state. This is true since the dispense item is the only event that can happen when $\mathit{item} = 2$ and this is true in every state where the maximum number of items have been chosen. Property ϕ_D is similar in style to ϕ_C and states that whenever *item* is 0 then the only thing that can happen is that an item will be selected and hence in the next state the value of *item* will be 1.

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 5.2.

$$\begin{aligned}
\alpha(\mathit{true}) &= \emptyset \\
\alpha([t]) &= \{t\} \\
\alpha(\neg\phi) &= \alpha(\phi) \\
\alpha(\phi_1 \vee \phi_2) &= \alpha(\phi_1) \cup \alpha(\phi_2) \\
\alpha(\phi_1 \wedge \phi_2) &= \alpha(\phi_1) \cup \alpha(\phi_2) \\
\alpha(\phi_1\ U\ \phi_2) &= \alpha(\phi_1)\ \cup\ \alpha(\phi_2) \\
\alpha(F\phi) &= \alpha(\phi) \\
\alpha(G\phi) &= \alpha(\phi) \\
\alpha(p) &= \emptyset \\
\alpha(e(t)) &= \{t\} \\
\alpha(X\phi) &= \alpha(\phi)
\end{aligned}$$

For example, we have $\alpha(\phi_A) = \{\mathit{selecItem}, \mathit{dispenseItem}\}$ and $\alpha(\phi_C) = \alpha(\phi_D) = \emptyset$ for ϕ_A, ϕ_C and ϕ_D above.

6. Preserving LTL properties

In this section we provide results to demonstrate when properties are preserved by refinement chains of the form: $M_0 \preccurlyeq_W M_1 \preccurlyeq_W \ldots \preccurlyeq_W M_n$. The results are general 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.

Recall that our running example contains an example of splitting where the events in VM_0 are each refined by two events: *selectItem* is refined by both *selectBiscuit* and *selectChoc*, and *dispenseItem* is refined by both *dispenseBiscuit* and *dispenseChoc*. Recall that a renaming function f associated concrete events with abstract events that they refine. For example $f_1(\mathit{selectChoc}) = \mathit{selectItem}$ and $f_1(\mathit{selectBiscuit}) = \mathit{selectItem}$ where f_1 is the renaming function appropriate for mappings between VM_1 and VM_0. In Section 2 we defined a compositional mapping $g_{i,j}$ in terms of the individual f mappings. Thus, for the chain $VM_0 \preccurlyeq_W VM_1 \preccurlyeq_W \ldots \preccurlyeq_W VM_4$, we obtain that $g_{1,4}(\mathit{selectBiscuit}) = g_{1,4}(\mathit{selectChoc}) = \mathit{selectItem}$, and $g_{1,4}(\mathit{dispenseBiscuit}) = g_{1,4}(\mathit{dispenseChoc}) = \mathit{dispenseItem}$, and $g_{1,4}$ is not defined on the remaining events of VM_4.

This section excludes a discussion of temporal properties that use the *enabled* operator, since these require the more elaborate treatment of Section 7.

We begin with a LTL property that will hold for all refinement chains that observe the strategy from Section 2.3. It is restated from [STWW14b].

6.1. General GF property preservation

Lemma 6.1 states that the final machine in the refinement chain must always eventually perform some event relating to an event in the initial machine. 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 6.1. 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 $M_n \models GF(\bigvee_{t \in g_{1,n}^{-1}(\alpha M_0)} [t])$

Observe that if there is no renaming or splitting, then $g_{1,n}$ is the identity function on the events in αM_0, and the disjunction simplifies to be the disjunction of the events in the alphabet of M_0.

6.2. General LTL property preservation (excluding the enabled operator)

The new results in this paper are given in Lemmas 6.3 and 6.4. They concern the preservation of temporal properties that could be introduced at any refinement step and are required to be preseved by subsequent refinement steps. We will see that Lemmas 6.3 and 6.4 are variants of each other. The two subtly different lemmas are needed due to the two ways of dealing with the negation operator in the associated proofs.

We have already observed from the vending machine example that events can be split as shown above and that new events can be introduced during a refinement, e.g., *pay*, *refill*, in VM_2 and VM_3. We aim for LTL properties to hold even though splitting of events may occur and new anticipated and convergent events are being introduced. i.e., we aim for a result that states when a property is introduced in refinement step i then the property must hold for machine M_i and the transformed property also holds at the end of the refinement chain at level n for machine M_n.

To aid the formalisation of the lemmas we introduce Definition 6.2 to define a translation function $trans_g$ with respect to renaming function g as mapping LTL predicates on abstract events to predicates on their corresponding concrete events as follows:

Definition 6.2. Let $g \in B \twoheadrightarrow A$ and define $N = B - dom(g)$

$$trans_g(true) = true \tag{5}$$

$$trans_g([t]) = \bigvee_{n \in N} [n] \; U \bigvee_{y | g(y) = t} [y] \tag{6}$$

$$trans_g(\neg\phi) = \neg trans_g(\phi) \tag{7}$$

$$trans_g(\phi_1 \vee \phi_2) = trans_g(\phi_1) \vee trans_g(\phi_2) \tag{8}$$

$$trans_g(\phi_1 \wedge \phi_2) = trans_g(\phi_1) \wedge trans_g(\phi_2) \tag{9}$$

$$trans_g(\phi_1 \; U \; \phi_2) = trans_g(\phi_1) \; U \; trans_g(\phi_2) \tag{10}$$

$$trans_g(G\phi) = G \; trans_g(\phi) \tag{11}$$

$$trans_g(F\phi) = F \; trans_g(\phi) \tag{12}$$

$$trans_g(p) = \exists s. \mathcal{J}(s, s') \wedge p \text{ is true in } s \tag{13}$$

$$trans_g(X\phi) = \bigvee_{n \in N} [n] \; U(\bigvee_{d \in dom(g)} [d] \wedge X(trans_g(\phi))) \tag{14}$$

The translation is with respect to g since it can be used over a number of refinement steps. When we apply the definition, B and A will be particular alphabets of machines and $\mathcal{J}$ will be the composition of the linking invariant between the two machines. This definition allows us to take into account the new events that are introduced during refinement steps as required in clauses (6) and (14). It is inspired by [WdRF12] and we will reflect on this in Section 8.

Clause(6) translates the occurrence of the abstract event t to the (repeated) occurences of the new events introduced during the refinement ($\bigvee_{n \in N} [n]$) until the occurrence of one of the concrete events that maps to the abstract event t. Note that the new events cannot happen forever since it is a strong until operator. Clause(14) translates $X\phi$ to the occurences of new events until one of the concrete events which do relate to an abstract event occurs, and translation of the property holds in the next state ($X(trans_g(\phi))$).

Clause(13) translates an abstract predicate p on states to a concrete predicate on states that relate to the abstract ones.

For example, consider $B = \alpha(VM_1)$ and $A = \alpha(VM_0)$. In this case $g = f_1$ and $N = \emptyset$. Hence,

$$trans_{f_1}(\phi_A) = G(([selectBiscuit] \vee [selectChoc]) \Rightarrow F([dispenseBiscuit] \vee [dispenseChoc]))$$

as $trans_{f_1}([selectItem]) = [selectBiscuit] \vee [selectChoc]$. Since there are no new events in VM_1 the expression containing the until operator simplifies to the disjunction of the corresponding mapped events. Similarly the translation of the occurrence *dispenseItem* event translates to the disjuction of the concrete dispense events.

The same ϕ_A property translates for a refinement chain as follows: consider $B = \alpha(VM_2)$ and $A = \alpha(VM_0)$. In this case $g = f_2; f_1$ and $N = \{pay, refund\}$. There are new events in VM_2 which have an impact

on ϕ_A. Hence,

$$trans_{f_2;\, f_1}(\phi_A) = G((\bigvee_{n \in N} [n]\ U([selectBiscuit] \vee [selectChoc])) \\ \Rightarrow F([dispenseBiscuit] \vee [dispenseChoc]))$$

It states that new events from VM_2 are permitted to occur until one of the mapped selection events happens the new events are again permitted to occur until one of the concrete dispensing events occurs. Notice that the eventually property is simplified from $F((\bigvee_{n \in N})[n]\, U([dispenseBiscuit] \vee [dispenseChoc]))$ since $F([a]\, U[b]) = F([b])$ as noted in Section 5.

Using the same simple refinement chain from VM_2 to VM_0 an example to illustrate the translation of Clause(13) for property ϕ_C is given as follows:

$$trans_g(\phi_C) \ = \ G(trans_g(item = 2) \Rightarrow trans_g(X(item = 1)))$$

The predicate $trans_g(item = 2)$ becomes $\exists\, item.(\#(chosen) = item \wedge item = 2)$ which simplifies to $\#(chosen) = 2$. Also, $trans_g(X(item = 1))$ simplifies to

$$([pay] \vee [refund]) \\ U([dispenseBiscuit] \vee [dispenseChoc] \vee [selectBiscuit] \vee [selectChoc]) \wedge X(\#(chosen) = 1)$$

In a state where $\#(chosen) = 2$, both the *dispenseBisuit* and *dispenseChoc* concrete events are possible and also one of the new events in N, namely *pay*. Hence, through the translation of $trans_g(X(item = 1))$ any number of *pay* events can occur until one of the dispense events occurs and then in the next state we assert there is a matching concrete state when $item = 1$ holds. There will be a matching concrete state because one of the dispense events will have occurred, i.e., the concrete state of *chosen* will have decreased the number of its elements by one and therefore the linking invariant will hold.

Two main results are presented in this section which identify additional conditions under which an LTL property ϕ will be preserved in a refinement chain that adheres to the development strategy given in Section 2.3. The strategy already makes clear that 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), restriction 6 of the Development Strategy of Section 2.3.

The common additional conditions for both new results are as follows:

Condition 1 ϕ must not contain the enabled (e) operator.

Condition 2 the final machine in the refinement chain must be deadlock-free.

These conditions are enough to ensure that ϕ is preserved through refinement chains in the case where the LTL also does not contain the negation $(\neg)$ operator. In particular, if a property ϕ is established for M_{i-1}, then the resulting system M_n will satisfy the translation of the property: $trans_{g_{i,n}}(\phi)$. Hence, by using Definition 6.2 with $g = g_{i,n}$ we are now able to define the first result: Lemma 6.3.

Lemma 6.3. If $M_i \models \phi$ and ϕ does not contain the enabled (e) and negation $(\neg)$ operators and $M_i \preccurlyeq_W \ldots \preccurlyeq_W M_n$ and $0 \leq i < n$, M_n is deadlock free and M_n does not contain any anticipated events then $M_n \models trans_{g_{i+1,n}}(\phi)$

Note the requirement for the refinement chain to adhere to restriction 6 of the Development Strategy of Section 2.3 we mention this explicitly in the lemma to emphasise it. The second lemma, Lemma 6.4, is similar but subtly different; in order to have a result on the negation operator we need to restrict the composition of the refinement relations $\mathcal{J}$ to be functional. This same restriction was identified by Derrick and Smith in [DS12]. Hence a further condition is needed for this result as follows:

Condition 3
The compositional linking invariant $\mathcal{J}$ must be functional. In other words, $\mathcal{J}(v_1, w) \wedge \mathcal{J}(v_2, w) \Rightarrow v_1 = v_2$

Lemma 6.4. If $M_i \models \phi$ and ϕ does not contain enabled (e) operator and $M_i \preccurlyeq_W \ldots \preccurlyeq_W M_n$ and $0 \leq i < n$, the composition of the refinement relations $\mathcal{J}$ is functional and M_n is deadlock free and M_n does not contain any anticipated events then $M_n \models trans_{g_{i+1,n}}(\phi)$

Note that unlike Lemma 6.1 the application of this lemma requires a property to be established of M_i.

The proofs for the lemmas are given in Appendix A. In order to apply Lemmas 6.3 and 6.4 in practice Condition 1 can be achieved by a simple review and Condition 2 is a mechanical check that can be done automatically using ProB. Condition 3 must also achieved by review, but note that if each of the linking invariants is functional then so is their composition. The proof obligations related to the refinement chain would need to be proved using Rodin or AtelierB.

6.3. Preserving Vending Machine properties

The purpose of this section is to consider the LTL properties of interest for the vending machine development, to highlight that:

- properties need not hold through all refinement steps, only for the machine in the refinement step in which the properties are introduced and again for the machine at the end of the refinement chain which does not contain anticipated events.
- properties can be introduced at different levels of refinement, minimising the need for duplicate LTL properties to be introduced in subsequent refinement steps,

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

$VM_0 \preccurlyeq_W VM_1 \preccurlyeq_W VM_2 \preccurlyeq_W VM_3 \preccurlyeq_W VM_4$

In this case we obtain immediately from Lemma 6.1 that

$$VM_4 \models GF([selectBiscuit] \lor [selectChoc] \lor [dispenseBiscuit] \lor [dispenseChoc]) \tag{15}$$

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 have already introduced four properties which we recap here as follows:

$\phi_A = G([selectItem] \Rightarrow F[dispenseItem])$
$\phi_B = (\neg GF[selectBiscuit]) \Rightarrow G([selectChoc] \Rightarrow F[dispenseChoc])$
$\phi_C = G(item = 2 \Rightarrow (X(item = 1)))$
$\phi_D = G(item = 0 \Rightarrow (X(item = 1)))$

Note that property ϕ_B, and properties ϕ_G, ϕ_H and ϕ_I introduced below, were discussed in the example in [STWW14b] but we restate them here so that it is clear that the new results apply to them.

Properties ϕ_A, ϕ_C and ϕ_D are appropriately introduced in the first development step upon specification of VM_0 since they relate to events in VM_0. We have checked that VM_0 satisfies each of these three properties using ProB. Once we have also established the refinement chain $VM_0 \preccurlyeq_W VM_1 \preccurlyeq_W VM_2 \preccurlyeq_W VM_3 \preccurlyeq_W VM_4$, and that VM_4 is deadlock free we can deduce using Lemma 6.3 that the translations of all three properties hold for VM_4.

For example, from the result for VM_0 that whenever *selectItem* occurs then *dispenseItem* will eventually occur, $VM_0 \models \phi_A$, we obtain from Lemma 6.3 that

$$VM_4 \models G(\ ([selectBiscuit] \lor [selectChoc]) \Rightarrow F([dispenseBiscuit] \lor [dispenseChoc])\ ) \tag{16}$$

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

Property ϕ_B is an appropriate property for first refinement step and $VM_1 \models \phi_B$, which we have checked using ProB. Once again provided we establish the correctness of the refinement chain and demonstrate that VM_4 is deadlock free then we can deduce using Lemma 6.4 that $VM_4 \models trans_{g_{1,4}}(\phi_C)$. Property ϕ_B states that if you do not always eventually have a *selectBiscuit* then you will be able to choose a chocolate and for it to be dispensed. There is a dual property when you do not always have a *selectChoc*.

Observe however that Lemmas 6.3 and 6.4 do not establish that the translation of ϕ properties hold in all refinement machines, only those deadlock free and with no anticipated events. For example, VM_2 does not satisfy the translations of ϕ_A nor ϕ_B since *pay* is anticipated and can be executed infinitely often and hence $VM_2 \not\models trans_{f_2;\, f_1}(\phi_A)$. The same applies for VM_3 since *pay* also remains anticipated in this machine.

There are well-known benefits to introducing a property as early as possible. Firstly, we can minimise duplication of properties. For example, we need not check the following property to be required of VM_1

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

We have already sufficient proof that it is true since $VM_0 \models \phi_A$ and by application of Lemma 6.3.

VM_0 has three states while VM_1 has more states and VM_4 has unboundedly many states. While model checking a machine with three states is trivial, the time taken to perform explicit state model checking grows exponentially with the size of the statespace and model checking an infinite statespace is intractable in this manner. Therefore, it is easier to check ϕ_A on VM_0 than on VM_1 and it is not possible to check it on VM_4. Hence, through the application of our theoretical results we can establish properties of infinite state systems using our development strategy within existing tools.

There may be properties that do not hold early in a refinement chain but do hold later. For example, consider the following property:

$$\phi_E = G([pay] \Rightarrow F([dispenseBiscuit] \vee [dispenseChoc]))$$

The infinite behaviour of *pay* means that ϕ_E is not satisfied in VM_2. Thus, even though ϕ_E satisfies Condition 1, Lemma 6.3 is not applicable in the second refinement step because Condition 2 is not satisfied. However, $VM_4 \models \phi_E$ since *pay* is no longer anticipated, i.e., convergent in VM_4. Hence, the property is not satisfied until the final refinement step. Since VM_4 is infinite state we cannot check this using ProB directly. However, it can be deduced from the result $VM_4 \models \phi_F$ below together with Lines 15 and 16 above.

There may be properties involving anticipated events which hold at an early point of the refinement chain, and hence by the applications of the lemmas also hold at the end of the refinement chain. For example,

$$\phi_F = G([pay] \Rightarrow (G[pay] \vee F([selectBiscuit] \vee [selectChoc]))$$

$VM_2 \models \phi_F$ and so by application of Lemma 6.3 VM_4 also satisfies the translation of ϕ_F. Since *pay* is anticipated, it cannot occur forever in VM_4 (as indicated in Line 15), the disjunct $G[pay]$ in ϕ_F cannot hold in executions of VM_4 and we obtain

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

This combines with the result of Line 16 to yield the result that $VM_4 \models \phi_E$. Hence even though ϕ_E could not be established for VM_4 by direct use of Lemma 6.3, it follows from other results which are established by that lemma and others.

7. Extending LTL properties to include enabled events

In this section we extend the theoretical results to support the inclusion of the enabled ($e(t)$) operator from our LTL grammar in Section 5. First we extend Definition 6.2 to include the following translation clause to the definition of $trans_g$ for the mapping the enabledness of an abstract event to the disjunction of its corresponding concrete events as follows:

Definition 7.1. Let $g \in B \twoheadrightarrow A$ and define $N = B - dom(g)$

$$trans_g(e(t)) = \bigvee_{n \in N} [n] \; U((\bigwedge_{n \in N} \neg e(n) \wedge \bigvee_{y \mid g(y) = t} e(y)) \vee (\bigvee_{n \in N} e(n) \wedge \bigvee_{d \in dom(g)} [d])) \tag{17}$$

This additional clause was inspired by the definition of availability in [WdRF12]. This translation clause means that the occurrence of one of the new events introduced during the refinement ($\bigvee_{n \in N}[n]$) is permitted until either:

1. a stable state is reached (none of the guards of the new events are enabled) and one of the guards of the concrete events that maps to the abstract event t is enabled, or
2. a concrete event in $dom(g)$ that corresponds to an abstract event occurs from an unstable state (where some of the new events are enabled).

To motivate this definition, consider the machines PM_0 and PM_1 of Figure 8. Here we see that the events *dispenseApple* and *dispensePear* are both enabled in PM_0 following a *pay* event, thus for example

machine PM_0
variables $paid \in \{0,1\}$
events
init $\widehat{=}$ $paid := 0$
pay $\widehat{=}$ $paid := 1$
dispenseApple $\widehat{=}$
when $paid = 1$
then $paid := 0$
end
dispensePear $\widehat{=}$
when $paid = 1$
then $paid := 0$
end
end

machine PM_1
refines PM_0
variables $paid$, $offers$
invariant $paid \in \{0,1\} \wedge offers \subseteq \{apple, pear\}$
variant $2 - \#(offers)$
events
init $\widehat{=}$ $offers := \{\} \parallel paid := 0$
pay $\widehat{=}$ $paid := 1$
dispenseApple $\widehat{=}$ **when** $apple \in offers$ **then** $offers := \{\} \parallel paid := 0$ **end**
dispensePear $\widehat{=}$ **when** $pear \in offers$ **then** $offers := \{\}$ **end**
fetch $\widehat{=}$ **when** $paid = 1 \wedge \{apple, pear\} \not\subseteq offers$ **then if** $apple \notin offers$ **then** $offers := offers \cup \{apple\}$ **else** $offers := offers \cup \{pear\}$ **end** **end**
end

Fig. 8. Machine PM_0 and its refinement PM_1

the predicate $e(dispensePear)$ holds following the sequence $[pay]$, i.e., $PM_0 \models \phi_G$, where $\phi_G = G([pay] \Rightarrow X(e(dispensePear))$. The refinement machine PM_1 maintains a set *offers* of pieces of fruit which are available, and also introduces a new event *fetch* which adds items to that set.

Consider the translation of ϕ_G, in particular, that of $e(dispensePear)$. Consider the clause $(\bigwedge_{n\in N} \neg e(n) \wedge \bigvee_{y|g(y)=t} e(y))$ of Definition 7.1 regarding reaching the stable state. For the translation of $e(dispensePear)$, this is simplified to $\neg e(fetch) \wedge e(dispensePear)$. Following the occurrence of *pay*, it is not immediately the case that $e(dispensePear)$ holds for PM_1, but if a user of that machine allows *fetch* to occur until it is no longer possible, corresponding to the sequence of events $[pay, fetch, fetch]$ then indeed $e(dispensePear)$ is true following that sequence. This corresponds to clause (1).

With respect to the clause $(\bigvee_{n\in N} e(n) \wedge \bigvee_{d\in dom(g)}[d])$ regarding unstable states, in our example becomes $e(fetch) \wedge ([pay] \vee [dispenseApple] \vee [dispensePear])$, consider the following: the introduction of the new event *fetch* means that there are some intermediate states in which *dispensePear* is not yet enabled while the sequence of *fetch* events is ongoing. In our example the event *dispenseApple* becomes enabled after a single *fetch*, and if a user selects that in preference to waiting for *fetch* to occur, corresponding to the sequence $[pay, fetch, dispenseApple]$, then a different path has been taken and the offer of *dispensePear* is not made anywhere along this path. This illustrates that some other event might occur before the event of interest becomes enabled in the refinement machine. We only wish to allow for this situation of *dispensePear* not yet being enabled if some further new event is still enabled, hence the inclusion of $\bigvee_{n\in N} e(n)$ in clause (2). If no further new events are possible, then an event such as *dispensePear* enabled in the abstract machine should also be enabled in the refinement machine.

In general, observe that if there are no new events N, then $\bigvee_{n\in N}[n] = false$, $\bigvee_{n\in N}(e(n)) = false$, $\bigwedge_{n\in N} \neg e(n) = true$, and hence $trans_g(e(t))$ simplifies in this case to $\bigvee_{y|g(y)=t} e(y)$

For example, in VM_1 there are no new events, so $trans_{f_1}(e(dispenseItem))$ simplifies to $e(dispenseBiscuit) \vee e(dispenseChoc)$.

Now we can examine whether the previous results from Section 6 remain applicable if we permit the enabled operator. In fact it turns out that they are not applicable; we need to strengthen the antecedents of the lemmas. We illustrate this on Lemma 6.3, but with the enabled operator e permitted.

Consider the new LTL property which is true of VM_4:

$$\phi_I = G([selectBiscuit] \Rightarrow (Xe(dispenseBiscuit)))$$

and the refinement chain $VM_4 \preccurlyeq_W VM_5$ where VM_5 is given in Figure 9. It is exactly the same as VM_4 except that the availability of the *dispenseBiscuit* is less, since its guard is strengthened so that a biscuit can only be dispensed once a biscuit *and* a chocolate are chosen. Hence, application of a relaxed Lemma 6.3

machine VM_5
... exactly as VM_4 apart from the following event:
dispenseBiscuit $\widehat{=}$ **status** : ordinary
refines dispenseBiscuit
when $credit > 0 \wedge biscuitStock > 0 \wedge chosen = \{biscuit, choc\}$
then ... || $chocStock := chocStock - 1$ **end**

Fig. 9. VM_5

would state that VM_5 satisfies the trivial translation of ϕ_I. However, this is not the case. For example, the infinite trace

$$\langle (0, \emptyset, FALSE), pay, (1, \emptyset, FALSE), pay, (2, \emptyset, FALSE), selectBiscuit, (2, \{biscuit\}, FALSE), \ldots \rangle$$

does not satisfy $trans_{f_5}(\phi_I) = \phi_I$ since *dispenseBiscuit* is not enabled in the fourth state; *selectChoc* has not yet occurred to increase the value of the set *chosen* appropriately.

This example illustrates that we need a stronger notion of refinement that deals with the liveness of events so that we can ensure that the enabledness of events is guaranteed through the refinement chain. In fact there is a proof obligation within Event-B requiring certain events from machine M to be enabled in the adjacent refinement machine M' [Abr10]. We define this proof obligation referred to as `S_NDF` in the same way as in [Abr10] but we include explicitly the event t it applies to, the corresponding concrete events, and the new events N_{i+1}.

S_NDF: Strong Deadlock Freedom with new events If an abstract event is enabled then either some concrete refinement of it is enabled, or some new event is enabled.

$$\vdash \quad I(v_i) \wedge J_{i+1}(v_i, v_{i+1}) \wedge G_t(v_i) \quad \Big| \quad \mathbf{S\_NDF}(t)$$
$$\bigvee_{y | f_{i+1}(y) = t} G_y(v_{i+1}) \vee \bigvee_{n \in N_{i+1}} G_n(v_{i+1})$$

Observe that the refinement from PM_0 to PM_1 in Figure 8 satisfies this proof obligation.

Let us examine this `S_NDF` proof obligation with respect to all the events in the vending machine development from VM_0 to VM_4. Consider again the refinement chain from VM_0 to VM_1. In this refinement step `S_NDF` holds for both the *selectItem* and the *dispenseItem* events.

For the refinement step between VM_1 and VM_2 `S_NDF` holds for *selectBiscuit*, *selectChoc*, *dispenseBiscuit* and *dispenseChoc* since in the refinement step even though the guards are strengthened, the new event *pay* can occur, hence discharging `S_NDF`. In the step between VM_2 and VM_3 again the new event *refill* supports the discharging of the `S_NDF` obligations for all the events in $\alpha(VM_2)$.

In the refinement step from VM_3 to VM_4, `S_NDF` holds vacuously for *refund* and `S_NDF` for *selectBiscuit*, *selectChoc*, *dispenseBiscuit*, *dispenseChoc*, *refill* simply holds by data refinement. For example in the proof for `S_NDF`(*dispenseBiscuit*) the strengthened guard $biscuitStock > 0$ is established by the following:

$J_4(v_3, v_4)$: $(biscuit \in stocked \Rightarrow biscuitStock > 0)$
$G_{db}(v_3)$: $\wedge\ biscuit \in stocked$
$G_{db}(v_4)$: $\Rightarrow biscuitStock > 0$

In VM_4 the *pay* event has a strengthened guard which stops pay happening forever, this is needed in order to turn it into a convergent event but of course this then means that `S_NDF` does not hold for it since its guard is strengthened and no new events are introduced in VM_4.

In general, once a machine has an anticipated event that is always enabled, then to make that event convergent its guard will need to be strengthened in a way that fails to meet `S_NDF`. Therefore, the anticipated approach of deferring convergence proofs is incompatible with the use of `S_NDF` for each event in all of the refinement steps. Therefore, we need to identify a weaker form of `S_NDF`. The inspiration for its definition comes from the same principle that we adopt for deadlock freedom, i.e., we do not need deadlock freedom to be a property that is true of each refinement step but it simply needs to hold at the end of a refinement chain.

Let us therefore also consider S_NDF to be a proof obligation that needs to hold between the refinement step in which an LTL property based on the enabled operator is introduced and the final machine in the refinement chain. It does not need to hold for all the intermediate steps.

We define a new proof obligation referred to as $\texttt{S\_NDF}_{i,j}(t)$ to relate the enabledness of t in M_i with its concrete refinement(s) in M_j as follows:

Definition 7.2. If an abstract event is enabled for M_i and $i < j$ then either some concrete refinement of it is enabled in M_j, or some new event is enabled in M_j.

$$\vdash \quad I(v_i) \wedge \mathcal{J}_{i,j}(v_i, v_j) \wedge G_t(v_i) \quad \Big| \quad \mathbf{S\_NDF}_{i,j}(t)$$
$$\bigvee_{y \mid g_{i+1,j}(y)=t} G_y(v_j) \vee \bigvee_{n \in N_j} G_n(v_j)$$

With this extra condition on the preservation of enabledness of events, we are able to extend the main results about the preservation of LTL properties to include the enabled operator (i.e., remove the restriction on its inclusion) as follows:

Lemma 7.3. If $M_i \models \phi$ and and ϕ does not contain the negation ($\neg$) operator and $M_i \preccurlyeq_W \ldots \preccurlyeq_W M_n$ and $0 \leq i < n$ and $\bigwedge_{t \in \phi} \texttt{S\_NDF}_{i,n}(t)$ and M_n is deadlock free and M_n does not contain any anticipated events then $M_n \models \mathit{trans}_{g_{i+1,n}}(\phi)$

Lemma 7.4. If $M_i \models \phi$ and $M_i \preccurlyeq_W \ldots \preccurlyeq_W M_n$ and $0 \leq i < n$ and $\bigwedge_{t \in \phi} \texttt{S\_NDF}_{i,n}(t)$ and the composition of the refinement relations $\mathcal{J}$ is functional and M_n is deadlock free and M_n does not contain any anticipated events then $M_n \models \mathit{trans}_{g_{i+1,n}}(\phi)$

The proofs of these lemmas is given in Appendix A.

Consider again the property ϕ_I. It is satisfied for VM_1 and VM_2, but not for VM_3, since if $\mathit{biscuit} \notin \mathit{stocked}$ in VM_3 then *selectBiscuit* can be followed by an infinite sequence of *pay* events with no *refill* event, and so $e(\mathit{dispenseBiscuit})$ never becomes true. However $\texttt{S\_NDF}_{1,4}$ holds, VM_4 has no anticipated events (thus *pay* cannot occur for ever) and Lemma 7.3 yields that $VM_4 \models \mathit{trans}_{g_{2,4}}(\phi_I)$. Expanding out $\mathit{trans}_{g_{2,4}}(\phi_I)$ simplifies to:

$$\begin{aligned} G([\mathit{selectBiscuit}] \Rightarrow X(&([\mathit{pay}] \vee [\mathit{refund}] \vee [\mathit{refill}]) \\ &U(\ (\neg e(\mathit{pay}) \wedge \neg e(\mathit{refund}) \wedge \neg e(\mathit{refill}) \wedge e(\mathit{dispenseBiscuit})) \\ &\quad \vee \\ &\quad ((e(\mathit{pay}) \vee e(\mathit{refund}) \vee e(\mathit{refill})) \\ &\quad\ \wedge ([\mathit{selectBiscuit}] \vee [\mathit{selectChoc}] \vee [\mathit{dispenseBiscuit}] \vee [\mathit{dispenseChoc}])))))) \end{aligned}$$

This states that, following *selectBiscuit*, the new events *pay*, *refund* and *refill* may occur until either none of them are enabled any more, and *dispenseBiscuit* is enabled; or any of the *select* or *dispense* might occur while the new events are still enabled. Thus if the customer is prepared to wait for *dispenseBiscuit*, and does not engage in any of the other events, then *dispenseBiscuit* will eventually become enabled.

The inclusion of the enabled operator allows for the expression of useful fairness properties. Fairness assumptions are often required when model checking systems for liveness properties, so as to dismiss infinite behaviours (that may violate liveness properties) that are judged to be unfair. Here fair typically means that "if a certain choice is possible sufficiently often, then it is sufficiently often taken" [AFK88]. Strong (resp. weak) event fairness frame fairness properties in the context of event-based formalisms such as Event-B [Lam00, PV01, SLDW08, Mur13]. Given a set of events Σ, weak event fairness with respect to Σ demands that any continually enabled event from Σ must occur infinitely often, whereas strong event fairness with respect to Σ demands that infinitely often enabled events occur from Σ infinitely often.

$$WEF = \bigwedge_{z \in \Sigma} (FGe(z) \Rightarrow GF[z])$$

$$SEF = \bigwedge_{z \in \Sigma} (GFe(z) \Rightarrow GF[z])$$

Consider the following new properties, with $\Sigma = \{selectChoc, selectBisc, dispenseChoc, dispenseBisc\}$:

$\phi_J = WEF \Rightarrow GF[dispenseChoc]$

Supposing each continually enabled event occurs infinitely often, then *dispenseChoc* occurs infinitely often.

$\phi_K = SEF \Rightarrow GF[dispenseChoc]$

Supposing each infinitely often enabled event occurs infinitely often, then *dispenseChoc* occurs infinitely often.

VM_1 satisfies both of these properties. One infinite trace that violates $GF[dispenseChoc]$ is the following: $\langle selectBiscuit, dispenseBiscuit\rangle^\omega$. In this trace *selectChoc* is continuously enabled but never taken and is thus deemed unfair. In fact the only traces that violate the property $GF[dispenseChoc]$ are those that (unfairly) continuously deny either *selectChoc* or *dispenseChoc*; $GF[dispenseChoc]$ is satisfied under the assumption of both strong and weak event fairness. From our results we may deduce that *dispenseChoc* is guaranteed to occur infinitely often under (a translation of) the assumption of weak event fairness and hence is also satisfied in VM_4.

8. Related work

One of the few papers to discuss LTL preservation in Event-B refinement is Groslambert [Gro06]. The LTL properties were defined in terms of predicates on system state where we also include events' occurrences and enableness in our formulation. 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 four main lemmas in our paper: Lemmas 6.3, 6.4, 7.3 and 7.4 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. This is possible by refining away paths to deadlock.

Groslambert deals with new events via stuttering and leaves them as visible events in a trace. This is why the LTL operators he uses do not include the next operator (X). As new events may happen this may violate the X property to be checked. On the other hand Plagge and Leuschel in [PL10] permit the use of the X operator since they treat the inclusion of new events as internal events which are not visible. In this paper we are also able to deal with the next operator.

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 Mussat in an early paper, [AM98], introduced proof obligations to deal with dynamic constraints in classical B. In a more recent paper [HA11] 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 [BAA+08] 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.

In [BDJK00, BDJK01], the authors propose an approach of combining proofs and model-checking verification techniques to reason about properties formulated in Propositional Linear Temporal Logic (dynamic properties). Similar to our approach, the PLTL properties are first specified at the abstract level (and verified by PLTL model-checking), and gradually reformulated during refinement. Several transformation patterns are mentioned with accompanying sufficient conditions that required to be discharged. As a result, the supported properties are restricted to those that are transformable, e.g., *dynamic invariants*, *leads-to*, or *until* properties. In our work, the translation function $trans_g$ is total and can be used to transform any property over a number of refinement steps. Moreover, our temporal language is also richer than PLTL, including the events' occurrences and enableness. In [BDJK00], fairness assumptions are required for reasoning about dynamic properties using PLTL model-checking tools. As a result, the transformation of dynamic properties is conditional on the presevation of fairness assumptions which is consider as future work in [BDJK00]. Since fairness assumptions can be encoded using events' enableness and occurrences in our language, they can be transformed as a part of the temporal properties. Furthermore, the transformation of fairness assump-

tions using our approach can be the basis for future research on preservation of fairness assumptions via refinement.

In [HH13], the Unit-B modelling method is proposed as a combination of UNITY [CM89] and Event-B [Abr10]. Unit-B extends the Event-B events with coarse- and fine-schedules, a generalisation of weak- and strong-fairness assumptions, stating how often an event has to occur. Liveness reasoning in Unit-B is borrowed from the UNITY logic and is adapted for the newly introduced scheduling assumptions. Refinement in Unit-B amounts to proving that the liveness assumptions are strengthened and the rules are based on reasoning about progress properties.

To compare with our work, the LTL used in Unit-B is defined in term of state variables and does not included references to event occurrences $[t]$ or event enableness $e(t)$. In fact the Unit-B traces are defined as sequence of states whereas we explicitly include executions of events into the definition of traces. Moreover, at the moment, Unit-B only considers superposition refinement, where variables of the abstract machine are retained in the concrete machine. As a result, liveness properties are preserved in LTL "as they are". Our work is more general by allowing data refinement, transforming the liveness properties accordingly via the translation function $trans_g$. Another difference is the fact that the preservation of properties in our work can stretch across several refinements, i.e., we can postpone the proof of properties preservation for some refinements, until all the necessary conditions (e.g., deadlock-free, event convergence) can be established. In Unit-B, it is required that all the necessary conditions are discharged at every refinement step, so that all liveness properties are maintained. Note also that Unit-B has a single general refinement rule preserving all liveness properties. With the inclusion of event enableness in our language, we have additional conditions for refinement depending on the properties we want to preserve.

In order to prove liveness properties relying on reasoning about convergence and deadlock-freeness, we made an assumption on event execution for each machine (called minimal progress in [HH13]): if the system is not deadlocked then one of the enabled events will be executed. In Unit-B, the assumption on event execution is finer-grained. Individual scheduling assumptions can be attached to each event. As a result, Unit-B can express stronger fairness assumptions and subsequently can prove more liveness properties relying on these assumptions.

Approaches to LTL preservation through refinement are wider than simply Event-B. For example, Derrick and Smith [DS12] discuss the preservation of LTL properties in the context of Z refinement, extending 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 refinement steps. Our paper does permit new events to be introduced during refinement steps and the conditions that are required to hold at each refinement step do not provide any restrictions on the properties that can be expressed.

9. Discussion and future work

The paper has provided the theoretical foundation to justify the use of LTL properties in Event-B development. It provides 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 [Gro06]. Moreover, the class of LTL operators that is supported by the theory also extends the LTL presented in [Gro06] and are those supported by ProB. Notably, our inclusion of the enabled operator, and its translation in Definition 7.1, (reflecting Lowe's available operator [Low08]) allows for the expression of temporal properties under various notions of fairness [Mur13, WdRF12]. The expression of fairness properties are typically required in proving pertinent liveness properties.

From the results in this paper, a complex (even infinite branching/infinite state) concrete machine can be shown to satisfy such a property by first model checking a small abstract machine using ProB and following a particular refinement strategy, discharging the proofs in Rodin or Atelier B.

The paper has removed one of the awkward restrictions of our previous work, which imposed restrictions on the temporal properties in terms of being β-dependent, which determined when a temporal property of interest should be introduced into the development chain.

In this paper we restricted two of the lemmas, Lemma 6.4 and Lemma 7.4, to refer to developments whose linking invariants were required to be functional in order to deal with negation in LTL properties. Event-B developments generally contain functional invariants, since abstract machines usually contain less

machine M
variables n
invariant $n \in 1 \ldots 4$
events
init $\widehat{=}$ $n := 1$
open $\widehat{=}$ **when** $n = 1$ **then** $n := n + 1$ **end**
close $\widehat{=}$ **when** $n = 2$ **then** $n :\in \{1, 3\}$ **end**
wedgeOpen $\widehat{=}$ **when** $n = 3$ **then** $n := n + 1$ **end**
alarm $\widehat{=}$ **when** $n = 4$ **then** *skip* **end**
end

Fig. 10. M

information than their refinements, and thus functional mappings in this case make sense. There are of course examples (e.g., the alternating bit protocol of [Abr10]) where this is not the case, but these are rare.

The novelty of the results, which is a clear extension from [STWW14b], is the requirement to establish strong deadlock freedom `S_NDF`, in certain circumstances, in order to guarantee the preservation of LTL properties that include the enabled operator. The conditions of the lemmas required to hold and the proof obligations required lend themselves to the practical application of Event-B. We are looking into how to provide the new `S_NDF` rule over a number of refinement steps in terms of chaining single step `S_NDF` proof obligations for the corresponding events, so that this proof obligation can be tool supported.

The proofs in this paper were defined on infinite traces. The proofs are not defined in terms of a CSP semantics model. Our work in [STW14] provided a CSP traces divergences and infinite traces semantics for Event-B. Our ongoing work is combining these results with those in this paper in order to provide a cohesive process algebra underpinning for Event-B and the preservation of LTL properties. It should not be surprising that the underpinning will be in terms of the refusal traces semantic model of CSP; Definitions 6.2 and 7.1 were based on [WdRF12, Low08], which addressed the preservation of LTL properties through refusal traces refinement. (In [Low08] temporal operators were defined in terms of the set of finite and infinite refusal traces they allow.)

We have focused in this paper on a refinement strategy that achieved convergence of all its new events by the end. We could also consider the impact on temporal property preservation in refinement chains where anticipated events remain present in the final machine. We have also yet to deal with extending the work to merging events in a refinement chain. Consider the example in Figure 10 the events *open* and *wedgeOpen* meet the conditions to allow them to be merged. In particular they have the same body $n := n + 1$. The LTL property $G(open \Rightarrow F(close))$ is satisfied by M. However, if *open* and *wedgeOpen* are merged then this property no longer holds. Hence, the results of the paper do not apply. A different approach would be required.

Acknowledgments. Thanks to Thierry Lecomte for discussions about Event-B and its practical use. We are also grateful to the reviewers for their thorough and detailed comments on this paper.

A. Proofs of Lemmas

In this appendix we provide the proofs for the main lemmas: Lemma 6.1, Lemma 6.3, Lemma 6.4, Lemma 7.3 and Lemma 7.4. We also provide additional supporting definitions and lemmas.

Proof of Lemma 6.1 This proof is restated from [STWW14a] Since M_n does not deadlock consider an infinite trace u of M_n. Let $u = u \upharpoonright (C_1 \cup \ldots \cup C_n \cup O_0)$ since M_n does not have any anticipated events. We aim to prove that $u \upharpoonright O_0$ is infinite. If $u \upharpoonright C_1 \cup \ldots \cup C_n$ is finite then $u \upharpoonright O_0$ is infinite. If $u \upharpoonright C_1 \cup \ldots \cup C_n$ is infinite then $u \upharpoonright O_0$ is infinite by Theorem 4.3. Hence, since $u \upharpoonright O_0$ is infinite then the ocurrence of one of the events in M_0 will be always eventually be possible. □

The following lemmas: Lemma A.1 and Lemma A.2, provide results that allow us to demonstrate the relationship between an infinite trace u_a preserving a temporal property and a corresponding trace u_c preserving a transformed temporal property. The two lemmas are subtly different and both are required due

to the subtleties in the proof when using the proposition and negation operators. The lemmas are required in order to prove Lemma 6.3 and Lemma 6.4 presented in Section 6 and proved below.

Lemma A.1. If ϕ does not contain the enabled e and the negation $\neg$ operators and $\alpha(\phi) \subseteq ran(g)$ and $u_a \ R \ u_c$ then $u_a \models \phi \Rightarrow u_c \models trans_g(\phi)$

Proof: By structural induction over the structure of ϕ.
Base Cases:
Consider where ϕ is true. Trivially holds.

Consider where ϕ is $[t]$: Let $u_a \ R \ u_c$ and $u_a = \langle s_0, t_0, s_1, t_1, \ldots \rangle$

If $u \models [t]$ then $t_0 = t$ by definition of $[t]$

Since $u_c \upharpoonright dom(g)$ is infinite then there is a first t'_k such that $t'_k \in dom(g)$.

So $g(t'_k) = t_0$ since $u_a \ R \ u_c \wedge \forall j < k . t'_j \in N$

So $u_c \models \bigvee_{n \in N} [n] \ U \bigvee_{y | g(y) = t} [y]$

So $u_c \models trans_g([t])$ by trans definition

Consider where ϕ is p: If $u_a \models \phi$ then p is true in s_0 by definition $\mathcal{J}(s_0, s'_0)$ holds from the definition of $u_a \ R \ u_c \Rightarrow u_c \models \exists \, s_0 . \mathcal{J}(s_0, s'_0) \wedge p \in s_0$.

Inductive Cases: Case $\phi_1 \vee \phi_2$:

$u_a \models \phi_1 \vee \phi_2$
$\Rightarrow u_a \models \phi_1 \vee u_a \models \phi_2$
$\Rightarrow u_c \models trans_g(\phi_1) \vee u_c \models trans_g(\phi_2)$
$\Rightarrow u_c \models trans_g(\phi_1 \vee \phi_2)$

Same for $\phi_1 \wedge \phi_2$.

Case $\phi_1 \ U \ \phi_2$:
If $u_a \models \phi_1 \ U \ \phi_2 \Leftrightarrow \exists \, k \geq 0 . \forall \, i < k . u^i_a \models \phi_1 \wedge u^k_a \models \phi_2$ by definition of U.
k is the k^{th} state of u_a and ϕ_2 holds for the path starting from that state, the point where ϕ_2 is reached.
Let l be the least value such that $u^k_a \ R \ u^l_c$ holds. Then $\forall \, j < l . u^j_c \ \exists \, i < k . u^i_a \ R \ u^j_c$. Then $u^i_a \models \phi_1$, so by ind hyp $u^j_c \models trans_g(\phi_1)$. Also since $u^k_a \models \phi_2$ by ind hyp $u^l_c \models trans_g(\phi_2)$. Therefore, $u_c \models trans_g(\phi_1 \ U \ \phi_2)$.

Case $X\phi$: If $u_a \models X\phi$ and $u_a = \langle s_0, t_0, s_1, t_1, \ldots \rangle$ then $u^1_a = \langle s_1, t_1, \ldots \rangle \models \phi$ by definition of X.
Let $u_a \ R \ u_c$. Since $u_c \upharpoonright dom(g)$ is infinite then there is a first t'_k such that $t'_k \in dom(g)$.
Let $u_c = \langle s'_0, t'_0, \ldots, s'_{k-1}, t'_{k-1} \rangle \frown \langle s'_k, t'_k, s'_{k+1}, t'_{k+1}, \ldots \rangle$.
So $u^1_a \ R \ u^{k+1}_c$ and by ind hyp $u^1_a \models \phi \Rightarrow u^{k+1}_c \models trans_g(\phi)$.
Since $\forall \, i < k \ u^i_c \models \bigvee_{n \in N} [n] \wedge u^k_c \models \bigvee_{d \in dom(g)} [d] \wedge X(trans_g(\phi))$ since $t'_k \in dom(g)$ by definition of X for u^k_c.
Therefore $\exists \, k \geq 0 . \forall \, i < k . u^i_c \models \bigvee_{n \in N} [n] \wedge (u^k_c \models \bigvee_{d \in dom(g)} [d] \wedge X(trans_g(\phi)))$. Hence by definition of $trans_g$
$u_c \models trans_g(\phi)$. □

Lemma A.2. If ϕ does not contain the enabled e operator and $\alpha(\phi) \subseteq ran(g)$ and $u_a \ R \ u_c$ and $\mathcal{J}$ is functional then $u_a \models \phi \Leftrightarrow u_c \models trans_g(\phi)$

Proof: By structural induction over the structure of ϕ.
Base Cases:
Consider where ϕ is true. Trivially holds.

Consider where ϕ is $[t]$:
Consider "$\Rightarrow$" as in Lemma A.1.

Consider "⇐": Let $u_a\ R\ u_c$ and $u_a = \langle s_0, t_0, s_1, t_1, \ldots\rangle$

If $u_c \models \bigvee_{n\in N} [n]\ U \bigvee_{y|g(y)=t} [y]$

$\Rightarrow \exists\, k \geq 0. \forall\, i < k. u_c^i \models \bigvee_{n\in N} [n] \wedge u_c^k \models \bigvee_{y|g(y)=t} [y]$ (definition of U)

$\Rightarrow i < k \Rightarrow \bigvee_{n\in N} (t_i' = n) \wedge t_k' \in \{y \mid g(y) = t\}$

$\Rightarrow i < k. t_i' \notin dom(g) \wedge t_k' \in dom(g) \wedge g(t_k') = t = t_0$

$\Rightarrow u_a \models [t]$

Consider where ϕ is p:
Consider"⇒" as in Lemma A.1.
Consider "⇐": If $u_c \models \exists! s_o. \mathcal{J}(s_0, s') \wedge p \in s_0$, since $\mathcal{J}$ is functional. Now consider $u_a\ R\ u_c$ then u_a has an initial state s_0 which is related through $\mathcal{J}$ to the concrete state and therefore p is true in s_0, thus $u_a \models p$.

Inductive Cases: Case $\neg\phi$:

$u_a \models \neg\phi$

$\Leftrightarrow \neg u_a \models \phi$

$\Leftrightarrow \neg(u_c \models trans_g(\phi))$ by ind hyp and requires $\Leftrightarrow$

$\Leftrightarrow u_c \models \neg trans_g(\phi)$

$\Leftrightarrow u_c \models trans_g(\neg\phi)$

Case $\phi_1 \vee \phi_2$: Similar to Lemma A.1 with $\Rightarrow$ replaced by $\Leftrightarrow$ in the steps.
Same for $\phi_1 \wedge \phi_2$.
Case $\phi_1\ U\ \phi_2$:
Consider "⇒" as in Lemma A.1.
Consider "⇐": If $u_c \models trans_g(\phi_1 U \phi_2)$ then $u_c \models trans_g(\phi_1)\ U\ trans_g(\phi_2)$ by definition of $trans_g(U)$.
Hence $\exists\, k \geq 0. \forall\, i < k. u_c^i \models trans_g(\phi_1) \wedge u_c^k \models trans_g(\phi_2)$ by definition of U.
Let l be the least value such that $u_a^l\ R\ u_c^k$ holds. Then $\forall\, j < l. u_a^j \models \phi_1 \wedge u_a^l \models \phi_2$ by ind hyp. Therefore, $u_a \models \phi_1\ U\ \phi_2$.

Case $X\phi$:
Consider "⇒" as in Lemma A.1.
Consider "⇐": If $u_c \models trans_g(X\phi)$ then $u_c \models \bigvee_{n\in N} [n] U (\bigvee_{d\in dom(g)} [d] \wedge X trans_g(\phi))$ by definition of $trans_g(X\phi)$
⇔
$\exists\, k \geq 0. \forall\, i < k. u_c^i \models \bigvee_{n\in N} [n] \wedge u_c^k \models \bigvee_{d\in dom(g)} [d] \wedge X\ trans_g(\phi)$ by definition of U.
Let $u_c = \langle s_0', t_0', s_1', t_1', \ldots, s_k', t_k', \ldots\rangle$. So $t_i \in N \forall\, i < k$, $t_k \in dom(g)$ and $u_c^{k+1} \models trans_g(\phi)$ by the definition of X).
Also given $u_a\ R\ u_c$ then $u_a^1\ R\ u_c^{k+1}$ then $u_a^1\ R\ u_c^{k+1}$ by ind hyp $u_a^1 \models \phi$ therefore $u_a \models X\phi$. □

The next lemma allows us to determine that there is a relationship between infinite concrete paths, which are infinite due to convergent and ordinary (and not anticipated) events, and an abstract path such that the refinement relation R introduced in Definition 4.1 holds.

Lemma A.3. If $M \preccurlyeq_W \ldots \preccurlyeq_W M'$ and u' is an infinite path of M' with no anticipated events, then $\exists\, u. u$ is a path of M and $u\ R\ u'$.

Proof: Let $u' = \langle s_0', t_0', s_1', t_1', \ldots\rangle$

$r(s_0, u') = \langle s_0, g(t_0')\rangle \frown r(s_1, \langle s_1', t_1', \ldots\rangle)$ if $t_0' \in dom(g)$ for s_1 where $\mathcal{J}(s_1, s_1')$

$r(s_0, u') = r(s_0, \langle s_1', t_1', \ldots\rangle)$ if $t_0' \notin dom(g)$ where $\mathcal{J}(s_0, s_1')$

Let $u = r(s_0, u')$. then by construction $u\ R\ u'$. □

Proof of Lemma 6.3 Let u' to be an infinite execution of M_n since M_n is deadlock free then M_n **sat** $CA(g^{-1}_{i+1,n}(C_{i-1}) \cup \ldots \cup C_n\ ,\ g^{-1}_{i+1,n}(O_i))$ thus $u' \upharpoonright dom(g_{i+1,n})$ is infinite since M_n has no anticipated events. From Lemma A.3 u is a path of M_i such that $u\ R\ u'$.
Since $M_i \models \phi$ then $u \models \phi$. Thus, from Lemma A.1 $u' \models trans_{g_{i+1,n}}(\phi)$ thus $M_n \models trans_{g_{i+1,n}}(\phi)$ as required. □

Note that $u' \upharpoonright dom(g_{i,n})$ being infinite is key within the proof in order to be able to establish the conditions for R to hold. Recall the example of $u_1 = \langle a, a, a, a, \ldots\rangle$ and $u_2 = \langle b, c, b, c, b, c, c, c, c, c, c, \ldots\rangle$ where b maps to a, and c is a new event. $u_1 \models GF(a)$ but $u_2 \not\models GF(b)$, i.e. the transformed property is not satisfied by u_2 since u_2 only has finitely many events in u_1 that it matched.

Proof of Lemma 6.4 The proof of this lemma is almost identical to Lemma 6.3, noting that the existence of a path does not rely on the fact that $\mathcal{J}$ is functional. But since $\mathcal{J}$ is functional the path will be unique and we are able to apply Lemma A.2 in the penultimate step so that $u' \models trans_{g_{i+1,n}}(\phi)$ thus $M_n \models trans_{g_{i+1,n}}(\phi)$ as required. □

The remainder of the appendix provides the proofs that support the most general results in the paper, i.e., Lemmas 7.3 and 7.4. We begin as above by providing a supporting lemma, Lemma A.4 to demonstrate the relationship between an infinite trace u_a preserving a temporal property which refers to the enabled operator and a corresponding trace u_c preserving a transformed temporal property. In this lemma we have been more explicit with our labelling with regard to which refinement steps are being used, i.e., between M_i and M_j, thus it is clearer between which refinement steps the strong deadlock freedom proof obligation needs to hold. We have also labelled the compositional function mapping g more explicitly for consistency within the proof.

Lemma A.4. If u_a is a path of M_i and u_c is a path of M_j such that $u_a\ R\ u_c$ and $u_a \models e(t)$ and $u_c \upharpoonright dom(g_{i+1,j})$ is infinite and $\texttt{S\_NDF}_{i,j}(t)$ holds $\Rightarrow u_c \models trans_{g_{i+1,j}}(e(t))$

Proof: Consider $u_a\ R\ u_c$ such that $u_c = \langle s'_0, t'_0, s'_1, t'_1, \ldots\rangle$ and $u_a = \langle s_0, t_0, \ldots\rangle$.
Then let $t'_0, t'_1, \ldots, t_{k-1} \in N$ and $t'_k \notin N$, i.e., $t'_k \in dom(g_{i+1,j})$ and $N = \alpha(M_j) - dom(g_{i+1,j})$
Then $\forall\, i < k. u^i_c \models \bigvee\limits_{n \in N} [n]$
Thus from Definition 7.1 we are required to prove that

$$u^k_c \models ((\bigwedge_{n\in N} \neg e(n) \wedge \bigvee_{y \mid g_{i+1,j}(y)=t} e(y)) \vee (\bigvee_{n\in N} e(n) \wedge \bigvee_{d \in dom(g_{i+1,j})} [d]))$$

Note that $u_a \models e(t) \Rightarrow G_t(s_0)$.
We have $\mathcal{J}(s_0, s'_k)$ since $t_i \in N$, $i < k$ and $u_a\ R\ u_c$.
Since $\texttt{S\_NDF}_{i,j}(t)$ we have by Definition 7.2 that $\mathcal{J}(s_0, s'_k) \wedge G_t(s_0) \Rightarrow \bigvee\limits_{y \mid g_{i+1,j}(y)=t} e(y)(s'_k) \vee \bigvee\limits_{n\in N} e(n)(s'_k)$.
It follows that $\bigvee\limits_{y \mid g_{i+1,j}(y)=t} e(y)(s'_k) \vee \bigvee\limits_{n\in N} e(n)(s'_k)$

$$\Rightarrow\ (\bigvee_{y \mid g_{i+1,j}(y)=t} e(y)(s'_k) \wedge \neg \bigvee_{n\in N} e(n)(s'_k)) \vee \bigvee_{n\in N} e(n)(s'_k) \qquad (\text{since } A \vee B = (A \wedge \neg B) \vee B)) \tag{18}$$

$$\text{Also } \bigvee_{d \in dom(g_{i+1,j})} [d](s'_k) \qquad \text{by construction of } s'_k, \text{ i.e., } t'_k \in dom(g_{i+1,j}) \tag{19}$$

Thus for s'_k we have (18) and (19) $\Rightarrow$

$$(\bigvee_{y \mid g_{i+1,j}(y)=t} e(y)(s'_k) \wedge \neg \bigvee_{n\in N} e(n)(s'_k))$$
$$\vee$$
$$(\bigvee_{n\in N} e(n)(s'_k) \wedge \bigvee_{d \in dom(g_{i+1,j})} [d](s'_k)) \qquad \text{since } (A \wedge \neg B) \vee B) \wedge C \Rightarrow (A \wedge \neg B) \vee (B \wedge C)$$

i.e.

$$u_c^k \models (\bigvee_{y|g_{i+1,j}(y)=t} e(y) \wedge \neg \bigvee_{n\in N} e(n)) \vee (\bigvee_{n\in N} e(n) \wedge \bigvee_{d\in dom(g_{i+1,j})} [d]) \quad (20)$$

Thus $u_c \models \bigvee_{n\in N} [n]$ U (20) □

Proof of Lemma 7.3 Let u' to be an infinite execution of M_n since M_n is deadlock free then M_n **sat** $CA(g_{i+1,n}^{-1}(C_i) \cup \ldots \cup C_n\ ,\ g_{i+1,n}^{-1}(O_i))$ thus $u' \upharpoonright dom(g_{i+1,n})$ is infinite since M_n has no anticipated events. From Lemma A.3 u is a path of M_i such that $u\ R\ u'$.
Since $M_i \models \phi$ then $u \models \phi$. We also have that $\texttt{S\_NDF}_{i,n}(t)$ holds, thus from Lemma A.4 (with $i = i$ and $j = n$) and Lemma A.1 we have that $u' \models trans_{g_{i+1,n}}(\phi)$ thus $M_n \models trans_{g_{i+1,n}}(\phi)$ as required. □

Proof of Lemma 7.4 This proof is identical to the proof of Lemma 7.3 but with the penultimate step applying Lemma A.2 since $\mathcal{J}$ is functional.

References

[ABH+10] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *STTT*, 12(6):447–466, 2010.

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

[AFK88] Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. *Distributed Computing*, 2(4):226–241, 1988.

[AM98] Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in B. In *B*, volume 1393 of *LNCS*, pages 83–128. Springer, 1998.

[BAA+08] Juan Bicarregui, Alvaro Arenas, Benjamin Aziz, Philippe Massonet, and Christophe Ponsard. Towards modelling obligations in Event-B. In *Abstract State Machines, B and Z*, volume 5238 of *LNCS*, pages 181–194. Springer, 2008.

[BDJK00] Françoise Bellegarde, Christophe Darlot, Jacques Julliand, and Olga Kouchnarenko. Reformulate dynamic properties during B refinement and forget variants and loop invariants. In Jonathan P. Bowen, Steve Dunne, Andy Galloway, and Steve King, editors, *ZB 2000: Formal Specification and Development in Z and B, First International Conference of B and Z Users, York, UK, August 29 - September 2, 2000, Proceedings*, volume 1878 of *Lecture Notes in Computer Science*, pages 230–249. Springer, 2000.

[BDJK01] Françoise Bellegarde, Christophe Darlot, Jacques Julliand, and Olga Kouchnarenko. Reformulation: A way to combine dynamic properties and B refinement. In José Nuno Oliveira and Pamela Zave, editors, *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*, volume 2021 of *Lecture Notes in Computer Science*, pages 2–19. Springer, 2001.

[But92] M. J. Butler. *A CSP approach to Action Systems*. DPhil thesis, Oxford U., 1992.

[Cle14] ClearSy. Atelier B version 4.2, 2014. http://www.atelierb.eu/en/download-atelier-b/ [accessed 21 April 2016].

[CM89] M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1989.

[DJK03] Christophe Darlot, Jacques Julliand, and Olga Kouchnarenko. Refinement preserves PLTL properties. In *ZB 2003: Formal Specification and Development in Z and B, Third International Conference of B and Z Users, Turku, Finland, June 4-6, 2003, Proceedings*, pages 408–420, 2003.

[DS12] John Derrick and Graeme Smith. Temporal-logic property preservation under Z refinement. *Formal Asp. Comput.*, 24(3):393–416, 2012.

[Gro06] Julien 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.

[HA11] Thai Son Hoang and Jean-Raymond Abrial. Reasoning about liveness properties in Event-B. In *ICFEM*, volume 6991 of *LNCS*, pages 456–471. Springer, 2011.

[HH13] Simon Hudon and Thai Son Hoang. Systems design guided by progress concerns. In *Integrated Formal Methods - 10th International Conference on Integrated Formal Methods (IFM2013)*, volume 7940 of *Lecture Notes in Computer Science*, pages 16–30. Springer-Verlag, 2013.

[HLP13] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Validation of formal models by refinement animation. *Science of Computer Programming*, 78(3):272 – 292, 2013.

[Lam00] Leslie Lamport. Fairness and hyperfairness. *Distributed Computing*, 13(4):239–245, 2000.

[LB08] Michael Leuschel and Michael J. Butler. ProB: an automated analysis toolset for the B method. *STTT*, 10(2):185–203, 2008.

[LFFP09] Michael Leuschel, Jérôme Falampin, Fabian Fritz, and Daniel Plagge. Automated property verification for large scale B models. In *FM*, volume 5850 of *LNCS*, pages 708–723. Springer, 2009.

[Low08] Gavin Lowe. Specification of communicating processes: temporal logic versus refusals-based refinement. *Formal Asp. Comput.*, 20(3):277–294, 2008.

[Mor90] C.C. Morgan. Of wp and CSP. *Beauty is our business: a birthday salute to E. W. Dijkstra*, pages 319–326, 1990.

[Mur13] Toby C. Murray. On the limits of refinement-testing for model-checking CSP. *Formal Asp. Comput.*, 25(2):219–256, 2013.

[PL10] Daniel Plagge and Michael Leuschel. Seven at one stroke: LTL model checking for high-level specifications in B, Z, CSP, and more. *STTT*, 12(1):9–21, 2010.

[PV01] Antti Puhakka and Antti Valmari. Liveness and fairness in process-algebraic verification. In *CONCUR 2001 - Concurrency Theory, 12th International Conference, Aalborg, Denmark, August 20-25, 2001, Proceedings*, pages 202–217, 2001.

[SLDW08] Jun Sun, Yang Liu, Jin Song Dong, and Hai H. Wang. Specifying and verifying event-based fairness enhanced systems. In *Formal Methods and Software Engineering, 10th International Conference on Formal Engineering Methods, ICFEM 2008, Kitakyushu-City, Japan, October 27-31, 2008. Proceedings*, pages 5–24, 2008.

[STW14] Steve Schneider, Helen Treharne, and Heike Wehrheim. The behavioural semantics of Event-B refinement. *Formal Asp. Comput.*, 26(2):251–280, 2014.

[STWW14a] Steve Schneider, Helen Treharne, Heike Wehrheim, and David Williams. Managing LTL properties in Event-B refinement. arXiv:1406:6622, to appear IFM2014, June 2014.

[STWW14b] Steve Schneider, Helen Treharne, Heike Wehrheim, and David M. Williams. Managing LTL properties in event-b refinement. In *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, volume 8739 of *Lecture Notes in Computer Science*, pages 221–237. Springer, 2014.

[WdRF12] David M. Williams, Joeri de Ruiter, and Wan 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.

[Z+14] ETH Zurich et al. Rodin version 3.1.0, 2014. http://www.event-b.org/ [accessed 21 April 2016].