Using Indexed and Synchronous Events to Model and Validate Cyber-Physical Systems

Chen-Wei Wang, Jonathan S. Ostroff and Simon Hudon

Technical Report EECS-2014-03

June 30 2014

Department of Electrical Engineering and Computer Science
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

Using Indexed and Synchronous Events to Model and Validate Cyber-Physical Systems

Chen-Wei Wang, Jonathan S. Ostroff, and Simon Hudon *

Department of Electrical Engineering & Computer Science, York University, Canada

Abstract

Timed Transition Models (TTMs) are event-based descriptions for modelling, specifying, and verifying discrete real-time systems. A system is composed of module instances. Each module declares an interface and a list of events. An event can be spontaneous, fair, or timed (i.e., with lower and upper time bounds). TTMs have a textual syntax, an operational semantics, and an automated tool, including an editor with type checking, a graphical simulator, and a verifier for linear-time temporal logic. In this paper, we extend TTMs and its tool with two novel modelling features: synchronous events and indexed events. Event synchronization allows developers to decompose simultaneous state updates into actions of separate events. To specify the intended data flow among synchronized actions, we reference the post-state (i.e., one resulted from taking the synchronized actions) using primed variables. The TTM tool automatically infers the data flow from synchronous events, and reports type errors when there are inconsistencies due to circular data flow. We apply synchronous events to verify function blocks from the IEC 61131 standard, and a part of the requirements of a nuclear shutdown system. Indexed events allow for concise description of behaviour common to a (possibly unspecified) set of actors. The indexing construct allows us to select a specific actor and specify a temporal property for that actor. We use indexed events to verify a mutual exclusion protocol and a train system. In all examples, the TTM tool is used to verify safety, liveness, and real-time properties.

* jackie@cse.yorku.ca; Corresponding author

Contents

1 Introduction

Cyber-physical systems integrate computational systems (the "controller") with physical processes (the "plant"). Such systems are found in areas as diverse as aerospace, automotive, chemical processes, civil infrastructure, energy, healthcare, manufacturing, transportation, and consumer appliances. A main challenge in developing cyber-physical systems is modelling the joint dynamics of computer controllers and the plant [3].

Certification of cyber-physical systems requires making arguments that convince the relevant certification authority that suitable steps have been taken to ensure the safety, reliability and integrity of safety critical systems. Formal methods are recognized as one of the methods used to provide evidence of safety and fitness for purpose. For example, DO-333 [4] provides guidance to software developers wishing to use formal methods in all phases of the design of airborne systems to satisfy DO-178C certification objectives. A recent case study [2] successfully adopts the guidance in DO-333 to satisfy DO-178C objectives, using different formal verification techniques (theorem proving, model checking, and abstract interpretation) on a flight guidance system. It is widely recognized that many problems start with incomplete or missing requirements. Thus the validation of requirements is an important problem.

Timed Transition Models (TTMs) are event-based descriptions for modelling, specifying, and verifying discrete real-time systems. A system is composed of module instances. Each module declares an interface and a list of events. An event can be spontaneous, fair, or timed (i.e., with lower and upper time bounds). In [13], we provided TTMs with a textual syntax, an operational semantics, and an automated tool, including an editor with type checking, a graphical simulator, and a verifier for linear-time temporal logic. In that paper, TTMs were used to verify that a variety of implementations satisfy their specifications.

Contributions. In this paper, we extend the TTM notation and tool with two novel modelling features: indexed events and synchronous events. These constructs provide expressive descriptions closer to the specification level, and can thus facilitate the validation of system requirements.

Indexed events allow for concise description of behaviour common to a (possibly unspecified) set of actors. The indexing construct allows us to select a specific actor (such as a train) and specify a temporal property for that actor. For example, let *loc* be an array of train locations (a train can be on either the entrance block, a platform, an exit block, or outside the station). An event *move_out* can be indexed with a set *TRAIN* of trains, which results in an indexed event *move_out(t: fair TRAIN)* describing the action of a train t moving out of a platform and into the exit block. As a result, the event index t can be used to specify the liveness property that every train t waiting at a platform eventually moves out, and into the exit block: $\Box(loc[t] \in PLF \Rightarrow \Diamond move\_out(t))$. A

stronger property states that each train waiting at a platform eventually departs from the exit block to leave the station: $\Box(loc[t] \in PLF \Rightarrow \Diamond(loc[t] = Out))$. Without the index t, we can only state a weaker property that some train eventually leaves the station (unless we introduce auxiliary variables or events). We can also assert the safety property that there are no collisions, i.e. no two trains are on the same block at the same time.

TTM notations are rich enough to provide a variety of models at different levels of abstraction. In Section 3.3, we provide two models for a train system. In both models, the TTM tool is used to verify safety, liveness, and real-time properties. The first model is an abstract description of the system, combining events of the plant and the controller and using the full power of indexed events. We then provide a refinement, where we remove the index t from the controller event, and replace the strong fairness assumption that was made by a round-robin queue (making the controller closer to implementation). The TTM tool allows the queue to be implemented in C# which makes the model description clearer while reducing state space.

Synchronous events are two (or more) events in different TTM modules that act together while simultaneously updating their local, output, and shared variables (subject to rules of consistent updates). In addition, we allow primed and unprimed variables in event updates, so that the the next-state value of an output variable o can be expressed using both the current (or pre-state) value of an input variable i and its next-state value (i'). This makes the update to output o instantaneous (i.e. the output changes at the same time as the input, rather than after). The TTM tool performs type-checking to ensure that there is no circularity in the data flow.

Synchronous events, together with primed variables, are suitable for describing the kind of high-level specifications used in shutdown systems of nuclear reactors [18]. In such systems, the next-state value of the system controlled variables are expressed in terms of the current-state and next-state values of the monitored variables of nuclear reactors. This allows for a simplified description of the requirements that will later be refined to code in the design phase.

As the first case study of synchronous events, we consider function blocks drawn from the IEC 61131-3 Standard. Programmable logic controllers (PLCs) are widely used for developing embedded systems. Function blocks are used to describe the components of PLCs. In the IEC 61131 standard [8], function blocks are described by *structured text* (a Pascal like programming language) and *function block diagrams*. In Section 4.2, we use synchronous events to show that systems composed of function blocks satisfy their intended specification.

Figure 6 (p.17) shows the declarations and implementations of function blocks we consider, as well as the three-step verification process we take (Figure 6a). First, we show that a hysteresis function block described by structured text (the implementation) satisfies an input-output relation (its specification). When hysteresis blocks are used in

compositions, we may now use their specifications instead of their implementation. This simplifies the verification of compositions. Second, we implement a *limits alarm* block using two hysteresis blocks (one block for a low alarm and one block for the high alarm). In the composed limits alarm block, the various events in the composition synchronize with each other and are thus taken simultaneously, as required by the semantics of function block composition. We then verify that the implementation of the limits alarm block satisfies its specification. Third, we show that a system composed of the limit alarm block and an unconstrained environment satisfies an important safety property, viz. that the high and low alarm will never trip at the same time.

As a second case study, we use synchronous events to describe the Neutron OverPower (NOP) controller in a nuclear reactor [18]. The main requirement (which we will call a response property) is that the NOP controller must trip the nuclear reactor if any one of the many monitored variables exceeds certain set point (high limit) with a hysteresis region. The software requirements of the NOP is specified using tabular expressions (also called function tables [9]). The system is treated as a black box taking input stimuli (monitored variables) and producing output responses (controlled variables). Hidden from the black box are intermediate function variables. The entire shutdown system, which the NOP is part of, is composed of a large number of function tables. Each table documents a controlled variable, or an intermediate function variable, as a mathematical function of its dependent monitored and function variables. In their model of requirements, environment changes (on monitored variables) and controller responses (on controlled variables) occur simultaneously as a single state transition. That is, in the function table for a controlled variable, the computation for its next-state value may depend on not only the pre-state, but also the post-state values of all its dependant monitored and function variables. Such idealized behaviour is only implementable if the response allowance [19] is also specified for every pair of controller and monitored variables. Our synchronous events, together with the primed notation, can thus be used to describe the various function tables in the NOP. However, function tables that specify individual components in the NOP cannot be used to validate the overall response property.

In Section 4.3, we present two TTMs of the NOP. The first model presents a high-level requirements model where the controller responds instantaneously to changes in the environment (the plant). We synchronize both the environment and controller events to model such instantaneity. In this abstract model, we use the TTM tool to check the critical response property as an invariant. In fact, for the abstract specification to be implementable, it must be specified with a response allowance (RA) on the controller's response to input stimuli [19].

However, to keep the example simple for illustration (and without loss of generality), we assume that the plant changes more slowly than the controller responds. Given this assumption on response allowance, we provide a second model, a refinement of the first, in which the plant and the controller are decoupled. That is, the plant and controller events

are interleaved. The assumption on response allowance is specified using time bounds of the plant and controller events. In this model, using a timer t, the NOP response property becomes: $\Box($ $(any\_monitored\_signal\_unsafe \wedge t = T) \Rightarrow \Diamond(tripped \wedge t < T + 2)$ $)$.

Outline. In Section 2 we summarize the syntax of TTMs. In Section 3 we illustrate indexed events using a mutual exclusion protocol and a train station. In Section 4 we illustrate synchronous events (and primed variables) using function blocks drawn from the IEC 61131-3 Standard on Programmable Logic Controllers and the requirements of a nuclear shutdown system. In Section 5 we discuss the related works.

Resources. The operational semantics for indexed and synched events are included as an appendix to this report . Complete TTM listings of all our case studies are available at: `https://wiki.eecs.yorku.ca/project/ttm/index_sync_evt`.

2 Preliminaries

We illustrate the basic features of textual TTM using the following small example[1].

```
#define N 10;
type
 ST = {on , off}
 PID = 1 .. (N−1)
end
share initialization
 a : share ARRAY[ST](N) = [on (N)]
end
timers t : 0 .. 2 disabledinit end
```

```
module M
interface
  p : in PID
  s : share ST
local
  b : BOOL = false
events
 e [1, 3] just
  when ... start t do ... end
end
```

```
instances
 m0 = M(in 0, share a[0])
end
composition
 ms = || p : PID @ M(in p, share a[p])
 system = m0 || ms
end
#assert system |= []<>tick;
#assert system |= []<>"ms[2].e";
```

We use *define* clauses to declare global constants (e.g., N) and macro functions (e.g., ***#define*** *min(x, y) (* ***if*** $x \leq y$ ***then*** x ***else*** y ***fi*** *)*). An explicit *call* statement is required for using a macro function (e.g., ***call****(min, 4, 44)*). Within a *type* ... *end* clause, we define a list of named enumerations (e.g., *BUTTON*) and intervals (e.g., *PID*). Within a *share initialization* ... *end* clause, we declare a list of global variables and their initial values shared by all module instances (e.g., the Boolean array a of constant size N is initialized to *false* for all cells). We may also declare a shared object typed to some imported C# class (e.g., *qe* : <*Queue*>). Within a *timers* ... *end* clause, we declare a list of global timers with a specified range (e.g., t). Each timer is disabled initially, meaning that it starts counting only after some event has explicitly started it; otherwise, the timer should be declared as *enabledinit*.

Each *module* ... *end* clause declares a module that consists of interface and local variables, and a list of events. We support primitive types including integer *INT*, Boolean

[1]For more examples, Figure 12 (p. 27) shows (part of) requirements of a nuclear shutdown system, which we will discuss in details in Section 4.3. A pacemaker in TTM is also available [13]. More examples and the grammar can be found in: `https://wiki.eecs.yorku.ca/project/ttm/`. We give an informal account on the behaviour of events. A formal, complete explanation of the semantics is reported [13].

BOOL, user-defined enumerations (e.g,. *ST*) and intervals (e.g., *PID*). Each module variable may be typed to a primitive or an array of primitives (e.g., $ARRAY[ST](N)$). Each interface variable is specified with a modifier that indicates the intended usage in its events: an *in* variable is real-only; a *shared* variable may be read and written by any module declaring such usage; and an *out* variable means that all other modules may only read it.

Each module event may be specified with a guarding predicate (via the *when* clause) and an action (via the *do* clause) using parallel assignments and nested if-statements. Assignments may be performed deterministically (e.g., *s := off*) or demonically (e.g., *s :: ST*). Each *demonic assignment* non-deterministically assigns a value from the specified type (which can be primitives or arrays). Demonic assignments are useful for modelling the environment (or plant) module that non-deterministically changes values of variables that are monitored by the controller module. Each event may also start or stop a list of global timers (using the *start* or *stop* clause).

Each module event e may be specified with a real-time constraint (i.e., $[l,\ u]$ meaning lower time bound l and upper time bound u) and with a fairness assumption [17] (i.e., *just* or *compassionate*). Informally speaking, as soon as its guard G_e is satisfied, an implicit timer T_e starts and keeps counting as long as G_e remains satisfied. Event T_e is *enabled* and must be taken when $l \leq T_e \leq u$. On the other hand, the strong *compassionate* (resp. weak *just*) fairness constraint introduces the assumption that if event e is enabled infinitely often (resp. is eventually continuously enabled), then it occurs infinitely often.

To construct a system for verification, we instantiate modules (via an *instances* ... *end* clause) and compose instances (via a *composition* ... *end* clause). In creating instances, we supply value expressions for *in* variables (e.g., 0 for variable p in module M) and rename *out* variables (e.g., $a[0]$ for variable s in module M, specifying that instance *m0* shares only a portion of the global array a). In composing instances, we support both binary parallel compositions (e.g., *system*) and indexed parallel compositions (e.g., *ms*). In each composition, events are qualified by names of their containing module instances (e.g., *m0.e* and $ms[2].e$). For simulation and verification, the TTM tool constructs a reachability graph by flattening the module structure: all qualified events are treated as possible state transitions.

In checking assertions on compositions, we support operators in the langue linear temporal logic (LTL) [11]: henceforth ($\Box$), eventually ($\Diamond$), next ($\bigcirc$), strong until ($\mathbf{U}$), and release ($\mathbf{R}$). However, the tool does not support the weak until ($\mathbf{W}$) operator, but we often find it useful for stating properties. As a result, we reformulate properties using $\mathbf{W}$ to use $\mathbf{R}$ instead. In assertions, we may refer to occurrences of events, including the clock tick to check that the system exhibits no Zeno-behaviour (i.e., $\Box\Diamond$ *tick*).

3 Indexed Events

3.1 Syntax and Informal Semantics

We introduce the syntax of indexed events using two versions of an event for trains to move out of the station:

move_out(t : *TRAIN*) **just**	*move_out*(t : **fair** *TRAIN*) **just**
when ...	**when** ...
do ...	**do** ...
end	**end**

Both versions parameterize the event with an index t ranging over a set of trains, and the guards and actions may reference t. Each version denotes a family of events, with each member event instantiating each of t with a particular train value. In the version where the index t is declared as *fair*, each individual member event is chosen by the scheduler (with the declared fairness assumption on the family, i.e., *just*). This makes it possible for us to assert the temporal property that a particular train can eventually move out of the station upon its entrance. On the other hand, in the version where the index t is not declared as *fair* (which we call a *demonic* index), the scheduler non-deterministically chooses a member event from the family (with the declared weak fairness assumption). This makes it only possible for us to assert a weaker property that some train can eventually move out of the station. In general, we may have a set of *fair* indices and a set of *demonic* indices declared for an event. Each combination of values for the *fair* indices will be chosen by the scheduler with the declared fairness assumption on the event family. On the other hand, each combination of the *demonic* indices will be chosen non-deterministically with no fairness assumptions.

3.2 Example: A Locking Protocol

We use indexed events to model, specify, and verify a locking protocol. There are two justifications for using the indexed events. First, all processes regulated by the locking protocol share a common behaviour of entering and leaving the critical section (CS). Second, by declaring that the indexed processes being fair, we can assert that once a process makes its request, that particular process (rather than an arbitrary process) is guaranteed to eventually enter its CS. We also assert, using a global timer and an observer, that once a process makes its request, its waiting time cannot be indefinite. Furthermore, to model the first-come-first-serve mechanism for processes to enter their CSs, we illustrate the use of a C# (FIFO) *Queue* object. Implementation details of queue operations such as *Enqueue*, *Dequeue*, and *First* are all encapsulated, resulting in a cleaner model description compared with using a native TTM array.

Given a finite number N of processes, we derive a set of process identifiers: $PID = 0\,..\,(N-1)$. Figure 1 illustrates the common behaviour (i.e., state transitions) of process $p \in PID$.

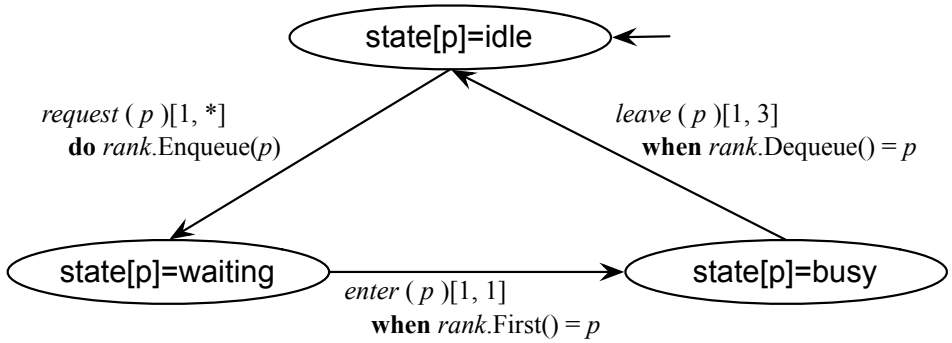


Figure 1: Locking Protocol: State Transitions of Process $p \in PID$

Each process has three possible states: (1) *idle* for a new request to enter its CS; (2) *waiting* for its turn to enter its CS; and (3) *busy* with performing tasks in its CS. Process that have requested to enter their CS do so on a first-come-first-serve basis. To achieve this in TTM, we declare a shared variable *rank* of type *Queue*, a C# class from the standard PAT library. Once it is process p's turn to enter its CS (i.e., $rank.First() = p$), it takes exactly one clock tick to complete the entrance without any indefinite delay. Similarly, once entering, the time of process p staying in its CS is also bounded (i.e., time bounds $[1, 3]$ for event *leave*).

We describe the common process behaviour (Figure 1) using indexed events (in module *LOCK*):

request(p : **fair** *PID*) [1, $*$]	*enter*(p : **fair** *PID*) [1, 1] **just**	*leave*(p : **fair** *PID*) [1, 3] **just**
when	**when**	**when**
$state[p] == idle$	$state[p] == waiting$ &&	$state[p] == busy$
do	$rank.First() == p$	**do**
$state[p] := waiting$,	**do**	$state[p] := idle$,
$rank.Enqueue(p)$	$state[p] := busy$	$rank.Dequeue()$
end	**end**	**end**

We declare index p of all three events as *fair*. This allows us to assert that a particular process p that makes its requests will eventually enter its CS. For example, for process 0, we assert that:

$$\Box\ (\ state[0] = waiting \Rightarrow \Diamond(state[0] = busy)\ ) \tag{1}$$

$$\Box\ (\ l.request(p{=}0) \Rightarrow \Diamond l.enter(p{=}0)\ ) \tag{2}$$

where l is an instance of module *LOCK*, and *request(p=0)* and *enter(p=0)* identify the occurrences of events *request* and *enter* specific for process 0. Otherwise, if these indices are not fair, then we can only assert a weaker property: when a process makes a request, then eventually another (not necessarily the same) process enters its CS.

In fact, since we impose a real-time constraint on the *enter* event of each process (i.e., it takes exactly one clock tick to complete), we can a stronger property than Equations 1 and 2. For example, by introducing a global timer t, we declare an *OBSERVER* module that monitors the state of process 0 using two instantaneous events (i.e., with zero lower and upper time bounds):

p0_starts_waiting[0, 0]	*p0_gets_busy*[0, 0]
when *state*[0]==*waiting* && !*t_on*	**when** *state*[0]==*busy* && *t_on*
start *t* **do** *t_on* := **true end**	**stop** *t* **do** *t_on* := **false end**

Then by creating an instance o of module *OBSERVER*, we assert that

$$\Box \left(\begin{array}{l} o.p0\_starts\_waiting \Rightarrow \\ \quad state[0] = waiting \wedge mono(t) \ \ \mathbf{U} \ \ o.p0\_gets\_busy \wedge (t \leq 4 \times (N-1) + 1) \end{array} \right) \quad (3)$$

Once making its request, the maximum amount of time for process p to wait is when it is at the end of the queue *rank*, in which case each of the $N-1$ process takes up to 4 clock ticks to enter and leave its CS, and when it is p's turn for entrance, it takes another 1 clock tick to do so.

Finally, to ensure that the locking protocol guarantees mutual exclusion, we may assert that while a process is still in its CS, the turn is not given to any other processes waiting.

$$\Box \ (\ \forall p : PID \ \bullet \ (state[p] = busy \Rightarrow rank.First() = p) \ ) \quad (4)$$

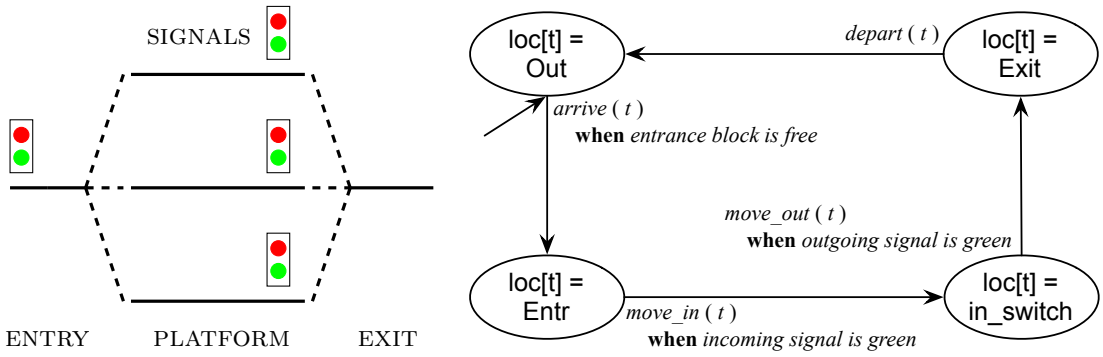
Another property that more explicitly asserts mutual exclusion is that no two distinct processes are at their CSs (i.e., in their *busy* states) simultaneously.

$$\Box \ (\ \forall p, \ q : PID \ \bullet \ (p \neq q \Rightarrow \neg \ (state[p] = busy \wedge state[q] = busy)) \ ) \quad (5)$$

3.3 Example: A Train Control System

We illustrate the use of TTM indexed events in a train control system. There are two reasons for using the indexed events. First, all trains entering and leaving the station share a common behaviour. Second, by declaring the events' indices (ranging over trains) as fair, we can assert that individual trains arriving at the station are guaranteed to depart, without being blocked indefinitely by other trains. We also assert that the system is safe: trains never collide. While satisfying these liveness and safety properties, we provide two versions for the train control system. The abstract version schedules trains (for leaving the station) non-deterministically. The refined, more realistic version

resolves the non-determinism by using a FIFO queue, implemented in C#, together with a weaker fairness assumption. By using a C# *Queue* object, implementation details of operations such as *Enqueue*, *Dequeue*, and *First* are all encapsulated, resulting in a model simpler than one using a native TTM array. Moreover, the refined version models the sensors and actuators in a more realistic manner, and the accuracy is stated using a gluing invariant.



(a) Topology

(b) State Transitions of Train $t \in TRAIN$

Figure 2: A Train Control System

Figure 2a shows the topology of the train control system [6, 7]. There is an entry block (*Entr*) and an exit block (*Exit*) on the two ends of the station. Between the entry and exit blocks is a set *BLOCK* of special blocks called platforms. At most one train may stay at the entry or exit block at a time. On the entry bock, there is a signal *isgn* regulating the incoming train, as a function of the availability of platforms. On each platform $p \in BLOCK$, there is a signal $osgn[p]$ regulating the outgoing train, as a function of the availability of the exit block. Figure 2b illustrate the common behaviour of all trains. Each train is initially travelling outside the station. The train may first arrive at the entry block of the station, provided that it is not occupied. When the signal *isgn* turns green, the train is directed via an in-switch to move in an available platform. For some train t, after it moved to platform p, it waits for the light signal of platform p to turn green and then moves away from p and onto the exit block. Then the train may depart from thestation.

The train system must satisfy the safety property that trains never collide (i.e., no two trains are ever at the same location). We ensure that once a train arrives, the scheduling mechanism for outgoing signals allows that particular train to eventually depart from

the station.

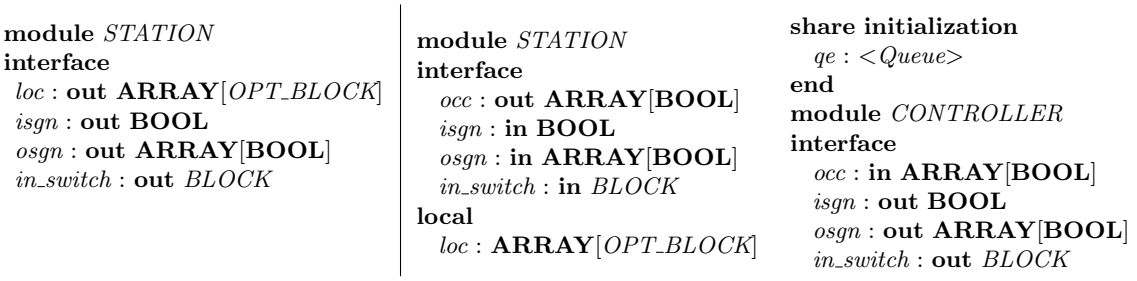
$$(\forall t1, t2 : TRAIN \bullet (\ t1 \neq t2 \wedge loc[t1] \neq Out \wedge loc[t2] \neq Out\ ) \Rightarrow\ loc[t1] \neq loc[t2]) \quad (6)$$

$$\Box(\ loc[t] = Entr \Rightarrow \Diamond(loc[t] = Out)\ ) \quad (7)$$

where the array variable *loc* maps a train to its location (outside or at the entry, exit, or a platform). We present two versions of TTM that satisfy both Equations 6 and 7.

Figure 3a presents the interface of an abstract version of TTM. At this level of abstraction, there is no separation between monitored and controlled variables. As a result, the abstract version contains a single *STATION* module that: (a) owns all variables; and (b) mixes all events of train movement (e.g., event *move_out* in Figure 4a) and of signal control (e.g., event *ctrl_platform_signal* in Figure 5a). On the other hand, Figure 3b presents the TTM interface of a refined version of TTM. This refined version distinguishes between one monitored variable (i.e., the set *occ* of occupied platforms) and three controlled variables (i.e, the signal *isgn* for an incoming train, the platform *in_switch* currently connected to the entrance block, and signals *osgn* for outgoing trains).

Consistently, the behavior of the controller and that of the trains are factored in separate events and placed in separate modules, respectively *CONTROLLER* and *STATION*. The monitored variable is owned by the *STATION* module (the environment), and is read-only for the *CONTROLLER* module, as indicated by the modifier ***in*** its interface. Similarly, the controlled variables are exclusively written by the *CONTROLLER* module, and are read-only for the *STATION* module.



(a) Abstract (b) Refined: Separate Station & Controller Events

Figure 3: Train Control System in TTM: Interfaces

The refinement of the abstract train control system changes the representation of the data used by control events. In the abstract version (Figure 3a), the array variable *loc* is used to map each train to its current location, constrained by type $OPT\_BLOCK \triangleq \{Out\} \cup BLOCK$ where $BLOCK \triangleq \{Entr,\ Exit\} \cup PLF$. All train events (e.g., *move_out* in Figure 4a) are indexed with the set of trains and update their location accordingly (e.g., $loc[t] :=$

Exit). All control events (e.g., *ctrl_platform_signal* in Figure 5a) query the value of *loc* in their guards (e.g., we write !*(||t: TRAIN @ loc*[*t*] == *Exit)* to check that the exit block is not occupied). However, a more realistic station controller may monitor platforms in the station only, rather than all trains including those travelling elsewhere outside the station. Consequently, in the refined version (Figure 3b), by refactoring *loc* as a local variable in the *STATION* module (the environment), we hide it from the *CONTROLLER*. The controller then only has access to the monitored variable *occ* (i.e., the set of occupied platforms) which encodes a coarser grain of information than *loc* (i.e., locations of all trains). Using the new monitored variable *occ* simplifies guards of controller events (e.g., in Figure 5b, we write !*occ*[*Exit*] instead of an existential quantification to express that the exit block is free). In addition, train events in the environment (e.g., Figure 4b) updates both the local variable *loc* and the output variable *occ*. This raises the question of whether the *CONTROLLER* module accesses the monitored variable *occ* in a way consistent with the corresponding events in the abstract model. Therefore, we assert the following invariant (where *s* is a *STATION* instance): a block is occupied if and only if it corresponds to the location of some train.

$$\Box\ (\forall b : BLOCK \bullet\ occ[b]\ \equiv\ (\exists t : TRAIN \bullet\ s.loc[t] = b)) \tag{8}$$

```
move_out(t : fair TRAIN) just
 when call(is_platform,loc[t]) && osgn[loc[t]]
 do loc[t] := Exit, osgn[loc[t]] := false end
```

(a) Abstract Version

```
move_out(t : fair TRAIN)[2, *] just
 when call(is_platform,loc[t]) && osgn[loc[t]]
 do loc[t] := Exit, occ[loc[t]] := false, occ[Exit] := true
      end
```

(b) Refined Version

Figure 4: Train Control System in TTM: the *move_in* Event in Module *STATION*

```
ctrl_platform_signal(p : fair BLOCK) compassionate
 when call(is_platform, p)
   && (&&p : BLOCK @ call(is_platform, p) -> !osgn[p
       ])
   && !(||t: TRAIN @ loc[t] == Exit)
   && (||t: TRAIN @ loc[t] == p)
 do osgn[p] := true end
```

(a) Abstract Version in module *STATION*

```
ctrl_platform_signal just
 when qe.Count() != 0
   && !osgn[qe.First()]
   && !occ[Exit]
   && occ[qe.First()]
 do osgn[qe.First()] := true end
```

(b) Refined Version in module *CONTROLLER*

Figure 5: Train Control System in TTM: Controller Events

The fundamental difference between the abstract and the concrete TTM models resides in the scheduling of the green signals that control the passage from the platforms to the exit block. While the abstract model specifies very little with respect to the order in which trains gain access to the exit block (i.e., the order is non-deterministic), the concrete model specifies the order uniquely. The signals are controlled by event *ctrl_platform_signal*. In the abstract version (Figure 5a), the event is indexed by the set of trains. When the exit block is not occupied, more than one train located at a platforms may be eligible to move on to the exit block. To satisfy Property 7, we declare the index on trains as fair and adopt a strong fairness assumption (i.e., *compassionate*) on the controller event. That is, a train infinitely often qualified to leave the station does so eventually. However, such fairness assumption cannot be implemented efficiently in a general manner. This is why, in the refined version (Figure 5b), we use a C# FIFO *Queue* to dictate the order of departure of the trains: the first train to reach a platform is also the first one to leave. The reduced non-determinism allows us to remove the fair index on trains and weaken the fairness assumption (i.e., the event becomes *just*).

4 Synchronous Events

4.1 Syntax and Informal Semantics

We introduce the syntax of synchronous events in TTM using the following example.

```
module PLANT
  interface
    x : out INT = 0
  events
    generate
    do
      x :: 0 .. 10
    end
end
```

```
module CONTROLLER
  depends p : PLANT
  interface
    x : in INT
    b : out BOOL = false
  events
    respond sync p.generate as act
    do
      if x' > 0 then b := true else b = false fi
    end
end
```

```
instances
  env = PLANT(out x)
  c = CONTROLLER(in x, out b)
        with
          p := env
        end
  sync_env_c ::= env || c
end
composition
  system = sync_env_c
end
```

At the module level (e.g., *CONTROLLER*), we use a *depends* clause to specify a list of instances that the current module depends on. At the event level (e.g., *respond*), we use a *sync* ... *as* ... clause to specify the list of events to be synchronized, qualified by names of the dependent instances (e.g., *p.generate*), and to rename the synchronized events with a new name (*act*). Actions of events that are involved in synchronization may reference the primed version of input variables to obtain their post-state values. For example, the *respond* event uses the post-state value of the input variable x (i.e., x') to compute the

post-state value of its output variable b. In creating an instance, we use a *with* ... *end* clause to bind all its dependent instances, if any. We use the $::=$ operator to rename the synchronized instances (e.g., *sync_env_c*). As the instances *env* and *c* are synchronized as the new instance *sync_env_c*, taking the event *sync_env_c.act* has the effect of updating, as one atomic step, the monitored variable x then the controlled variable b. Consistency rules are enforced to ensure that dependencies at the module, event, and action levels are acyclic.

4.2 Example: Two Function Blocks from the IEC 61131 Standard

In this section, we use synchronous events to show that systems composed of function blocks from the IEC 61131 Standard [8] satisfy their intended specification. Programmable logic controllers (PLCs) are widely used for developing embedded systems. Function blocks (FBs) are used to describe the components of PLCs. Part 3 of the IEC 61131 Standard [8] was published by the International Electrotechnical Committee (IEC) to standardize the syntax of FBs for building PLCs, and to supply a library (Annex F) of example FBs. For verification, we choose the *HYSTERESIS* and *LIMITS_ALARM* blocks, for which the standard supplies, respectively, a structured text (ST) implementation and a function block diagram (FBD) implementation.

Figure 6a (p.17), from right to left, summarizes our three-step verification process for the two chosen FBs. The construct of synchronous events is used in the second and third steps. First, we verify that the ST implementation of the *HYSTERESIS* block supplied by IEC 61131-3 (Figure 6c) exhibits the expected behaviour (Figure 6f) by satisfying an input-output relation (its specification). When using hysteresis blocks in compositions, we may use their specifications instead of their implementation. This simplifies the verification of compositions. Second, using the specification of *HYSTERESIS* (Figure 6f), we verify that the FBD implementation of the *LIMITS_ALARM* block supplied by IEC 61131-3 (Figure 6e) exhibits the expected behaviour (Figure 6g). In the composed limits alarm block, the various events in the composition synchronize with each other and are thus taken simultaneously, as required by the semantics of function block composition. Third, we show that a system composed of the *LIMITS_ALARM* block and an unconstrained environment satisfies an important safety property, viz. that the high and low alarm will never trip at the same time.

Figure 6b shows the input-output declaration of the *HYSTERESIS* block. It takes three real values as inputs: a sensor signal value *XIN1*, a set-point value *XIN2*, and a hysteresis band (or dead band) size *EPS*. It outputs a Boolean alarm Q. Figure 6f shows the expected input-output relation of the *HYSTERESIS* block. The output alarm Q is tripped (i.e., set to *true*) if the sensor value is strictly above the hysteresis band (i.e., to the left of the closed interval $[XIN2 - EPS,\ XIN2 + EPS]$), and not tripped if it is

strictly below the band. Otherwise, if the sensor value is within the hysteresis band[2], the alarm value Q remains unchanged. Figure 6c shows the (Pascal-like) structured text (ST) implementation supplied by IEC 61131-3[3]. The ST implementation trips the alarm Q when the alarm is currently not tripped (i.e., $\neg\ Q$), and when the input signal value is above the hysteresis band (i.e., $XIN1 > XIN2 + EPS$); and similarly for when to un-trip the alarm. The no-change case is covered by the two implicit "else" statements.

Figure 6d shows the input-output declaration of the *LIMITS_ALARM* block. It takes four real values as inputs: a sensor signal value X, a high limit H, a low limit L, and a hysteresis band size *EPS*. It outputs three Boolean alarms: high alarm QH, low alarm QL, and system Q. Figure 6g shows the expected input- output relation of the *LIMITS_ALARM* block. The *LIMITS_ALARM* block works by running two instances of the *HYSTERESIS* blocks: one for computing the high alarm *QH*, and the other for the low alarm *QL*. The output high alarm *QH* (resp., *QL*) is tripped if the sensor value is strictly above (resp., below) the hysteresis band $[H - EPS,\ H]$ (resp., $[L,\ L + EPS]$), and not tripped if it is strictly below (resp., above) the band. Otherwise, if the sensor value is within the hysteresis band, the alarm value *QH* (resp., *QL*) remains unchanged. The system alarm Q is tripped as long as either *QH* or *QL* is tripped. The FBD implementation supplied by IEC 61131-3 (Figure 6e) invokes two instances *HYSTERESIS* $(X,\ H - \frac{EPS}{2},\ \frac{EPS}{2})$ and *HYSTERESIS* $(L + \frac{EPS}{2},\ X,\ \frac{EPS}{2})$. The input *EPS* of the *LIMITS_ALARM* is not the same as that of the *HYSTERESIS* block. Consequently, the variable *w1* is used to capture the result of computing the half of *EPS* input to *LIMITS_ALARM*, and variables *w2* and *w3* capture results of, respectively, the addition and subtraction.

Step 1: Verification of *HYSTERESIS* Implementation. The TTM model in Figure 7a conforms to the structure in Figure 7b: to separate the *HYSTERESIS* block and its operating environment (or the plant). The input-output instantiation corresponds to that of Figure 6b (p. 17).

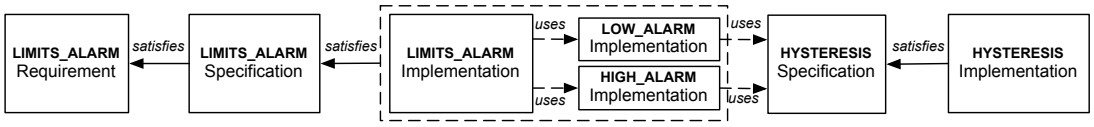
Abstraction on Input Values. The TTM tool, like other model checking tools, cannot handle the real-valued inputs *XIN1*, *XIN2*, and *EPS*. Instead, we choose partition the infinite domain of *XIN1* into five disjoint intervals: (1) $XIN1 < XIN2 - EPS$; (2) $XIN1 = XIN2 - EPS$; (3) $XIN2 - EPS < XIN1 < XIN2 + EPS$; (4) $XIN1 = XIN2 + EPS$; and (5) $XIN1 > XIN2 + EPS$. We fix inputs *EPS* and *XIN2* as integer constants, and accordingly, construct a finite integer set *SIGNAL_RANGE* that covers all the five intervals. We perform reachability checks to ensure that the chosen *SIGNAL_RANGE* is complete.

In module *PLANT*, we declare an event *generate* that updates the signal value X via a demonic assignment (i.e., $X :: SIGNAL\_RANGE$)[4]. The specified lower and upper time

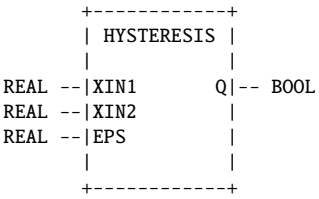
[2]In practice, considering the oscillation of the input signal value, the hysteresis region (of size $2 \times EPS$) is meant to prevent the output alarm value Q from alternating too often.

[3]In IEC 61131, integers 0 and 1 are used as Boolean values while our TTM model uses **true** and **false** instead.

[4]When creating the instance *env*, this variable is renamed to *XIN1*.



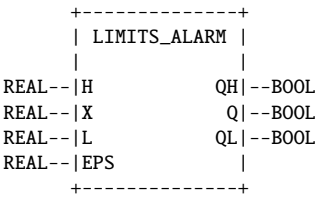
(a) Outline of Verification for the *HYSTERESIS* and *LIMITS_ALARM* Function Blocks



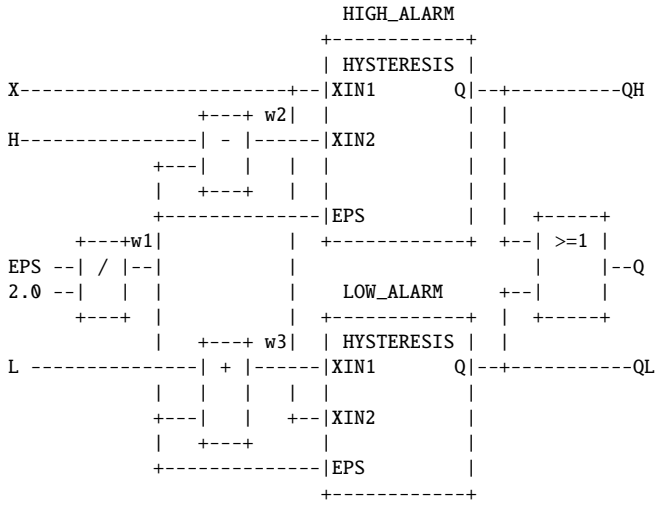
(b) *HYSTERESIS* Block: Declaration

```
FUNCTION_BLOCK HYSTERESIS
  VAR_INPUT XIN1, XIN2, EPS : REAL; END_VAR
  VAR_OUTPUT Q : BOOL := 0; END_VAR
  IF Q THEN IF XIN1 < (XIN2 - EPS) THEN Q := 0; END_IF ;
  ELSIF XIN1 > (XIN2 + EPS) THEN Q := 1 ;
  END_IF ;
END_FUNCTION_BLOCK
```

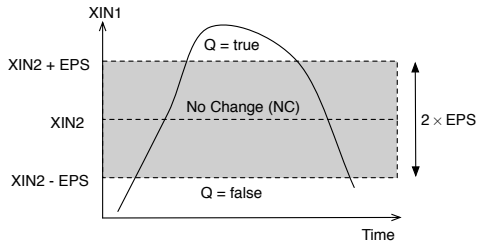
(c) *HYSTERESIS* Block: ST Implementation



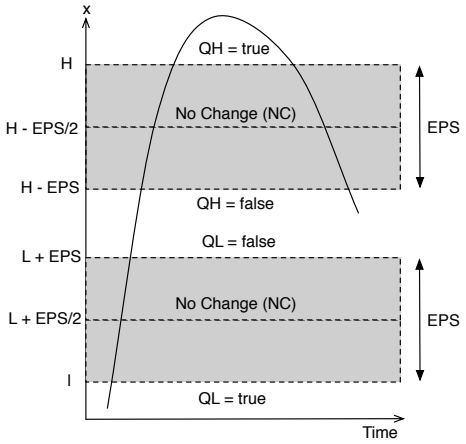
(d) *LIMITS_ALARM* Block: Declaration



(e) *LIMITS_ALARM* Block: FBD Implementation



(f) *HYSTERESIS* Block: Behaviour



(g) *LIMITS_ALARM* Block: Behaviour

Figure 6: *HYSTERESIS* and *LIMITS_ALARM* from IEC 61131-3 [8]

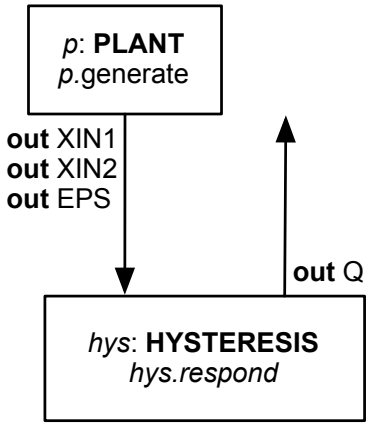
```
module HYSTERESIS
interface
  XIN1: in INT
  XIN2: in INT
  EPS : in INT
  Q : out BOOL = false
local
  q_old : BOOL = false
events
  respond[1, 1]
  do q_old := Q,
     if Q then
      if XIN1 < XIN2 − EPS
       then Q := false
      else skip fi
     elseif XIN1 > XIN2 + EPS
      then Q := true
     else skip fi
  end
end
```

```
module PLANT
interface
  X : out INT = DEFAULT
events
  generate[2, ∗]
  do
    X :: SIGNAL_RANGE
  end
end

instances
  env = PLANT(out XIN1)
  hys = HYSTERESIS(
    in XIN1, in XIN2, in EP
end

composition
  system = env || hys
end
```

(a) TTM



(b) Structure

Figure 7: Modelling *HYSTERESIS* in TTM

bounds for event *generate* indicate that each signal change from the plant occurs at the fastest rate of every two ticks, or it may never occur. On the other hand, to ensure that the *HYSTERESIS* block responds fast enough to input changes from the plant, we specify the upper time bound of the *respond* event being strictly smaller than the lower time bound of the *generate* event. Also, in the *HYSTERESIS* module, we declare a local variable *q_old*, later used for defining its functional specification, to record value for the output alarm Q that is last updated. Action of the *respond* is a straightforward translation from the ST implementation (Figure 6c) using if-statements.

The specification for the composed *system* is based on the three partitions of input signal *XIN1*:

$$spec \triangleq \begin{pmatrix} & (XIN1 > XIN2 + EPS & \Rightarrow & Q) \\ \wedge & (XIN2 - EPS \leq XIN1 \leq XIN2 + EPS & \Rightarrow & Q = hys.q\_old) \\ \wedge & (XIN1 < XIN2 - EPS & \Rightarrow & \neg Q) \end{pmatrix} \tag{9}$$

However, the assertion $system \models \Box\ spec$ fails as the specified time bounds $[1,\ 1]$ for the *hys.respond* event ensure that the response is, realistically, not instantaneous. Instead, we

assert that the *hys.respond* event is able to establish the above functional specification.

$$\Box(\ hys.respond \Rightarrow spec\ ) \tag{10}$$

$$\Box(\ spec \Rightarrow (spec\ \mathbf{W}\ env.generate\ ) \tag{11}$$

$$\Box(\ env.generate \Rightarrow (\bigcirc\ \neg\ env.generate)\ \mathbf{U}\ hys.respond\ ) \tag{12}$$

Equation 10 states that the ST implementation satisfies its intended specification *spec*. Equation 11[5] states that only the spontaneous action of the environment can break *spec*. Equation 12 states that the hysteresis block responds each environment change before the next one occurs.

Step 2: Verification of *LIMITS_ALARM* Implementation. The TTM model in Figure 8a conforms to the structure of synchronization in Figure 8b. Following the same rationale from the previous step, we partition the infinite domain of the monitored signal X into disjoint intervals: (1) $X < L$; (2) $X = L$; (3) $L < X < L + EPS$; (4) $X = L + EPS$; (5) $L + EPS < X < H - EPS$; (6) $X = H - EPS$; (7) $H - EPS < X < H$; (8) $X = H$; and (9) $X > H$. We then fix inputs *EPS*, H, and L as integer constants, and accordingly, construct a finite integer set *SIGNAL_RANGE* that covers all the intervals.

Value of the system alarm Q depends on those of the low alarm QL and high alarm QH, each calculated using the *HYSTERESIS* block (Figure 6e, p. 17). As a result, synchronous events are suitable for modelling such data dependancy. In module *LIMITS_ALARM*, we declare that: (1) it depends on two instances *low* and *high* of module *HYSTERESIS*; (2) its *respond* event synchronizes, as one atomic step, with the *respond* event from the two instances; and (3) the new synchronous event is renamed as *respond* for use in assertions. When creating the instance *limits*, its two dependent *HYSTERESIS* instances are bound to *low* and *high* (via a **with** ... **end** clause). We also rename the synchronization of these three instances (via the *::=* operator) so that the automatically-generated action of event *alarms.respond* combines actions of updating Q, QL, and QH. As we can see from the composition, the plant instance *env* is separated from the synchronous instance *alarms*. Again, time bounds of the two events *env.generate* and *alarms.respond* are specified such that the alarms can respond fast enough to signal changes from the plant.

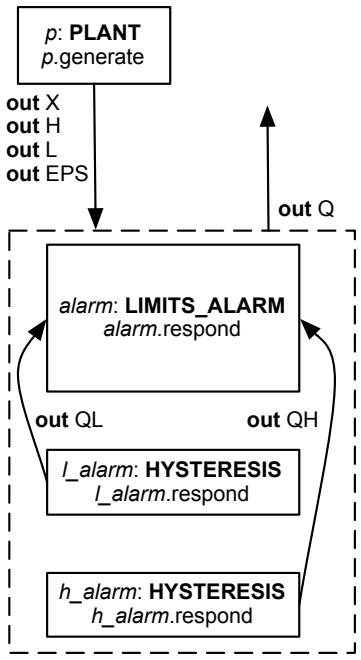
The use of synchronous events allows us to decompose the updates (on Q, QL, and QH) into three *respond* events in instance *limits* of module *LIMITS_ALARM* and two instances *low* and *high* of module *HYSTERESIS*. In the previous step of verification, we verified that the action of event *respond* in module *HYSTERESIS* (Figure 7a) satisfies the high-level specification of Equation 9. As far as the calculations of QL and QH are concerned, it suffices to replace the verified action with a version that encodes, using if-statements, Equation 9. On the other hand, the *respond* event in module *LIMITS_ALARM* reference

[5]The textual TTM assertion language does not support the weak-until operator **W**. Instead, we check an equivalent safety property using the next ($\bigcirc$) and release (**R**) operators: $\Box(\ spec \Rightarrow (\bigcirc\ env.generate)\ \mathbf{R}\ spec\ )$.

```
module LIMITS_ALARM
  depends
    low : HYSTERESIS
    high: HYSTERESIS
  interface
    QH: in BOOL
    QL: in BOOL
    Q : out BOOL = false
  events
    respond[1, 1]
    sync
      low.respond,
      high.respond as respond
    do
      if QH' || QL' then
        Q := true
      elseif !QH' && !QL' then
        Q := false
      else
        skip
      fi
   end
end
```

```
module PLANT
  interface
    X : out INT = SIGN
  events
    generate[2, *]
    do
      X :: SIGNAL_RANG
    end
end
instances
  env = PLANT (...)
  high = HYSTERESIS (.
  low = HYSTERESIS (..
  limits = LIMITS_ALAR
    with
      high := high, low :=
    end
  alarms ::= low || high || l
end
composition
  system = env || alarms
end
```

(a) TTM with Synchronous Events



(b) Sync Structure

Figure 8: Modelling *LIMITS_ALARM* in TTM

primed values of the low and high alarms (i.e., *QL'* and *QH'*), indicating that other parts of the same synchronization should first update those variables. Moreover, the action of the *respond* event in module *LIMITS_ALARM* is structured to correspond to that of its FBD implementation (Figure 6e, p. 17), except we eliminate the intermediate variables *w1*, *w2*, and *w3* by passing the correct value expressions in the instantiations.

Properties that we assert for the composed system of *LIMITS_ALARM* are identical to those for *HYSTERESIS*, except that Equation 9 is revised to reflect the new input partitions.

$$spec \triangleq \left(\begin{array}{llllllll} & (H & < & X & & \Rightarrow & QH \wedge \neg QL \wedge Q) \\ \wedge & (H - EPS & \leq & X & \leq H & \Rightarrow & QH = high.q\_old = Q \wedge \neg QL) \\ \wedge & (L + EPS & < & X & < H - EPS & \Rightarrow & \neg QH \wedge \neg QL \wedge \neg Q) \\ \wedge & (L & \leq & X & \leq L + EPS & \Rightarrow & QL = low.q\_old = Q \wedge \neg QH) \\ \wedge & (X & < & L & & \Rightarrow & \neg QH \wedge QL \wedge Q) \end{array} \right) \tag{13}$$

Step 3: Validation of *LIMITS_ALARM* Specification. Following the same rationale from the previous step, as far as the calculations of Q, QL and QH are concerned, it suffices to replace the verified synchronous event with a version whose action encodes, using if-statements, Equation 13. We then validate the specification of *LIMITS_ALARM* with respect to invariant that is not immediately obvious from its specification. For example, we assert that the high and low alarms are not tripped at the same time.

$$\Box(\ \neg\ (QH \wedge QL)\ ) \tag{14}$$

To conclude this section, we note that our specifications for *HYSTERESIS* (Equation 9) and *LIMITS_ALARM* (Equation 13) assume that the hysteresis band size is positive (i.e., $EPS > 0$). Furthermore, Equation 13 assumes that the hysteresis bands for high and low alarms are disjoint (i.e., $H - EPS > L + EPS$). When the chosen constants for EPS, H, and L do not meet these assumptions, the TTM tool is able to find the appropriate counter-examples.

4.3 Example: Tabular Requirement of a Nuclear Shutdown System

We illustrate the use of synchronous events on a slice of the software requirements of a shutdown system for the Darlington Nuclear Generating Station in Ontario, Canada. We present two versions of the system. The first version presents a high-level requirements [18] where the controller responds instantaneously to environment changes. We synchronize both the environment and controller events to model such instantaneity, and check it via an invariant property. However, for such requirements model to be implementable, some specified allowance on the controller's response is required [19]. The second, refined version illustrates how we can incorporate the response allowance as time bounds on the environment and controller events (i.e, that the controller responds fast enough to environment changes). We also decouple the controller from the environment, and check its response via a real-time liveness property.

Requirements of the shutdown system are described mathematically using tabular expressions (a.k.a. function tables) [9]. Figure 9 exemplifies tabular requirements for two units: Neutron OverPower (NOP) Parameter Trip (Figure 9a) and Sensor Trips (Figure 9b). In the first column, rows are Boolean conditions on monitored variables (i.e., input stimuli). In the second column, the first row names a controlled variable (i.e., output response); the remaining rows specify a value for that controlled variable. We use the formalism of tabular expressions to check the completeness (i.e., no missing cases from the row conditions) and the disjointness (i.e., no two row conditions are satisfied simultaneously) of our requirements [9].

The NOP Parameter Trip unit (which we call the NOP controller) depends on 18 instances of the Sensor Trip units (which we call the NOP sensors). There are two monitored variable for each NOP sensor i: (1) a floating-point calibrated NOP signal value

Condition	*Result* $c\_NOPparmtrip$
$\exists i \in 0\,..\,17 \bullet f\_NOPsentrip[i] = e\_Trip$	e_Trip
$\forall i \in 0\,..\,17 \bullet f\_NOPsentrip[i] = e\_NotTrip$	$e\_NotTrip$

(a) Function Table for NOP Controller

Condition	*Result* $f\_NOPsentrip[i]$
$calibrated\_nop\_signal[i] \geq f\_NOPsp$	e_Trip
$f\_NOPsp - k\_NOPhys < calibrated\_nop\_signal[i] < f\_NOPsp$	$(f\_NOPsentrip[i])_{-1}$
$calibrated\_nop\_signal[i] \leq f\_NOPsp - k\_NOPhys$	$e\_NotTrip$

(b) Function Table for NOP Sensors (sensor i monitors $calibrated\_nop\_signal[i]$, $i \in 0\,..\,17$)

Figure 9: Tabular Requirement for the Neutron Overpower (NOP) Trip Unit

calibrated_nop_signal$[i]$; and (2) a floating-point set point value *f_NOPsp*. The monitored signal is bounded by the two pre-set constants *k_NOPLoLimit* and *k_NOPHILimit*. The monitored set point is typed to a set of four constants: *k_NOPLPsp* (low-power mode), *k_NOPAbn2sp* (abnormal mode 2), *k_NOPAbn1sp* (abnormal mode 1), and *k_NOPnormsp* (normal mode).

Each sensor i determines if the monitored signal goes above a safety range (i.e., $\geq$ *f_NOPsp*), in which case it trips by setting the function variable *f_NOPsentrip*$[i]$ to *e_Trip*. To prevent the value of *f_NOPsentrip* from alternating too often due to signal oscillation, a hysteresis region (or dead band) with constant size *k_NOPhys* is created. That is, the hysteresis region is an open interval (*f_NOPsp* $-$ *k_NOPhys*, *f_NOPsp*). When the monitored signal falls within this region, then the new value of *f_NOPsentrip* remains as that in the previous state, denoted as $f\_NOPsentrip_{-1}$. On the other hand, the NOP controller is responsible for setting the controlled variable *c_NOPparmtrip*, based on values of *f_NOPsentrip*$[i]$ from all its dependant sensors. If there is at least one sensor that trips, then the NOP parameter trips by setting *c_NOPparmtrip* to *e_Trip*.

According to the requirements, the system is initialized in a conservative manner. Each calibrated NOP signal is set to its low limit *k_NOPLoLimit*, but each *f_NOPsentrip*$[i]$ for sensor i and the controlled variable *c_NOPparmtrip* are all set to *e_Trip*. As we will see in our specification below (i.e., Equation 16), to ensure that the system satisfies the tabular specification in Figure 9, the NOP controller must have completed its very first response (denote as predicate $\neg$*init_response*).

The requirements model in Figure 9 uses a finite state machine, with an arbitrarily small clock tick, that describes an idealized behaviour. At each time tick t, monitored and

controlled variables are updated instantaneously. State data such as $f\_NOPsentrip_{-1}$ are stored and used for the next state. However, to make such requirements implementable, some allowance on the controller's response must be provided [19]. As a result, we present two versions of the NOP system in TTM: (1) an abstract version with plant and controller taking synchronized actions; and (2) a refined version with the response allowance incorporated as time bounds of the environment and controller events. The refined version allows us to assert timed response properties (e.g., once the monitored signal goes above the safety range, the controller trips within 2 ticks of the clock).

Abstraction of Input Signal Values. The TTM tool, like other model checking tools, cannot handle the real-valued monitored variables *f_NOPsp* and *calibrated_nop_signal*$[i]$. Instead, based on the given constants mentioned above, we partition the infinite domains of these two monitored variables into disjoint intervals. First, we know that the four possible constant values for *f_NOPsp* have a fixed order and are bounded by constant low and high limits of the calibrated NOP signal. More precisely, we have 6 boundary cases to consider:

$$k\_NOPLoLimit < k\_NOPLPsp < k\_NOPAbn2sp < k\_NOPAbn1sp < k\_NOPnormsp < k\_NOPHiLimit$$

Second, each of the four possible set points has an associated hysteresis band, whose lower boundary is calculated by subtracting the constant band size *k_NOPhys*, resulting in 4 additional boundaries[6] to consider: (a) $k\_NOPLPsp - k\_NOPhys$; (b) $k\_NOPAbn2sp - k\_NOPhys$; (c) $k\_NOPAbn1sp - k\_NOPhys$; and (d) $k\_NOPnormsp - k\_NOPhys$. Consequently, we have 10 boundary cases and 9 in-between cases (e.g., $k\_NOPLoLimit < signal < k\_NOPLPsp$) to consider. Accordingly, we construct a finite integer set *cal_nop* that covers all the 19 intervals. We perform reachability checks to ensure that the chosen *cal_nop* is complete.

For the purpose of modelling and verifying the NOP controller and sensors in TTM, we parameterize the system by a positive integer N denoting the number of dependant sensors.

Version 1: Synchronizing Plant and Controller. We first present an abstract version of the model that couples the NOP controller and its plant by executing their actions synchronously. Figure 10 illustrates the structure of synchronization. The dashed box in Figure 10 indicates the set of synchronized modules instances: plant p, controller *nop*, and 18 sensors *sensor_i* ($i \in 0 \,..\, 17$).

Figure 12 (p. 27) presents the complete[7] TTM listing of the NOP unit as described above. The *generate* event of the plant non-deterministically updates the value of a global array that is shared with sensors attached to the NOP controller. The update is per-

[6] Value of (a) is still greater than *k_NOPLoLimit*, and similarly value of (d) is still smaller than *k_NOPHiLimit*.

[7] For clarity, we present a version with one monitoring sensor. The full version with 18 sensors involves just declaring and instantiating additional dependent sensors. We also exclude definitions of constants and assertions.

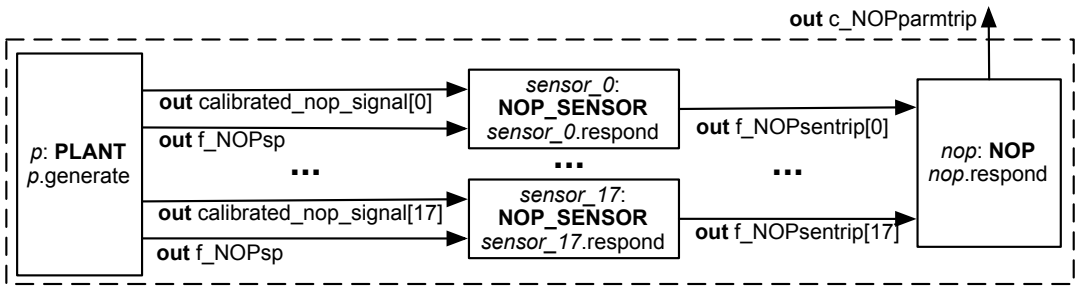


Figure 10: Neutron Overpower (NOP): Abstract Version – Synchronized Plant and Controller

formed via the demonic assignment *calibrated_nop_signals* :: ***ARRAY***[*cal_nop*](N) (Lines 5 – 6). On the other hand, the NOP controller module (Lines 8 – 26) depends on two module instances (Lines 9–11). First, the controller depends on a plant p that generates an array of calibrated NOP signals (specified by the **out** array argument *calibrated_nop_signal* at Lines 4 and 47). Second, the controller depends on a sensor *sensor_0* that monitors a particular signal value (specified by the **in** argument *calibrated_nop_signal*[0] at Lines 30 and 48) and provides feedback (specified by the **share** argument *f_NOPsentrip*[0] at Line 31 and 48) for the central NOP controller to make a final decision (specified by the **out** argument *c_NOPparmtrip* at Lines 14 and 49).

Actions of the *respond* events of the NOP controller (Lines 19 – 24) and of its dependent sensors (Lines 36 – 43) correspond to the tabular requirements (Figure 9a and Figure 9b, respectively). We use primed variables in these actions to specify the intended flow of actions. Actions of the NOP sensor reference *f_NOP'* and *calibrated_nop_signal_i'* (Lines 37, 39, and 41) to indicate, that only after the instance p (in the same synchronous set) has written to these two variables can they be used to calculate the new value of *f_NOPsentrip*[i]. Similarly, actions of the NOP controller reference *f_NOPsentrip'*[j] (Lines 20 and 22) to indicate, that only after all sensor instances have written to this array can it be used to calculate the new value of *c_NOPparmtrip*.

We require that the *respond* event of the NOP controller, the *respond* events of its dependent sensors, and the *generate* event of the plant, are always executed synchronously (as a single transition). In declaring the controller event *respond*, we use a ***sync*** ... ***as*** ... clause to specify the events to be included in the synchronous set. When instantiating the NOP controller, we use a ***with*** ... ***end*** clause to bind its dependent plant and sensor instances (Line 49). Finally, we rename the synchronized plant, controller, and sensor instances for references in assertions (Line 50).

We check two invariant properties on this abstract version of NOP. First, as all depen-

dent sensors have written to the shared array *f_NOPsentrip*, the NOP controller responds instantaneously.

$$\Box \left(\begin{array}{ll} & (\exists i : 0 \,..\, N \bullet f\_NOPsentrip[i] = e\_Trip) \Rightarrow c\_NOPparmtrip = e\_Trip \\ \wedge & (\forall i : 0 \,..\, N \bullet f\_NOPsentrip[i] = e\_NotTrip) \Rightarrow c\_NOPparmtrip = e\_NotTrip \end{array} \right) \quad (15)$$

Second, since all actions of the plant, the NOP controller, and sensors are synchronized together, we can assert that the controlled variable *c_NOPparmtrip* is updated as soon as the plant has updated the two monitored variables *f_NOPsp* and *f_NOPsentrip*.

$$\Box \left(\left(\begin{array}{ll} & \neg\ init\_response \\ \wedge & f\_NOPsp = k\_NOPLPsp \\ \wedge & k\_NOPLPsp \leq calibrated\_nop\_signal[0] \leq k\_CalNOPHiLimit \\ & \quad \Rightarrow c\_NOPparmtrip = e\_Trip \end{array} \right) \right) \quad (16)$$

However, the satisfaction of Equation 16 is an idealized behaviour without the realistic concern of some allowance on the controller's response [19]. That is, we shall instead allow the state predicate *c_NOPparmtrip* = *e_Trip* to be established within a bounded delay.

Version 2: Separating Plant and Controller. We refine the TTM of NOP in Figure 12 by decoupling actions of the controller[8] and its plant. Figure 11 illustrates the refined structure of synchronization: the plant instance p is no longer synchronized with the controller. Consequently, the plant event *generate* and the synchronous controller event *respond* are interleaved.

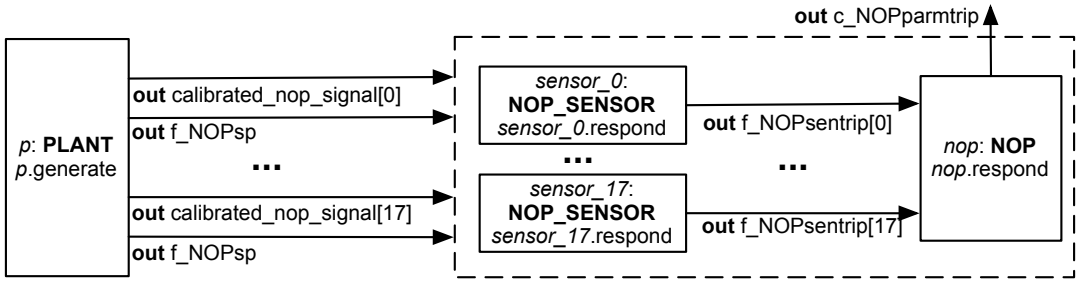


Figure 11: Neutron Overpower (NOP): Refined Version – Separate Plant and Controller

The resulting system would fail to satisfy Equation 16, as we introduce some allowance on the response time (termed *response allowance* in [19]) of the NOP controller to en-

[8]In the NOP controller, actions of the NOP parameter trip unit and sensor units remain synchronized.

vironment changes. On the other hand, as we still consider the controller's response actions, once initiated, take effect instantaneously, the resulting system should still satisfy Equation 15.

Minimal changes are required to produce the refined TTM from Figure 12. First, in module *NOP*, we remove the declaration of *p : PLANT* as a dependent instance (Line 10). We also remove the declaration of *p.generate* as an event to be synchronized with the *respond* event (Line 17). Second, in creating the instance *nop* of module *NOP*, as it no longer depends on a *PLANT* instance, we remove the binding statement (Line 49), i.e., *env := env*. Third, in renaming the synchronous instance, we remove the plant instance (Line 50), i.e., *controller ::= sensor_0 || nop*. Finally, we add the plant instance into the composition (Line 52), i.e., *system = env || controller*.

By declaring a timer t and adding a ***start*** t clause to the *generate* event in module *PLANT* (Line 6), we can satisfy the following real-time response property:

$$\Box \left(\left(\begin{array}{ll} & f\_NOPsp = k\_NOPLPsp \\ \wedge & k\_NOPLPsp \leq calibrated\_nop\_signal[0] \leq k\_CalNOPHiLimit \\ \wedge & t = 0 \\ & \quad \Rightarrow mono(t) \ \ \mathbf{U} \ \ (\ c\_NOPparmtrip = e\_Trip \wedge t < 2\ ) \end{array} \right) \right) \tag{17}$$

As soon as the set point value and monitored signal value are updated by the plant, the controller produces the proper response within two ticks of the clock. Before the controller responds, timer t must not be interrupted (i.e., reset by other events), so as not to provide an inaccurate estimate.

To conclude this section, we note that the demonic assignment performed by the plant's *generate* event (Line 6 in Figure 12, p. 27) imposes no constraints on the change of NOP signal. With the assumption that signal values do not suddenly increase or decrease dramatically, we are able to: (a) let the plant generate $delta \in \{-1,\ 0,\ 1\}$ for each monitored signal; and (b) let the sensor estimate what the actual signal value is accordingly. This effectively reduces the number of possibilities to consider for array *calibrated_nop_signal*.

5 Discussion

In this paper, we report how our new TTM notations facilitate the formal validation of cyber-physical system requirements. In the mutual exclusion protocol (Section 3.2 and train control system (Section 3.3), the indexing construct allows us to select a specific actor (i.e., a process or a train) and specify a temporal property for that actor. The synchronous construct, together with primed variables, allow us to check (real-time) response properties of the function blocks from the IEC 61131-3 Standard (Section 4.2) and tabular requirements of a nuclear shutdown system (Section 4.3).

```
1  module PLANT      /* Template for Nuclear Reactor */
2    interface
3      f_NOPsp : out INT = k_NOPLPsp
4      calibrated_nop_signal : out ARRAY[cal_nop](NUM_SENSORS) = [k_CalNOPLoLimit (
           NUM_SENSORS)]
5    events generate[1, 1]
6              do calibrated_nop_signal :: ARRAY[cal_nop](NUM_SENSORS), f_NOPsp := k_NOPLPsp
                   end
7  end
8  module NOP      /* Template for Neutron Overpower Controller */
9    depends
10     env : PLANT
11     sensor_0 : NOP_SENSOR
12   interface
13     f_NOPsentrip : share ARRAY[y_trip](NUM_SENSORS)       /* shared, but read only */
14     c_NOPparmtrip : out y_trip = e_Trip
15   local after_init_response : BOOL = false
16   events
17     respond[1, 1] sync env.generate, sensor_0.respond as respond
18     do
19       after_init_response := true,
20       if (|| j: 0..(NUM_SENSORS−1) @ f_NOPsentrip'[j] == e_Trip) then
21         c_NOPparmtrip := e_Trip
22       elseif (&& k: 0..(NUM_SENSORS−1) @ f_NOPsentrip'[k] == e_NotTrip) then
23         c_NOPparmtrip := e_NotTrip
24       else skip fi
25     end
26 end
27 module NOP_SENSOR      /* Template for Sensors */
28   interface
29     f_NOPsp : in INT
30     calibrated_nop_signal_i : in cal_nop
31     f_NOPsentrip_i : share y_trip       /* shared, but write only */
32   local f_NOPsentrip_i_old : y_trip = e_Trip
33   events
34     respond[1, 1]
35     do
36       f_NOPsentrip_i_old' = f_NOPsentrip_i,
37       if f_NOPsp' <= calibrated_nop_signal_i' then
38         f_NOPsentrip_i := e_Trip
39       elseif (f_NOPsp' − k_NOPhys < calibrated_nop_signal_i') && (calibrated_nop_signal_i' < f_NOPsp
             ') then
40         f_NOPsentrip_i := f_NOPsentrip_i_old
41       elseif calibrated_nop_signal_i' <= f_NOPsp' − k_NOPhys then
42         f_NOPsentrip_i := e_NotTrip
43       else skip fi
44     end
45 end
46 instances
47   env = PLANT (out f_NOPsp, out calibrated_nop_signal)
48   sensor_0 = NOP_SENSOR(in f_NOPsp, in calibrated_nop_signal[0], share f_NOPsentrip[0])
49   nop = NOP(share f_NOPsentrip, out c_NOPparmtrip) with env := env, sensor_0 := sensor_0 end
50   sys ::= env || sensor_0 || nop       /* named synchronous instance */
51 end
52 composition system = sys end
```

Figure 12: Requirement of NOP Trip Unit in TTM: Synchronized Plant and Controller

To our knowledge, the introduced notations of indexed events and synchronous events (and its combination with primed variables) are novel. For synchronous events, the conventional Communicating Sequential Processes (CSP) [15] and its tool [5] support multi-way synchronization by matching names of event in parallel compositions. However, the conventional CSP does not allow processes to modify a shared state. Instead, the system state can only be managed as parameters of recursive processes, making it impossible to synchronize events that denote different parts of simultaneous updates. The notations of un-timed CSP# and the stateful timed CSP (extended with real-time process operators such as time-out, deadline, etc.) [16] allow events to be attached with state updates. However, their semantics and tool support do not allow events that are attached with attached to be synchronized. The UPPAAL model checker and its language of timed automata [10] support the notion of broadcast channel for synchronizing multiple state-updating transitions (one sender and multiple receivers). However, the RHS of assignments can only reference values evaluated at the pre-state. There is no mechanism, such as the notion of primed variables supported in TTM, for specifying the intended data flow.

Also relevant to our synchronous events is the recent work on using the PVS proof assistant to verify the IEC 61131 function blocks [14]. Their notion of timed variables (i.e., variables parameterized by discretized time units) makes it straightforward for specifying expressions at the pre- and post-states. However, their semantics results in idealized systems (where the controlled and monitored variables are updated simultaneously). The lack of response allowance [19] in their approach makes it nonsensical to check real-time response properties

For indexed events, the tool support for both conventional CSP [15] and UPPAAL [10] do not allow the verification with fairness assumptions. In the case of UPAAL, it is likely to manually construct an observer. However, such solution does not scale in large systems and is prone to errors. On the other hand, the PAT tool allows users to choose fairness assumptions at the event, process, or global level [17] for verifying the un-timed CSP# and stateful timed CSP [16]. However, our extended notion of indexed events are of finer-grained for imposing fairness assumptions, as we allow the declaration of a subset of event indices as fair.

References

[1] J.-R. Abrial. *Modeling in Event-B.* Cambridge University Press, 2010.

[2] Darren Cofer and Steven Miller. Do-333 certification case studies. In *NASA Formal Methods*, volume 8430 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 2014.

[3] Patricia Derler, Edward A. Lee, and Alberto Sangiovanni-Vincentelli. Modeling cyber-physical systems. *Proceedings of the IEEE (special issue on CPS)*, 100(1):13 – 28, 2012.

[4] RTCA & EUROCAE. DO-333 – formal methods supplement to DO-178C and DO-278A. Technical report, December 2011.

[5] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and AndrewW. Roscoe. FDR3 – a modern refinement checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *LNCS*, pages 187–201. Springer, 2014.

[6] Simon Hudon, Thai Son Hoang, and Jonathan Ostroff. The Unit-B method – refinement guided by progress concerns. *Software and Systems Modeling (SoSyM)*, 2014. Accepted.

[7] Simon Hudon and ThaiSon Hoang. Systems design guided by progress concerns. In *Integrated Formal Methods*, volume 7940 of *LNCS*, pages 16–30. Springer, 2013.

[8] IEC. *61131-3 Ed. 2.0 en:2003: Programmable Controllers — Part 3: Programming Languages*. International Electrotechnical Commission, 2003.

[9] Ryszard Janicki, DavidLorge Parnas, and Jeffery Zucker. Tabular representations in relational documents. In *Relational Methods in Computer Science*, Advances in Computing Sciences, pages 184–196. Springer Vienna, 1997.

[10] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[11] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, New York, 1992.

[12] Jonathan S. Ostroff. Composition and refinement of discrete real-time systems. *ACM Transaction on Software Engineering Methodology*, 8(1):1–48, 1999.

[13] JonathanS. Ostroff, Chen-Wei Wang, Simon Hudon, Yang Liu, and Jun Sun. Ttm/-pat: Specifying and verifying timed transition models. In *FTSCS*, volume 419 of *Communications in Computer and Information Science*, pages 107–124. Springer, 2014.

[14] Linna Pang, Chen-Wei Wang, Mark Lawford, and Alan Wassyng. Formalizing and verifying function blocks using tabular expressions and pvs. In *FTSCS*, volume 419 of *Communications in Computer and Information Science*, pages 125–141. Springer, 2014.

[15] A.W. Roscoe. *Understanding Concurrent Systems*. Springer, 1st edition, 2010.

[16] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed csp. *ACM Trans. Softw. Eng. Methodol.*, 22(1):3:1–3:29, 2013.

[17] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, LNCS 5643, pages 709 – 714, 2009.

[18] A. Wassyng and M. Lawford. Software tools for safety-critical software development. *STTT*, 8(4-5):337–354, 2006.

[19] A. Wassyng, M. Lawford, and X. Hu. Timing tolerances in safety-critical software. In *FM*, pages 157–172, 2005.

6 Appendix: Semantics of TTM (extended for Indexed & Synchronous Events)

We provide a one-step operational semantics for TTMs.

6.1 Abstract Syntax.

Following [12] and using the mathematical conventions of Event-B [1], we define the abstract syntax of a TTM module instance $\mathcal{M}$ as a 5-tuple, i.e., $\mathcal{M} = (V, s_0, T, t_0, E)$ where 1) V is a set of variable identifiers, declared local or in a module interface; 2) T is a set of timer identifiers; 3) E is a set of events that may change the state; 4) $s_0 \in \text{STATE}$ is the initial variable assignment, with $\text{STATE} \triangleq V \to \text{VALUE}$; and 5) $t_0 \in \text{TIME}$ is the initial timer assignment, with $\text{TIME} \triangleq T \to \mathbb{N}$.

Concrete syntax of event e:

event_id (x : **fair** T_x; $y : T_y$) [l,**u**] **just**
when *grd*
start t_1, t_2
stop t_3, t_4
do $v_1 := exp_1$,
if *condition* **then** $v_2 := v_1' + exp_2$
else skip fi,
$v_3 :: 1..4$
end

Abstract syntax of the event e:

- $e.id \in \text{ID}$;
- $e.f\_ind \subseteq \text{ID}$; $e.d\_ind \subseteq \text{ID}$; $e.ind \triangleq e.f\_ind \cup d\_ind$
- $e.l \in \mathbb{N}$; $e.u \in \mathbb{N} \cup \{\infty\}$
- $e.fair \in \{\text{spontaneous}, \text{just}, \text{compassionate}\}$
- $e.grd \in \text{STATE} \times \text{TIME} \to \text{BOOL}$;
- $e.start \subseteq T$;
- $e.stop \subseteq T$;
- $e.action \in \text{STATE} \times \text{TIME} \leftrightarrow \text{STATE}$;

Figure 13: Concrete and Abstract syntax of TTM events

We use an 10-tuple $(id, f\_ind, d\_ind, l, u, fair, grd, start, stop, action)$ to define the abstract syntax of an event e, and we use the dot notation "." to access the fields, as shown on the right of Fig. 13. The string identifier of an event e is written as $e.id$. $e.f\_ind$ and $e.d\_ind$ are sets of indices that can be referenced in the event. As explained below, each index can be either fair (if it is in $e.f\_ind$) or demonic (if it is in $e.d\_ind$). The guard of event e, i.e., $e.grd$, is any Boolean expression referencing state variables (from V), timers (from T) or ; in the example on the left of Fig. 13, $V = \{v_1, v_2, v_3, \cdots\}$ and $T = \{t_1, t_2, t_3, t_4, \cdots\}$. Functions $boundt \in T \to \mathbb{N}$ and $type \in T \to \mathbb{P}(\mathbb{N})$ [9] provide, respectively, the upper bound and the type of each timer. For example, if timer t_1 is

[9] l and u could be arbitrary state expressions as long as $grd \implies l \leq u$ is invariant and $grd \wedge l \leq t \implies t \leq u$ is only falsified by the tick event, with t the timer of the event. these side conditions

declared in the TTM as $t_1 : 0..5$, then $boundt(t_1) = 5$ and $type(t_1) = \{0..6\}$. As will be detailed below, timers count up to one beyond the specified bound at which point they remain fixed until they are restarted.

An event e must be taken between its lower time bound $e.l$ and upper time bound $e.u$, provided that its guard $e.grd$ remains true. The event action involves simultaneous assignments to $v_1, v_2, \cdots$. The notation $v_3 :: 1..4$ is an example of a demonic assignment in which v_3 takes any value from 1 to 4. All the assignments in the event action are applied simultaneously in one step.

In an assignment $y := exp$, the expression on the right may use primed (e.g. x') and unprimed (e.g. x) state variables as well as the initial value of timers. A variable with a prime refers to the variable's value in the next state and a variable without prime refers to its value in the current state. The use of primed variables in expressions allows for simpler and more expressive descriptions of state changes. The state changes effected by an event e is described in the abstract syntax by a before-after predicate $e.action$. The concrete syntax also allows for assignments to be embedded in (possibly nested) conditional statements.[10]

6.2 Formal Semantics.

We provide a one-step operational semantics of a TTM module instance $\mathcal{M}$ in LTS (Labelled Transition Systems).

Definition: LTS: Given a TTM module instance $\mathcal{M}$, an LTS (Labelled Transition System) is a 4-tuple $\mathcal{L} = (\Pi, \pi_0, \mathbf{T}, \rightarrow)$ where 1) Π is a set of system configurations; 2) $\pi_0 \in \Pi$ is an initial configuration; 3) $\mathbf{T}$ is a set of transitions names (defined below); and 4) $\rightarrow \subseteq \Pi \times \mathbf{T} \times \Pi$ is a transition relation.

We now describe the LTS semantics of TTMs. We define E_{id} as the set of names of the transitions corresponding to events — as opposed to the monotonicity breaking transitions and the tick transition — and E_{fair} as the set of transition names prefixes, removing the value of the demonic indices:

$$E_{id} \triangleq \{e, m \mid e \in E \wedge m \in e.f\_ind \rightarrow \text{VALUE} \bullet (e.id, m)\} .$$

In the following, we use $e(x)$ to stand for the name of the transition corresponding to e with x the values of e's fair indices. When talking about the occurrence of e, in an

are mostly axiomatic and would be hard to check and use with a model checker. The latter constraint can be formulated as **stable** $grd \wedge l \leq t \implies t \leq u$ **in** M, with M standing for a TTM machine without the tick event. Reasonable alternatives for it would be **stable** $grd \implies l \leq t \leq u$ **in** M and $l = l_0 \wedge u = u_0$ **unless** $\neg grd$.

[10] With all the complexity of structures allowed by the syntax of actions, sequential composition is not allowed. This is in an effort to make actions into specifications rather than implementations. This would allow us to generalize TTMs to allow an Event-B style of symbolic reasoning.

LTL formula for instance, $e(x, y)$ also specify the values of e's demonic indices as y. The demonic indices are otherwise treated as internal non-deterministic choice within the event.

A configuration $\pi \in \Pi$ is defined by a 6-tuple (s, t, m, c, x, p). We explain each of the six components as follows:

• $s \in$ STATE is a value assignment for all the variables of the system. The state can be read and changed by any transition corresponding to an event in E.

• $t \in$ TIME is a value assignment for the timers of the system. Events (and hence their corresponding transitions) may only start, stop and read timers. As will be discussed below, we introduce a special transition, called *tick*, which also changes the timers. Timers t_i that are stopped have values $boundt(t_i) + 1$.

• $m \in T \rightarrow$ BOOL records the status of monotonicity of each timer. Suppose event e_1 in a TTM starts t_1. In LTL we might write $\Box(\ e_1 \wedge t_1 = 0 \ \rightarrow \ \Diamond(q \wedge t_1 \leq 4)\ )$ (note that $t_1 = 0$ is redundant) to specify that q becomes true within 4 time units of event e_1 occurring. However, other events might stop or restart t_1 before q is satisfied hence breaking the synchronicity between t_1 and a global clock.[11] Instead, we express the intended property as $\Box(\ e_1 \wedge t_1 = 0 \ \rightarrow \ m(t_1)\ \mathcal{U}\ (q \wedge t_1 \leq 4)\ )$. The expression $m(t_1)$ (standing for monotonicity of t_1) holds in any state where t_1 is not stopped or being reset. We explain monotonicity further below.

• $c \in E_{id} \rightarrow \mathbb{N} \cup \{-1\}$ is a value assignment for a clock implicitly associated with each event. These clocks are used to decide whether an event has been enabled for long enough and whether it is urgent. An event $e \in E$ is enabled when its clock's value is between the event's lower time bound (i.e., $e.l$) and its upper time bound (i.e., $e.u$). Furthermore, the type (or range) of $c(e.id, x)$ is $\{-1, 0, ...e.u\}$. When an event's clock is disabled, as opposed to the convention used with timers, the clock's value is -1.

• $x \in E_{id} \cup \{\perp\}$ is used as a sequencing mechanism to ensure that each transition e is immediately preceded by an $e\#$ transition whose only function is to update the monotonicity record m. For example, in the following execution $\cdots \stackrel{e_1}{\rightarrow} \underset{x=\perp}{\pi_1} \stackrel{e_2\#}{\rightarrow} \underset{x=e_2}{\pi_2} \stackrel{e_2}{\rightarrow} \underset{x=\perp}{\pi_3} \rightarrow \cdots$, suppose in π_1 the value of timer t_2 is 3 and that e_2 restarts t_2. Then, in π_2, we have $x = e_2 \wedge t_2 = 3 \wedge m(t_2) =$ false. In π_3, we have $x = \perp \wedge t_2 = 0 \wedge m(t_2) =$ true. In order to record the breaking of monotonicity, the $e_2\#$ transition sets $m(t_2)$ to false, which gets set back to true in the next execution step. The precise effect of these transitions will be described below.

[11] Suppose that event e_2 also starts t_1, that e_3 establishes q and that the events occur in the following order: $\pi_0 \stackrel{e_1}{\rightarrow} \underset{t_1=0}{\pi_1} \stackrel{tick^3}{\rightarrow} \underset{t_1=3}{\pi_4} \stackrel{e_2}{\rightarrow} \underset{t_1=0}{\pi_5} \stackrel{tick^2}{\rightarrow} \underset{t_1=2}{\pi_7} \stackrel{e_3}{\rightarrow} \underset{t_1=2 \ \wedge \ q}{\pi_8} \cdots$. This execution satisfies the first LTL formula but does not satisfy the intended specification: when q becomes true, $t_1 = 2$ but it is 5 ticks away from the last occurrence of e_1.

• $p \in E_{id} \cup \{tick, \bot\}$ holds the name of the last event to be taken at each configuration. It is $\bot$ in the initial configuration as no event has yet occurred. It allows us to refer to events in LTL formulas in order to state that they have just occurred. For instance, in the formula above, $(s, t, m, p) \models e_1 \wedge t_1 = 0$ (which reads: the configuration *satisfies* the formula) evaluates to $p = e_1 \wedge t(t_1) = 0$.

Given a flattened module instance $\mathcal{M}$, the transitions of its corresponding LTS are given as $\mathbf{T} = E_{id} \cup E\# \cup \{tick\}$. As explained above, for each event $e \in E$, we introduce a monotonicity breaking transition $e.id\#$. We thus define $E\# \triangleq \{e \in E_{id} \bullet e\#\}$. The *tick* transition represents one tick of a global clock. Explicit timers and event lower and upper time bounds are described with respect to this tick transition. We define the enabling condition of event $e \in E$ with fair index x and demonic index y as $e.en(x) \triangleq (\exists y \bullet e.grd(x, y)) \wedge e.l \leq e.c \leq e.u$, where $e.c$ evaluates to $c(e.id, x)$ in a configuration whose clock component is c. Thus an event is enabled in a configuration that satisfies its guard and where the event's implicit clock is between its lower and upper time bound.

The initial configuration is defined as $\pi_0 = (s_0, t_0, m_0, c_0, \bot, \bot)$, where s_0 and t_0 come from the abstract syntax of the TTM. m_0 and c_0 are given by:

$$\begin{aligned} m_0(t_i) &\equiv t_0(t_i) = 0 \\ c_0(e_i.id, x) &= \begin{cases} 0 & (s_0, t_0) \models (\exists y \bullet e_i.grd(x, y)) \\ -1 & (s_0, t_0) \not\models (\exists y \bullet e_i.grd(x, y)) \end{cases} \end{aligned}$$

for each $t_i \in T$ and $e_i \in E$. It is implicit in the above formula that $m_0(t_i)$ depends only on whether or not t_i is initially enabled (specified using the keyword **enabledinit** or **disabledinit**). If the keyword **enabledinit** is specified, $t_0(t_i) = 0$; otherwise, if the keyword **disabledinit** is specified, $t_0(t_i) = boundt(t_i) + 1$.

An execution σ of the LTS is an infinite sequence, alternating between configurations and transitions, written as $\pi_0 \xrightarrow{\tau_1} \pi_1 \xrightarrow{\tau_2} \pi_2 \rightarrow \cdots$ where $\tau_i \in \mathbf{T}$ and $\pi_i \in \Pi$. Below, we provide constraints on each one-step relation ($\pi \xrightarrow{e} \pi'$) in an execution. If an execution σ satisfies all these constraints then we call σ a *legal* execution. We let $\Sigma_{\mathcal{L}}$ denote the set of all legal executions of the labelled transition system $\mathcal{L}$. The set $\Sigma_{\mathcal{L}}$ provides a precise and complete definition of the behaviour of $\mathcal{L}$. If a state-formula q holds in a configuration π, then we write $\pi \vDash q$. In some formulas, such as guards, all the components of a configuration are not necessary. We express this by dropping some components of the configuration on the left of the double turnstile ($\models$), as in $(s_0, t_0) \models e.grd(x, y)$. Given a temporal logic property φ and an LTS $\mathcal{L}$, we write $\mathcal{L} \vDash \varphi$ iff $\forall \sigma \in \Sigma_{\mathcal{L}} \bullet \sigma \vDash \varphi$. The three possible transition steps are:

$$(s,t,m,c,\bot,p) \overset{(e,x)\#}{\rightarrow} (s,t,m',c,(e,x),p) \tag{18}$$

$$(s,t,m,c,(e,x),p) \overset{(e,x)}{\rightarrow} (s',t',m',c',\bot,(e,x)) \tag{19}$$

$$(s,t,m,c,\bot,p) \overset{tick}{\rightarrow} (s,t',m',c',\bot,tick) \tag{20}$$

Each of the above transitions has side conditions which we now enumerate.

6.2.1 Taking $e\#$

The monotonicity breaking transition $(e,x)\#$, specified in Equation 18 (p35), is taken only if $(s,t,c) \vDash e.en$ and the x-component of the configuration is $\bot$. For each $t \in T$, $m'(t) \equiv t \notin e.start \wedge m(t)$. This ensures that, for timer t, just before it is (re)started, $m(t) \equiv$ false. It is set back to true by the immediately following event, e, and it remains true as long as t is not restarted and has not reached its upper bound. Transition $e\#$ modifies only m and x in the configuration, and thus maintains the truth of $(s,t,c) \vDash e.en(x)$.

6.2.2 Taking e

The transition $e(x)$, specified in Equation 19 (p35), is taken only if $(s,t,c) \vDash e.en(x)$ and the x-component of the configuration is e. The component s' of the next configuration in an execution is determined nondeterministically by $e.action(x,y)$, which is a relation rather than a function. This means that any next configuration that satisfies the relation can be part of a valid execution, i.e., s' is only constrained by $(s,t,s') \in e.action(x,y)$. The other components are constrained deterministically. The following function tables specify the updates to m, t and c upon occurrence of transition e.

For each timer $t_i \in T$		$m'(t_i)$	$t'(t_i)$
$t_i \in e.start$	$t_i \in e.stop$	*impossible*	
	$t_i \notin e.stop$	true	0
$t_i \notin e.start$	$t_i \in e.stop$	false	$boundt(t_i)+1$
	$t_i \notin e.stop$	$m(t_i)$	$t(t_i)$

For each event $e_i \in E$, $x \in e_i.f\_ind \rightarrow$ VALUE		$c'(e_i.id)$
$(s',t') \nvDash (\exists y \bullet e_i.grd(x,y))$		-1
$(s',t') \vDash (\exists y \bullet e_i.grd(x,y))$	$(s,t) \vDash (\exists y \bullet e_i.grd(x,y)) \wedge \neg e_i = e$	$c(e_i.id,x)$
	$(s,t) \nvDash (\exists y \bullet e_i.grd(x,y)) \vee e_i = e$	0

In the above, we start and stop the implicit clock of e_i as a consequence of executing e, according to whether $e_i.grd$ becomes true, is false (i.e., becomes or remains false) or

remains true. Since event e_i becomes enabled $e_i.l$ units after its guard becomes true, this allows us to know when to consider e_i as enabled, i.e., ready to be taken. As a special case, the implicit clock of event e (under consideration) is restarted when $e.grd$ remains true.

6.2.3 Taking *tick*

The tick transition, specified in Equation 20 (p35), is taken only if $\forall e \in E \bullet c(e.id, x) < e.u$ and the x-component of the configuration is $\perp$ (thus preventing *tick* from intervening between any $e\#$ and e pair). For any timer $t_i \in T$, the updates to t', m' and c' are:

$$t'(t_i) \;=\; (t(t_i) \downarrow boundt(t_i)) + 1$$
$$m'(t_i) \;\equiv\; \neg\, (t(t_i) = boundt(t_i){+}1)$$

For each event $e \in E$, $x \in e.f\_ind \rightarrow$ VALUE		$c'(e.id, x)$
$(s', t') \not\models (\exists y \bullet e.grd(x, y))$		-1
$(s', t') \models (\exists y \bullet e.grd(x, y))$	$(s, t) \not\models (\exists y \bullet e.grd(x, y))$	0
	$(s, t) \models (\exists y \bullet e.grd(x, y))$	$c(e.id, x) + 1$

Thus, *tick* increments timers and implicit clocks to their upper bounds. Transition *tick* also marks timers as non-monotonic when they reach their upper bound and reset clocks when the corresponding events are disabled.

6.2.4 Scheduling (including Indexed Events)

So far, we have made no mention of scheduling: we constrained executions so that the state changes in controlled ways, but a given execution may still make no progress. To make progress, we need to assume fairness. In the current implementation of TTM/PAT, the possible scheduling assumptions[12] on TTM events are restricted to the following four:

1. Spontaneous event. Even when it is enabled, the event might never be taken. This is assumed when no fairness keyword is given and the upper time bound is * or unspecified.

2. Just event scheduling (also known as weak fairness [17]). For any execution $\sigma \in \Sigma_{\mathcal{L}}$, if an event e eventually becomes continuously enabled, it has to occur infinitely many times, that is

$$\sigma \;\models\; (\forall x \;\bullet\; \Diamond\Box e.en(x) \;\rightarrow\; \Box\Diamond(\exists y \bullet e(x, y))) \;,$$

where x ranges over the fair indices of e and y over its demonic indices. This is where the distinction takes all its importance. The fairness assumption guarantees that $e(x, \_)$ is treated fairly for every single value of x. For example, if x was a process identifier, making it a fair index means that as long as it's active, each process is eventually given

[12] The scheduling assumptions are taken care of by the model-checking algorithms [17].

CPU time. In contrast, if x (still a process identifier) is treated as demonic index, it means that, as long as some process is ready, infinitely often a process (possibly always the same) will be given CPU time.

This is assumed when the keyword **just** is given next to the event and the upper time bound is * or unspecified. We use $e.en$ and not $e.grd$ in the fairness formula as the event can only be taken $e.l$ units after its guard became true.

3. Compassionate event scheduling (also known as strong fairness [17]). For any execution $\sigma \in \Sigma_{\mathcal{L}}$, if an event e becomes enabled infinitely many times, it has to occur infinitely many times, that is

$$\sigma \models (\forall x \bullet \Box\Diamond e.en(x) \rightarrow \Box\Diamond(\exists y \bullet e(x,y)) .$$

This is assumed when the keyword **compassionate** is given next to the event and the upper time bound is * or unspecified.

4. Real-time event scheduling. The (finite) upper time bound (u) of the event e is taken as a deadline: if the event's guard is true for u units of time, it has to occur within u units of after the guard becomes true or after the last occurrence of e. To achieve this effect, the event e is treated as just. Since *tick* will not occur as long as e is urgent (i.e., $e.c = e.u$), transition e will be forced to occur (unless some other event occurs and disables it).

To accurately model time, the *tick* transition is treated as compassionate in the LTS. This ensures that time progresses except in cases of Zeno-behaviors (discussed below). Spontaneous events cannot be used to establish liveness properties. Justice and compassion are strong enough assumptions to establish liveness properties but not real-time properties. Finally, real-time events can establish both liveness and real-time properties.

The above semantics allows for Zeno behaviours which occur when there are loops involving events with zero upper time bound (i.e., $e[0,0]$). We could ban $e[0,0]$ events altogether, but that would eliminate behaviours that are feasible and useful, e.g., where we describe a finite sequence of immediately urgent events (not in a loop). We can check that the system is non-Zeno by checking that the system satisfies $\Box\Diamond tick$.

The abstract TTM semantics provided above can be (and has been) implemented efficiently. For example, in the abstract semantics every event e is preceded by a breaker of monotonicity $e\#$. Most of the $e\#$ events do not change the configuration monotonicity component m and can thus be safely omitted from the reachability graph thereby shrinking it.

6.3 Semantics of Module Composition.

We have specified so far the semantics of individual TTM machines. However, the TTM notation includes a composition operator which was not discussed so far. The semantics

of systems comprising many machines is defined through flattening, i.e. by providing a single machine which, by definition, has the same semantics as the whole system.

6.3.1 Instantiation

.

When integrating modules in a system, they first have to be instantiated. This means that the interface variables of the module must be linked to variables of the system it will be a part of. For example if we had a *Phil* module (for philosopher) with two shared variables, *left_fork* and *right_fork*, and two global fork variables $f1$ and $f2$, we could instantiate them as:

instances *p1* = *Phil*(**share** *f1*, **share** *f2*) ; *p2* = *Phil*(**share** *f2*, **share** *f1*) **end**

This makes $f1$ the left fork of $p1$ and the right fork of $p1$, and makes $f2$ the left fork of $p2$ and the right fork of $p1$. Philosopher $p1$ is therefore equivalent to the module *Phil* with its references to *left_fork* substituted by $f1$ and its references to *right_fork* substituted by $f2$.

6.3.2 Composition (including Synchronous Events).

The composition $m1||m2$ is an associative and commutative function of two module instances. Before flattening the composition, we rename the local variables and the events so that the name of each local variable will be unique across the whole system. The renaming is done by prepending the name of the module instance to the name of all the instance's events and local variables. This is strictly a syntactic change and does not affect the semantics of the instances.

m1 / m2	in	out	share
in	in	out	share
out	out	–	–
share	share	–	share

Table 1: Mode of interface variables in compositions.

We then proceed to creating the composite machine. Its local variables will be the (disjoint) union of the local variables of the two instances. Its interface variables will be the (possibly non-disjoint) union of the interface variables of both instances with their mode (in, out, share) adjusted as shown by Table 1.

In the simplest case of composition, the set of the events of the composition is the union of the set of events of both machines. However, events from separate machines

can executed synchronously with each other. This can be specified by adding: 1) a *sync* section to one of the events; and 2) a *depends* clause to the enclosing module.

```
module M                          module A                          module B
depends                           depends                           interface
      a : A ; b : B                     b : B                       ...
...                               ...                               events
events                            events                              evt2 (i_b : fair T; j_b : T) [l_b, u_b] just
  evt0 (i_m : fair T; j_m : T)      evt1 (i_a : fair T; j_a : T)       when grd_b
  [l_m, u_m] just                   [l_a, u_a] just                    do z := exp_z
   sync a.evt1, b.evt2               sync b.evt2                       end
   when grd_m                        when grd_a                     end
   do x := exp_x                     do y := exp_y
   end                               end
end                               end
```

In the above illustration, we say module *M depends on* modules *A* and *B*, and the three events (qualified by their containing modules) *M.evt0*, *A.evt1*, and *B.evt2* form a *synchronous event set*.

Specifying *depends* clauses (at the module level) and *sync* clauses (at the event) level results in three dependency graphs (assuming that *MOD* denotes the set of declared modules, *EVT* the set of declared events qualified by their containing modules, e.g., *M.evt0*, and *VAR* the set of interface and local variables):

1. The *Module Dependency Graph* contains the set of vertices $V = MOD$, and the set of edges consisting of (m_1, m_2), where module m_1 depends on m_2.

   From the module dependency graph, we construct a number of *synchronous event sets*. For each connected component of the module dependency graph, from each event e declared in module m, where $m \in MOD$ and e declares a *sync* clause, we collect event e and all events under its *sync* clause (all collected events are qualified by names of their containing modules).

2. Each *Event Dependency Graph* contains the set of vertices $V = EVT$, and the set of edges consisting of (e_1, e_2), where e_1 and e_2 are in a synchronous event set and e_2 is declared under the *sync* clause of e_2.

   For each synchronous event set we construct an action graph. We denote all variables involved in actions of a synchronous event set s as VAR_s.

3. For each synchronous event set s, its corresponding *action graph* contains the set of vertices $V = VAR_s$, and the set of edges consisting of (v_1, v_2), where the computation of v_1's new value depends on that of v_2. There are two cases: 1) v_2 appears on the RHS of an equation where v_1' is the LHS (i.e., $v_1' = \ldots\ v_2\ \ldots$); and 2) v_2 appears on the RHS of an assignment where v_1 is the LHS (i.e., $v_1 := \ldots\ v_2\ \ldots$).

   We perform a topological sort on each action graph to calculate the order of variable assignments. From the calculated order of variables, we calculate an order of variable projections. The projection for each variable v is a pair (v, act), where *act*

is either an unconditional assignment (i.e., $v := exp$), or an conditional assignment (i.e., ***if*** b_1 ***then*** $v := exp_1$ ***elseif*** b_2 ***then*** $v := exp_2 \ldots$ ***else*** $\ldots$). The latter case is resulted from the fact that changes on v (either through assignments or the primed notation) occur inside nested if-statements.

The above three graphs must satisfy the following constraints (otherwise it is reported by the TTM tool as an error):

- Acyclic
- Each synchronous event set (which will end up forming a flattened event) must include at most one event from each module instance of the system.
- Each flattened (or compound) event has to assign at most one value to each variable.

The flattening of those events in the above example is done by

module *COMPOUND*
interface
...
events
  sync_evt $(i_m, i_a, i_b$: **fair** T; $j_m, j_a, j_b : T)$ $[l_m \uparrow l_a \uparrow l_b,\ u_m \downarrow u_a \downarrow u_b]$ **just**
   when $grd_m \wedge grd_a \wedge grd_b$
   do $x := exp_x$, $y := exp_y$, $z := exp_z$
   end
end

The flattened event has its lower time bound being the maximum of those of its source events, its upper time bound their minimum, its guard the conjunction of theirs, and its action constructed using the topological sort and calculations of variable projections as described above.

In m, u_a tells us that the event should be taken before u_a ticks after grd_a became true. However, in COMPOUND, the event should be taken before u_a (and before u_a and before u_b but that's beside the point) after $grd_m \wedge grd_a \wedge grd_b$ become true which may be much later than when grd_a becomes true. For instance, let's ignore grd_b for a moment (i.e. assume it remains true) and assume that grd_m becomes true at time 5 and grd_a becomes true at time 1. Furthermore, assume u_a is 3 and u_b and u_m are *. According to the information in module A (which has no access to module M), the $evt1$ (or its counterpart in a flatten event) should occur before time 4. However, the information in COMPOUND states that it should be between time 5 and 8. It makes it hard to understand module A on its own. Therefore, sync events are non compositional. We might remediate this by requiring each compound event to include only one event with a timing assumption.

6.3.3 Iterated Composition

. Iterated composition is the mechanism that allows us to compose a number of similar instances without specifying each individually. For example, in the case of a network of processes, we may want to specify the processes once and instantiate them many times with a different process identifier.

system = || *pid* : *PID* @ *Process*(**in** *pid*)

where PID is the set of process identifiers. It allows us to change the number of processes by just changing that set. In this case, if $PID = 1..3$, the above is equivalent to:

instances *p1* = *Process*(**in** 1) ; *p2* = *Process*(**in** 2) ; *p3* = *Process*(**in** 3) **end**
composition *system* = *p1* || *p2* || *p3* **end**