Towards Modelling Obligations in Event-B

Juan Bicarregui*, Alvaro Arenas*, Benjamin Aziz*,
Philippe Massonet† and Christophe Ponsard†

*e-Science Centre, STFC Rutherford Appleton Laboratory, UK
†Centre of Excellence in Information and Communication Tech. (CETIC), Belgium
{j.c.bicarregui, a.e.arenas, b.aziz}@rl.ac.uk, {phm, cp}@cetic.be

Abstract. We propose a syntactic extension of Event-B incorporating a limited notion of obligation described by triggers. The trigger of an event is the dual of the guard: when a guard is not true, an event must not occur, whereas when a trigger is true, the event must occur. The obligation imposed by a trigger is interpreted as a constraint on when the other events are permitted. For example, the simplest trigger *next*, which states that the event must be the next one to be executed when the trigger becomes true, is modelled as an extra guard on each of the other events which prohibits their execution at this time. In this paper we describe the modelling of triggers in Event-B, and analyse refinement and abstract scheduling of triggered events.

1 Introduction

In Event-B, a system is defined as a *state* consisting of a set of *variables* and some *events* that cause the state to change by updating the values of the variables as defined by the *generalised substitution* of the event.

Each event is guarded by some condition, which when satisfied implies that the event is permitted in the current state. However, the guard is not an obligation to perform the event as an event may be delayed as a result of, for example, the interleaving with other permitted events. The choice to schedule permitted events is made non-deterministically.

In this paper, we introduce a dual of guards which we call *triggers*. The trigger of an event expresses an *obligation* on when the event must be executed. This is useful in a number of modelling situations, for example to ensure that if a request for a service is made, then the service will eventually be delivered, or that if a hazard state is encountered an alarm will be promptly raised. Often there is a caveat to the obligation, for example, if the receiver remains ready to receive the service, or if the alarm system is in working order.

Triggers model such obligations as constraints on when other events are permitted. For example, the simplest trigger is *next* which states that the event must be the next one executed when the trigger becomes true. This is in effect an extra guard on each of the other events which prohibits their execution at this time.

Event-B also incorporates a refinement methodology which is used to incrementally develop a model of the system. Our model of triggers enables the

abstract specification of certain constraints on the ordering of events. In refinement, further constraints can be added as these reduce the non-determinism inherent in the choice of events. Thus triggers can be strengthened in refinement.

This work is motivated by a desire to close the gap between requirements and specifications, in particular, when using the KAOS goal-oriented requirements methodology for describing requirements [15] and Event-B for specifying software systems. In KAOS, system goals are refined into requirements under the responsibility of agents. Agents performs operations in order to fulfill requirements. Operations are specified by pre- and post-conditions, which represent state transitions in the usual way, and a trigger condition, which captures an obligation to perform the operation when the condition becomes true provided the domain precondition is true.

The structure of the paper is as follows. Triggers are introduced to Event-B in section 2. The use of triggers is demonstrated and compared with classical Event-B on a motivating exemplar in section 3. The way obligations are interpreted in Event-B is fully described in section 4. Refinement with triggers is discussed in section 5. Then, section 6 discusses abstract schedulability, as triggers introduce constraints on the order of event which may introduce deadlocks for which extra proof obligations are required. Section 7 explores some related work. Finally the paper summarises main results and highlights future work in section 8.

2 Events in Event-B

An Event-B model describes number of *events* which manipulate the state. Events are defined by the following syntax:

$$ev ::= \quad \textbf{EVENT}\ e\ \textbf{WHEN}\ G\ \textbf{THEN}\ S\ \textbf{END}$$

Where G is the guard, expressed as a first-order logical formula in the state variables, and S is the generalised substitution, defined by the syntax of Figure 1.

$S ::=$	$\textbf{SKIP}$	Do nothing
	$\mid\ x := E(var)$	Deterministic substitution
	$\mid\ \textbf{ANY}\ t\ \textbf{WHERE}\ P(t, var)$	
	$\textbf{THEN}\ x := F(t, var)\ \textbf{END}$	Non-deterministic substitution
	$\mid\ S \parallel S'$	Parallel substitution

Fig. 1. The syntax of generalised substitutions.

For a comprehensive description of the Event-B language and its formal meaning, we refer the reader to more detailed references such as [14].

2.1 Events with Triggers

As mentioned above, in standard Event-B systems the next event to be executed is chosen non-deterministically from all those whose guards are true. If a par-

ticular order of execution of events is required this must be described explicitly through the guards by incorporating any required flags or other protocol in the model of the state In this paper, we introduce the ability to implicitly model a particular form of obligation, which we believe gives some of the benefits of richer expressibility without changing the underlying semantic framework of Event-B.

For this purpose, we introduce a new syntactic construct to Event-B which we define within the standard semantics by extending the model. This syntactic sugar provides a way to abstractly describe requirements on the order of execution of events without explicitly detailing a model of how the scheduling is performed. Methodologically, triggers have the advantage of associating with each event any obligation as to when it is performed, but the disadvantage is that they implicitly impose constraints on other events which may be unwelcome.

Note that the obligation imposed by a trigger is similar to a partial correctness guarantee: it ensures that *if* something happens, it will be the right thing but it does not guaranteed that anything will happen at all. That is, it does not guarantee that the system will not deadlock.

The new construct replaces the guard with a trigger and is indicated by changing the **THEN** keyword to **NEXT**. In the simplest case it forces the event to be the next event which happens

$$ev ::= \quad \textbf{EVENT } e \textbf{ WHEN } T \textbf{ NEXT } S \textbf{ END} \ .$$

A weaker form requires the event to happen some time in the future

$$ev ::= \quad \textbf{EVENT } e \textbf{ WHEN } T \textbf{ EVENTUALLY } S \textbf{ END} \ .$$

Both of these are special cases of the **WITHIN** construct which gives an upper bound to the number of other events which may occur before the triggered event

$$ev ::= \quad \textbf{EVENT } e \textbf{ WHEN } T \textbf{ WITHIN } n \textbf{ NEXT } S \textbf{ END}$$

where n is zero for **NEXT** and n is unboundedly non-deterministically chosen for **EVENTUALLY**.

The above syntax introduces a trigger condition, T, into the specification of an event. This condition is a predicate on states which defines those states which, if reached, oblige a particular behaviour to follow. This behaviour can be seen as a bounded form of the `leads-to` modality [12]. Let $\Box$ denote the *always* temporal operator and $\Diamond$ denote the *eventually* operator. Given a certain predicate P defined on the states variables of an evolving system, then $\Box P$ means that P always holds whatever the evolution of the system. The statement $\Diamond P$ means that P holds at system start up or that it will certainly hold after system start up whatever the evolution of the system. Given predicates P and Q, the statement P `leads-to` Q means that it is always the case that once P holds then Q holds eventually, which is formalised as $\Box(P \Rightarrow \Diamond Q)$. Our triggered events are modelling the behaviour $\Box(P \Rightarrow \Diamond_{\leq n}(P \Rightarrow Q))$, meaning that once P holds then $(P \Rightarrow Q)$ will occur before at most n time units. So the **WITHIN** event introduced above models the behaviour $\Box(T \Rightarrow \Diamond_{\leq n}(T \Rightarrow e))$.

Our triggers model a class of queuing behaviour which are common in resource management but also occur in other situations such as in telephone services or ticket controlled supermarket counters. If on entering the queue, the requester is ready to be served, and thereafter remains ready to be served, the service will eventually be delivered. But if the requester leaves the queue, the request is cancelled. In the next section, we introduce a very simple example to motivate the most basic form of such requirement, when the service must be delivered in the next cycle.

3 Motivating Example

We illustrate the use of triggers with a very small example, which although simplified to the point of triviality still illustrates some of the advantages and disadvantages of triggers. The example, taken from [11], is about the sump in a mine which is used to control the drain water out of the main areas. In this system, water seeps into the sump from the mine and the level of water is kept within bounds by operating a pump. Additionally, an alarm must be immediately sounded if methane is detected in the sump. The requirements on the system are as follows:

1. The pump must be activated when the water level reaches a high water sensor in order to keep the mine dry.
2. The pump must be deactivated when the water level reaches a low water sensor in order to avoid the pump running dry which would damage it.
3. If methane is detected in the sump then the pump must be deactivated immediately in order to avoid the risk of explosion, and an alarm sounded in the main areas in order to warn of the eventual risk of flooding.

A partial specification in Event-B is given in Figure 2 in two versions, one using triggers and the other without. Note how the use of the trigger allows the specification of the events to closely reflect the description of the problem given in the requirements in that it captures the immediacy specified in the third requirement alongside other aspects of that requirement. On the other hand, it has the disadvantage of being rather implicit. In order to fully understand the behaviour of the pump, the reader will now have to consider the specification of the methane event.

Note also how the conflict between requirements 1 and 3, in the case where high water and methane are both detected, is handled. The extra guard in the event which switches the pump on ensures that requirement 3 is met. We will show in section 4.1 that the two specifications are equivalent by definition.

In this simple example, both versions can be easily created and understood but as we will see later, the situation is not so simple when there are several more complex timing requirements. In these situations, triggers can be used to raise the level of abstraction by formalising requirements concerning the order of execution of events without explicitly elaborating a model which exhibits them.

```
INVARIANTS
  lowwater : Bool
  highwater : Bool
  methane : Bool
  pump : {ON, OFF}
  bell : {ON, OFF}

EVENTS
high_water_detected
  WHEN highwater = true
  THEN pump := ON
  END

low_water_detected
  WHEN lowwater = true
  THEN pump := OFF
  END

methane_detected
  WHEN methane = true
  NEXT pump := OFF || bell := ON
  END
```

```
INVARIANTS
  lowwater : Bool
  highwater : Bool
  methane : Bool
  pump : {ON, OFF}
  bell : {ON, OFF}

EVENTS
high_water_detected
  WHEN highwater = true
      AND not (methane = true)
  THEN pump := ON
  END
low_water_detected
  WHEN lowwater = true
      AND not (methane = true)
  THEN pump := OFF
  END
methane_detected
  WHEN methane = true
  THEN pump := OFF || bell := ON
  END
```

Fig. 2. A simple example with and without the use of triggers

4 The Interpretation of Triggered Events

4.1 NEXT Events

Let us consider the interpretation of triggers for the simplest case, that is, when a trigger forces an event to be the next one executed. Consider a system with two events e and f, as shown in the upper box of Figure 3.

EVENT e **WHEN** G **THEN** S **END**
EVENT f **WHEN** T **NEXT** R **END**

EVENT e **WHEN** $G \wedge \neg T$ **THEN** S **END**
EVENT f **WHEN** T **THEN** R **END**

Fig. 3. Simple case of **NEXT** trigger and its interpretation

In this case, whenever T become true, then e must be prohibited so that the only remaining possibility is that f is the next event, representing the obligation $\Box(T \Rightarrow \circ f)$, where $\circ$ denote the *next* temporal operator. This can be modelled by extending the guard on e with the negation of T as shown in the lower box of Figure 3. Thus the trigger in f can be considered as a syntactic sugar for an

extra guard on e which ensures that e will be disabled when trigger T is true. It is clear that if G is always false when T is false, that is if $G \Rightarrow T$, then the un-triggered event will never be executed.

The case where there are several triggered events is given in Figure 4. Here all other events must be disabled when any trigger becomes true so if more than one trigger becomes true simultaneously, the machine will be "deadlocked".

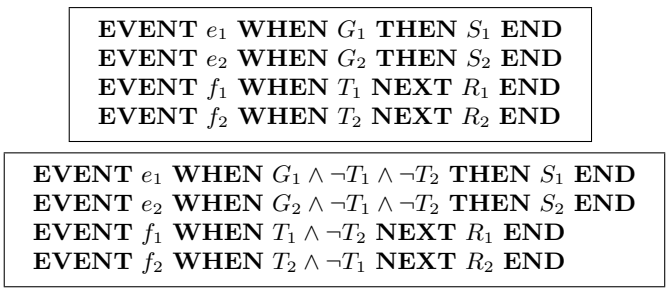

Fig. 4. The interpretation of **NEXT** triggers as extra guards on other events

This will show up in the following deadlock-freeness condition which must be shown alongside the usual one that at least one guard (or trigger) must always be true.

Law 1 (Deadlock-free of NEXT events) *Let M be a system with k next events **EVENT** e_i **WHEN** T_i **NEXT** S_i, for $i = 1 \cdots k$. System M is deadlock-free with relation to its **NEXT** events if all the trigger conditions associated with the **NEXT** constructor are pairwise disjoint, i.e. $\neg(T_i \wedge T_j)$ for $i \neq j$.*

A more general discussion of this law, including the general form of this proof obligation is presented in section 6.

4.2 WITHIN Events

A generalisation of the **NEXT** constructor is the **WITHIN** constructor. In this case, if the trigger becomes true the triggered event must be executed before at most n other events are executed (provided the trigger remains true). If the trigger becomes false within these n steps, the obligation is cancelled.

Again let us consider the simple case of a system with just two events, one of them being a triggered one, as shown in the upper box of Figure 5.

To "un-sugar" this system, as shown in lower box of Figure 5, we must extend the state with a counter for event f. We add an integer valued $counter_f$, which is set with the value n whenever T becomes true, and is decremented each time e is executed whilst T remains true. Here, we borrow the conditional operator from UTP [9], and write $x := exp1(var) \lhd b(var) \rhd exp2(var)$ to denote the substitution **ANY** z **WHERE** $((b(var) \Rightarrow z = exp1(var)) \wedge (\neg b(var) \Rightarrow$

EVENT e **WHEN** G **THEN** S **END**
EVENT f **WHEN** T **WITHIN** n **NEXT** R **END**

$Inv \triangleq \ldots \wedge 0 \leq counter_f \leq n$
$Init \triangleq \ldots \parallel counter_f := n$
EVENT e **WHEN** $G \wedge (\neg T \vee counter_f > 0)$ **THEN**
$S \parallel counter_f := ((counter_f - 1) \triangleleft T \triangleright n)$ **END**
EVENT f **WHEN** T **THEN** $R \parallel counter_f := n$ **END**

Fig. 5. Simple case of **WITHIN** trigger and its interpretation

$z = exp2(var)))$ **THEN** $x := z$ **END** . If $counter_f$ reduces all the way to zero, then e becomes disabled and consequently f becomes obliged. If T becomes false while the counter is active, then it is reset to n. Here we are modelling obligation $\Box(T \Rightarrow \Diamond_{\leq n} f)$, which corresponds to a bounded version of the `leads-to` modality.

The case where there are several triggered events is given in Figure 6. Here the state is extended with a counter for each triggered event and each event is extended with extra clauses in the guards and substitutions to manipulate these counters.

EVENT e_1 **WHEN** G_1 **THEN** S_1 **END**
EVENT e_2 **WHEN** G_2 **THEN** S_2 **END**
EVENT f_1 **WHEN** T_1 **WITHIN** n_1 **NEXT** R_1 **END**
EVENT f_2 **WHEN** T_2 **WITHIN** n_2 **NEXT** R_2 **END**

$Inv \triangleq \ldots \wedge 0 \leq counter_{f_1} \leq n_1 \wedge 0 \leq counter_{f_2} \leq n_2$
$Init \triangleq \ldots \parallel counter_{f_1}, counter_{f_2} := n_1, n_2$
EVENT e_1 **WHEN** $G_1 \wedge (\neg T_1 \vee counter_{f_1} > 0) \wedge (\neg T_2 \vee counter_{f_2} > 0)$ **THEN**
$S_1 \parallel counter_{f_1} := ((counter_{f_1} - 1) \triangleleft T_1 \triangleright n_1)$
$\parallel counter_{f_2} := ((counter_{f_2} - 1) \triangleleft T_2 \triangleright n_2)$ **END**
EVENT $e_2 \ldots$
EVENT f_1 **WHEN** $T_1 \wedge (\neg T_2 \vee counter_{f_2} > 0)$ **THEN**
$R_1 \parallel counter_{f_1} := n_1$
$\parallel counter_{f_2} := ((counter_{f_2} - 1) \triangleleft T_2 \triangleright n_2)$ **END**
EVENT g_1 **WHEN** $T_2 \wedge (\neg T_1 \vee counter_{f_1} > 0)$ **THEN**
$R_2 \parallel counter_{f_2} := n_2$
$\parallel counter_{f_1} := ((counter_{f_1} - 1) \triangleleft T_1 \triangleright n_1)$ **END**

Fig. 6. The interpretation of **WITHIN** triggers introduces a counter for each trigger

It is clear that this model will quickly become quite complex if there are several triggers. In fact the analysis of deadlock for such systems is not trivial as we will see in section 6.

The **NEXT** trigger corresponds to the particular case of **WITHIN** with n equal to 0.

Theorem 1 (Relation between NEXT and WITHIN). *The event* ***EVENT*** e ***WHEN*** T ***NEXT*** R *is equivalent to the event* ***EVENT*** e ***WHEN*** T ***WITHIN*** 0 ***NEXT*** R.

The proof which is omitted relies on the counter being always zero.

4.3 Events with Guards and Triggers

So far, our examples of triggered events have not included guards. We have interpreted this as the guard being the same as the trigger, that is the event is triggered exactly when it is permitted. Another possibility is that the guard of the triggered event is always true. In the most general case, a triggered event can have specified a guard indicating the states in which the event is permitted, as well as a trigger indicating when it is obliged

$ev ::=$ **EVENT** e **WHEN** $(Trigger\ T, Guard\ G)$ **WITHIN** n **NEXT** S **END**

In this case the following healthiness condition, which relates triggers and guards would apply: if an event is obliged, then surely it must be permitted.

Definition 1 (Well formedness of triggers). *For all events* ***EVENT*** e ***WHEN*** $(Trigger\ T, Guard\ G)$ ***WITHIN*** n ***NEXT*** S ***END*** *, we have that* $T \Rightarrow G$.

On the other hand, the classical definition of an event in Event-B corresponds to an event with false trigger.

Theorem 2 (Relating un-triggered and triggered events). *The event* ***EVENT*** e ***WHEN*** G ***THEN*** R *is equivalent to* ***EVENT*** e ***WHEN*** $(false, G)$ ***WITHIN*** n ***NEXT*** R, *where* n *is any arbitrary integer greater than or equal to 0.*

The proof is omitted for brevity.

4.4 EVENTUALLY Events

The unbounded case, described by **WHEN** T **EVENTUALLY** S, is modelled by **WITHIN** with an unbounded non-deterministic choice of n. Note that in this approach, the choice of n is made when the trigger becomes true and so the deadline would be set at that time although it would be only known internally.

5 Refinement with Triggers

Refinement allows one to build a model incrementally by making it more and more precise, that is closer to the reality. In this section we analyse refinement with triggers. We use notation $e \sqsubseteq f$ to indicate that abstract event e is refined by concrete event f, meaning that feasibility, guard and invariant refinement laws hold between e and f, as stated in the Event-B manual [14, pp. 11, Fig. 20].

5.1 Refinement of Duration

It is clear that the addition of triggers to a system restricts its possible behaviours by strengthening its guards and so constitutes a refinement of that system. This is formalised in the theorem below. On the other hand, it may of course introduce the possibility of deadlock which is considered in the next section.

Theorem 3 (Refinement of duration). *Let P be a predicate on states, S be a substitution and let $0 \leq n \leq m$ be integers. Then we have:*

$$
\begin{array}{ll}
 & \textbf{\textit{EVENT}}\ e\ \ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{EVENTUALLY}}\ S \\
\sqsubseteq & \textbf{\textit{EVENT}}\ e_1\ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{WITHIN}}\ m\ \textbf{\textit{NEXT}}\ S \\
\sqsubseteq & \textbf{\textit{EVENT}}\ e_2\ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S \\
\sqsubseteq & \textbf{\textit{EVENT}}\ e_3\ \textbf{\textit{WHEN}}\ P\ \textbf{\textit{NEXT}}\ S
\end{array}
$$

5.2 Refinement of the Trigger Predicate

As mentioned above, guards can be strengthened in refinement and so, by duality, we would expect that triggers can be weakened in refinement[8]. To motivate this, consider the abstract obliged behaviours. These are a minimal set of behaviours necessary for the requirement to be satisfied. During refinement we would expect to ensure that the set of obliged behaviours does not decrease as this could invalidate a requirement.

This can also be understood mechanistically as the trigger is interpreted by adding its negation to the other guards, weakening a trigger is in effect strengthen ing the other guards.

Theorem 4 (Refinement of trigger predicates). *Let M be a system deadlock-free of* ***NEXT*** *events (as defined in Law 1), which includes abstract event* ***EVENT*** e_a ***WHEN*** T_a ***WITHIN*** n ***NEXT*** S*. If system M is deadlock-free of* ***NEXT*** *events when event e_a is replaced by event* ***EVENT*** e_c ***WHEN*** T_c ***WITHIN*** n ***NEXT*** S *and $T_a \Rightarrow T_c$, then we have that*

$$
\begin{array}{ll}
 & \textbf{\textit{EVENT}}\ e_a\ \textbf{\textit{WHEN}}\ T_a\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S \\
\sqsubseteq & \textbf{\textit{EVENT}}\ e_c\ \textbf{\textit{WHEN}}\ T_c\ \textbf{\textit{WITHIN}}\ n\ \textbf{\textit{NEXT}}\ S
\end{array}
$$

Proof. The proof is straightforward by the usual refinement of guards.

5.3 Removing Triggers

From the above we see that we have $T_a \Rightarrow T_c \Rightarrow G_c \Rightarrow G_a$. That is, during refinement, triggers will get ever closer to guards. There are three limiting cases. A false trigger is the degenerate case where nothing is obliged and the specification reverts to a standard Event-B semantics. A true trigger means that the event is always obliged and may therefore block the execution of any other event. The third limiting case is when the trigger becomes equal to the guard. At this point there is no choice left and the permitted behaviours are equal to the obliged ones. Depending on the form of the obligation we have modelled and type of concurrency in the system, this may mean that only one event can execute at any given time and therefore that we have, in effect, partitioned the states by the possible events.

5.4 Implementing Triggers

We have seen how the definition of refinement can be extended to incorporate triggers and how this ensures that obligations are preserved during refinement. However, it is still required, at the end of the refinement process, to ensure that the most concrete specification does indeed implement the triggers and so satisfies the obliged behaviors all the way back up the refinement chain.

Whilst the usual refinement process will ensure the model developed implicitly using triggers is necessarily correct in this sense, it is perhaps unlikely that this model will yield a satisfactory implementation. So we expect to have to build into the implementation a mechanism for scheduling the events which has the desired properties. This concrete model, which itself will have no triggers, is then shown to be correct against the triggered version in the usual way. Thus we do not expect to allow triggers in an implementation but instead develop a model ourselves which implements the required behaviour.

6 Scheduling

The interpretation of triggered events with counters in Event-B is an example of the inclusion of abstract scheduling in a specification, as advocated in [2]. Such techniques have been used in the past for modelling dynamic contraints in B [1] or to specify abstract scheduling of real-time programs [3].

In this section we consider the scheduling of triggered events and develop a sufficient condition for schedulablity. We begin with the case where an event is triggered immediately.

6.1 Deadlock Freeness for NEXT

As stated earlier it is clear that the system will deadlock if two "**WITHIN** 0" triggers become true at any one time. Let us define an active counter to be one whose corresponding trigger is currently true, then it is clear that there must be at most one active counter whose value is equal to zero.

Definition 2 (Active Counter). *For all events,*
EVENT *e* ***WHEN*** *T* ***WITHIN*** *n* ***NEXT*** *S, we say that e has an* **active counter** *if T is true in the current state.*

Definition 3 (Deadlock-free for NEXT). *A system is deadlock-free for* ***NEXT*** *if at all times there is at most one active counter whose value is equal to zero.*

This condition must be true in addition to the usual condition that at least one guard is true to ensure that the system is not currently in deadlock. It is slightly more general than the disjointness of triggers for next events given earlier as it also requires that any "**WITHIN** n" event which may have been triggered earlier does not clash with a "**WITHIN** 0" event just triggered.

6.2 Schedulability

To generalise the above notion of deadlock for triggered events with non-zero counters, we develop some properties related to the schedulability of triggered events.

Definition 4 (Schedulability of a WITHIN event).
Event ***EVENT*** *e* ***WHEN*** *T* ***WITHIN*** *n* ***NEXT*** *S is schedulable if whenever T becomes true, there are at most n other active counters whose value is less than or equal to n.*

This says that whenever a "**WITHIN** n" event is triggered, there are not too many other events already triggered for the next $n + 1$ slots. This is not actually a sufficient condition to guarantee that the system will not deadlock within this period as it is possible that more events with shorter within clauses will be triggered whilst this counter is active. Neither is it a necessary condition, as some triggers which are currently true may become false before their event is executed and therefore liberate some of the slots. It simply states that as far as we can tell at the moment, it is not going to be impossible to schedule this event.

Definition 5 (Schedulability of a system with WITHIN events). *An event system is schedulable if all its* ***WITHIN*** *events are schedulable at all times.*

Schedulability is not easy to prove in general, as it is not at all easy to characterise which counters are active in any given state as this depends on the history of the trace to this point, that is, on which triggers have been true in the past.

Given the above definitions, however, we can give the following characterisation of schedulability.

Theorem 5 (System schedulability). *An event system with* ***WITHIN*** *events is schedulable iff at all times, for all n, there are at most $n + 1$ active counters whose value is less than or equal to n.*

This condition can be considered to be an invariant of a well defined system and can therefore be added as an extra proof obligation for each event which, if true, ensures that the system is deadlock-free. It states that, for any execution of any event, if the system is schedulable beforehand, it must still be schedulable afterwards. This then becomes an inductive prove of deadlock-freeness.

Note that we have assumed that the events meet the healthiness condition given earlier, that is, that the guards are true whenever the triggers are true. This is necessary so that if the scheduling of events requires that an event will be next, we can be sure that it is permitted at this point. The healthiness condition ensures this since, if the guard were false, then so would be the trigger, and so the counter would become inactive and the event removed from the queue.

7 Related Work

There has been several proposals to model obligations in event-based languages like B. In their seminal paper on dynamic constraints in B [1], Abrial and Mussat propose modelling the `leads-to` and the `until` modalities in B. Given P and Q state predicates, the leads-to modality $\Box(P \Rightarrow \Diamond Q)$ means that it is always the case that once P holds then Q holds eventually. They model this modality, for a particular set of events, as a loop which is selected when condition P holds and then iterates, executing one of the events until condition Q becomes true. A variant condition guarantees termination of the loop. By contrast, our triggered events model a bounded leads-to modality $\Box(P \Rightarrow \Diamond_{\leq n} Q)$, which means that once that once P holds then Q will occcur before at most n other events. So, Abrial and Mussat's model can be seen as a general case of our trigger model, but there are some importance differences. For our triggered events, P is an additional predicates on states and Q is the generalised substitution of the event. When P becomes true, a counter is set running which ensures that no more than n *other* events can occur before the triggered event is executed.

In [13], Méry and Merz propose an event language with deontic concepts such as permissions, right and obligations, and develop a stepwise refinement method. Their approach is close to ours in that their notion of obligation corresponds to our trigger condition: a predicate associated to an event indicating the liveness property that when the predicate is true it may lead to the occurrence of the associated event. However, we interpret our triggers through an extension of the usual event-B model rather than introducing a more complex semantic framework.

In [8], Fiadeiro and Maibaum propose a relationship between deontic logic structures, which use the notions of permissions and obligations, and temporal logic through the definition of a consequence operator. This relationship then permits the derivation of normative behaviours of systems, which could include both safety and liveness properties, as well as the reasoning on the relationship between normative states and normative trajectories that could lead to non-normative states, e.g. the performing of permitted actions that lead to obligations that cannot be fulfilled. Our work could be considered as the application of one

aspect of their framework, namely the description of a particular class of deontic property, to Event-B systems.

Other attempts to deal with liveness properties in B include [5], which presents a proposal of specification and proof of liveness properties in Event-B. Here proof obligations are defined in terms of weakest preconditions, inspired by the UNITY logic.

Our work is also related to extensions of Event-B to deal with real-time. In [6], the authors present a refinement method that allows refined events to be guarded by time constraints using the concept of active times. The main difference from our current work is that active times are a form of guards and thus do not express any obliged behaviour. Colin *et al.* describe in [7] the alternative approach of extending the semantic model of B with the duration calculus in order to deal with real-time issues.

More recent work on CSP‖B allows designers to add *control flow annotations* to machine operations [10]. One of their possible annotations is NEXT which introduces the set of operations that should be enabled after an operation is executed. There is an interesting relation between this annotation and our. Interpretation of our **NEXT** event can correspond in some cases to annotating *other* events with NEXT annotations although this correspondence is not straightforward. However, we focus on identifying circumstances when an event will be executed next, rather that defining directly the order in which events must occur.

8 Conclusion

This paper has presented a syntactic extension to Event-B to model the notion of obligation throughout the use of triggers. The obligation imposed by a trigger is interpreted as a constraint on when other events can be permitted. We have analysed issues related to the refinement and schedulability of triggered events.

There are some limitations in our proposal that we plan to address as future work. One restriction is related to the abstract scheduling of events through counters, which could make it difficult to incorporate other scheduling policies into the model. One potential solution could be the use of the VARIANT clause in the model, as advocated in [1]. There are some open questions in relation to eventuality and scheduling, since the selection of n must not lead to deadlock. We also plan to develop a more complete proof method for Event-B with obligations, which will allow one to proof event properties without need to expand events into the classical Event-B.

As mentioned before, our motivation is to link KAOS requirements with Event-B specifications. Triggered events as presented here are suitable for modelling the KAOS *achieve* pattern [16]; we would like to investigate the representation of other modalities as events, so that we can model other KAOS patterns such as *maintain* and *cease*. Finally, we would like to model and reason about obligation policies in our framework. Initial work on this line has been reported in [4].

Acknowledgement

This work is funded by the European Commission under the FP6 IST project GridTrust (project reference number 033817).

References

1. J. R. Abrial and L. Mussat. Introducing Dynamic Constraints in B. In D. Bert, editor, *B98: Recent Advances in the Development and Use of the B Method*, volume 1393 of *Lecture Notes in Computer Science*. Springer-Verlag, 1998.
2. K. R. Apt and E.-R. Olderog. Proof Rules and Transformations Dealing with Fairness. *Science of Computer Programming*, 3(1):65–100, 1983.
3. A.E. Arenas. An Abstract Model for Scheduling Real-Time Programs. In C. George and H. Miao, editors, *Formal Methods and Software Engineering*, volume 2495 of *Lecture Notes in Computer Science*, pages 204–215. Springer, 2002.
4. A.E. Arenas, B. Aziz, J.C. Bicarregui, and B. Matthews. Managing Conflicts of Interests in Virtual Organisations. In *STM 2007, ERCIM Workshop on Security and Trust Management*, volume 197 of *Electronic Notes in Theoretical Computer Science*, pages 45–56. Elsevier, 2008.
5. H. Ruíz Barradas and D. Bert. Specification and Proof of Liveness Properties under Fairness Assumptions in B Event Systems. In *International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.
6. D. Cansell, D. Mery, and J. Rehm. Time Constraint Patterns for Event B Development. In *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2007.
7. S. Colin, G. Mariano, and V. Poirriez. Duration Calculus: A Real-Time Semantic for B. In *Theoretical Aspects of Computing - ICTAC 2004*, volume 3407 of *Lecture Notes in Computer Science*, pages 431–446. Springer, 2005.
8. J. Fiadeiro and T. Maibaum. Temporal Reasoning over Deontic Specifications. *Journal of Logic Computation*, 1(3):357–395, 1991.
9. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice Hall Series in Computer Science, 1998.
10. W. Ifill, S. Schneider, and H. Treharne. Augmenting B with Control Annotations. In *B 2007: Formal Specification and Development in B*, volume 4355 of *Lecture Notes in Computer Science*. Springer, 2007.
11. M. Joseph. *Real-Time Systems: Specification, Verification and Analysis*. Prentice Hall International, 1996.
12. Z. Manna and A. Pnueli. *The Reactive Behavior of Reactive and Concurrent System*. Springer-Verlag, 1992.
13. D. Méry and S. Merz. Event Systems and Access Control. In D. Gollmann and J. Jürjens, editors, *6th Intl. Workshop Issues in the Theory of Security*, pages 40–54, Vienna, Austria, 2006. IFIP WG 1.7, Vienna University of Technology.
14. C. Métayer, J. R. Abrial, and L. Voisin. Event-B Language. Rodin Deliverable D3.2, 2005.
15. A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Guided Tour. *Fifth IEEE International Symposium on Requirements Engineering*, 2001.
16. A. van Lamsweerde and E. Letier. Deriving Operational Software Specifications from System Goals. In *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002*, pages 119–128. ACM, 2002.