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 and Computer Science,
York University, Canada
{jackie, jonathan, simon}@cse.yorku.ca

Timed Transition Models (TTMs) are event-based descriptions for modelling, specifying, and verifying discrete real-time systems. An event can be spontaneous, fair, or timed with specified bounds. TTMs have a textual syntax, an operational semantics, and an automated tool supporting linear-time temporal logic. We extend TTMs and its tool with two novel modelling features for writing high-level specifications: indexed events and synchronous events. Indexed events allow for concise description of behaviour common to a set of actors. The indexing construct allows us to select a specific actor and to specify a temporal property for that actor. We use indexed events to validate the requirements of a train control system. Synchronous events allow developers to decompose simultaneous state updates into actions of separate events. To specify the intended data flow among synchronized actions, we use primed variables to reference the post-state (i.e., one resulted from taking the synchronized actions). The TTM tool automatically infers the data flow from synchronous events, and reports errors on inconsistencies due to circular data flow. We use synchronous events to validate part of the requirements of a nuclear shutdown system. In both case studies, we show how the new notation facilitates the formal validation of system requirements, and use the TTM tool to verify safety, liveness, and real-time properties.

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, 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 [1].

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 [6], 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. So far, TTMs were used to verify that a variety of implementations satisfy their specifications.

In this paper, we extend the TTM notation, semantics, and tool for two novel modelling features: indexed events and synchronous events. These constructs are suitable for writing high-level specification, 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 one of the platforms (denoted by the set *PLF*) eventually moves out, and into the exit block: $\square(loc[t] \in PLF \;\Rightarrow\; \lozenge move\_out(t))$. 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).

Synchronous events allow developers to decompose simultaneous state updates into actions of separate events. However, without a mechanism to reference the post-state values of monitored variables, we cannot properly model the joint actions of the environment and controller. For example, the synchronized action $m := exp \;||\; c := f\,(m)$ specifies that the new (or next-state) value of controlled variable c is computed on the basis of the old (or pre-state) value of monitored variable m (i.e., *exp*). To resolve this, we use primed variables on the RHS of assignments in event actions to denote post-state values. For example, the synchronized action $m := exp \;||\; c := f\,(m')$ specifies that the post-state value of c is now a function on the post-value value of m. Synchronous events, together with primed variables, are suitable for describing high-level specifications used in shutdown systems of nuclear reactors [11]. 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.

Contributions. To support indexed and synchronous events for validating requirements, we extend the semantics of TTM (Sec. 2), and we extend our tool accordingly. For synchronous events, our tool automatically infers the data flow, and reports on inconsistencies due to circular data flow. We conduct two realistic case studies: a train control system (Sec. 3) using indexed events, and a part of a nuclear shutdown system (Sec. 4) using synchronous events.

Resources. Complete details of the two case studies are included in an extended report [10], which also contains more case studies of cyber physical systems (i.e., a mutual exclusion protocol, and function blocks from the IEC 61131 Standard for programmable logic controllers) that can be specified using the new notations. Complete TTM listings of the case studies are available at: `https://wiki.eecs.yorku.ca/project/ttm/index_sync_evt`.

2 Semantics for Indexed and Synchronous Events

We extend the one-step operational semantics of TTMs reported in [6] to support both indexed events (Sec. 2.2) and synchronous events (Sec. 2.3). The extensions involve redefining: 1) the abstract syntax of events which affects the rules of transitions and scheduling; and 2) the rules of module compositions. We include the most relevant details to present these extensions, while the complete account of the new semantics is included in an extended report [10, Sec. 6].

2.1 Abstract Syntax: Introducing Fair and Demonic Event Indices

We define the abstract syntax of a TTM module instance $\mathscr{M}$ as a 5-tuple (V, s_0, T, t_0, E) where 1) V is a set of local or interface variables; 2) T is a set of timers; 3) E is a set of state-changing events; 4) $s_0 \in$ STATE is the initial state (STATE $\triangleq V \rightarrow$ VALUE); and 5) $t_0 \in$ TIMER is the initial timer assignment (TIMER $\triangleq T \rightarrow \mathbb{N}$). We define $type \in T \rightarrow \mathbb{P}(\mathbb{N})$ and $boundt \in T \rightarrow \mathbb{N}$ for querying about, respectively, the type and upper bound of each timer. For example, if timer t_1 is declared as $t_1 : 0..5$, then $boundt(t_1) = 5$ and $type(t_1) = \{0..6\}$. Timers count up to one beyond the specified bound, and remain unchanged until they are started again. The figure below presents the generic form of a TTM event, where $V = \{v_1, v_2, v_3, \cdots\}$ and $T = \{t_1, t_2, t_3, t_4, \cdots\}$.

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:

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

We use a 10-tuple $(id, f\_ind, d\_ind, l, u, fair, grd, start, stop, action)$ to define the abstract syntax of an event e. We write $e.id$ for its identifier. Sets $e.f\_ind$ and $e.d\_ind$ contain, respectively, fair and demonic indices that can be referenced in the event. Its fairness assumption (i.e., *e.fair*), as discussed in Sec. 2.2, filters out certain execution traces that will be considered in the model checking process. Its guard (i.e., *e.grd*) is a Boolean expression referencing state variables, timers, or its indices. An event e must be taken between its lower time bound (LTB) $e.l$ and upper time bound (UTB) $e.u$, while its guard $e.grd$ remains true. The event action involves simultaneous assignments to $v_1, v_2, \cdots$. We write $v_3 :: 1..4$ for a demonic (non-deterministic) assignment to v_3 from a finite range. Therefore, its state effect is a relation *e.action* on state variables and timers. On the RHS of an assignment $y := x$, the state variable x may be "primed" (x') or "unprimed". A primed variable refers to its value at the *next* state, or its current-state value if it is unprimed. The use of primed variables in expressions allows for more expressive descriptions of state changes, especially when combined with the use of synchronous events (Sec. 2.3).

2.2 Operational Semantics

Given a TTM module instance $\mathscr{M}$, an LTS (Labelled Transition System) is a 4-tuple $\mathscr{L} = (\Pi, \pi_0, \mathbf{T}, \to)$ 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) $\to \subseteq \Pi \times \mathbf{T} \times \Pi$ is a transition relation.

We define E_{id} as the set of event transition names, and E_{fair} as the set of transition name prefixes, excluding values of demonic indices (i.e., including values of fair indices): $E_{id} \triangleq \{e, m \mid e \in E \wedge m \in e.f\_ind \to \text{VALUE} \bullet (e.id, m)\}$. On the one hand, we use $e(x)$ to denote the (external) transition name of event e with x, the values of its fair indices. On the other hand, when referring to the occurrence of e, in an LTL formula for instance, we use $e(x,y)$ to include y, the values of its demonic indices; otherwise, values of demonic indices are 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), where:

- $s \in$ 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.
- $t \in$ TIMER is a timer valuation function. Event transitions may start, stop, and read timers. A *tick* transition representing a global clock changes the timers.
- $m \in T \to$ BOOL records the status of monotonicity of each timer. Suppose event e_1 starts t_1, then we may specify that a predicate p becomes true within 4 ticks after e_1's occurrence. However, other events might stop or restart t_1 before p is satisfied, making t_1 not in sync with the global clock. The expression $m(t_1)$ (monotonicity of timer t_1) holds in any state where t_1 is not stopped or reset.

• $c \in E_{id} \to \mathbb{N} \cup \{-1\}$ 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 ($c(e.id,x) \geq e.l$) and whether it is urgent ($c(e.id,x) = e.u$).

• $x \in E_{id} \cup \{\perp\}$ provides a sequencing mechanism: each transition e is immediately preceded by a transition $e\#$ to update the monotonicity record m.

• $p \in E_{id} \cup \{tick, \perp\}$ holds the name of the last event to be taken at each configuration. It is $\perp$ in the initial configuration. It allows us to refer to events in LTL formula, to state that they have just occurred.

We focus on components s and c that are affected the most by fair and demonic indices, whereas components t, m, and x, as to how the monotonicity status of timers is maintained, are less relevant and included in [10, Sec. 6].

Given a flattened module instance $\mathscr{M}$, transitions of its corresponding LTS are given as $\mathbf{T} = E_{id} \cup E\# \cup \{tick\}$, where $E\# \triangleq \{e \in E_{id} \bullet e\#\}$ is the set of monotonicity-breaking transitions as mentioned above. 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 when its guard is satisfied, and when its implicit clock is in-between its specified bounds: ($e.en(x) \triangleq (\exists y \bullet e.grd(x,y)) \wedge e.l \leq c(e.id,x) \leq e.u$).

The initial configuration is defined as $\pi_0 = (s_0, t_0, m_0, c_0, \perp, \perp)$, where s_0 and t_0 come from the abstract (Sec. 2.1). The value of each event e_i's implicit clock depends on its guard being satisfied initially. More precisely, $c_0(e_i.id,x)$ equals 0 (the clock starts) if $(s_0,t_0) \models (\exists y \bullet e_i.grd(x,y))$[1]; otherwise, it equals -1.

An execution σ of the LTS L is an infinite sequence $\pi_0 \xrightarrow{\tau_1} \pi_1 \xrightarrow{\tau_2} \pi_2 \to \cdots$, alternating between configurations $\pi_i \in \Pi$ and transitions $\tau_i \in \mathbf{T}$. 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. To characterize the complete behaviour of $\mathscr{L}$, we let $\Sigma_{\mathscr{L}}$ denote the set of all its legal executions. Given a temporal logic property φ and an LTS $\mathscr{L}$, we write $\mathscr{L} \models \varphi$ iff $\forall \sigma \in \Sigma_{\mathscr{L}} \bullet \sigma \models \varphi$. There are two possible transition steps (event $e(x)$ and $tick$):

$$(s,t,m,c,e(x),p) \xrightarrow{e(x)} (s',t',m',c',\perp,e(x)) \tag{1}$$

$$(s,t,m,c,\perp,p) \xrightarrow{tick} (s,t',m',c',\perp,tick) \tag{2}$$

Taking *e* The transition $e(x)$ specified in Eq. 1 is taken only if the x-component of the configuration is e (meaning that $e\#$ was just taken, so e is the only event allowed to be taken) and $(s,t,c) \models e.en(x)$. The component s' of the *next* configuration in an execution is determined non-deterministically by $e.action(x,y)$, which is a relation as demonic indices or assignments may be used. Consequently, 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 following function tables specify the updates to c upon occurrence of transition $e(x)$.

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

[1]If a state-formula q holds in a configuration π, then we write $\pi \models q$. For formulas such as guards which do not depend on all components of a configuration, we drop some of its components on the left of $\models$, as in $(s_0,t_0) \models e.grd(x,y)$.

We start and stop the implicit clock of e_i as a consequence of executing e, according to whether $e_i.grd$ just becomes or remains false (1st row), remains true (2nd row), or just becomes true (3rd row). Event e_i is ready to be taken if it becomes enabled $e_i.l$ units after its guard becomes true.

Taking *tick* The tick transition specified in Eq. 2 is taken only if the x-component of the configuration is $\bot$ (thus preventing $tick$ from intervening between any $e\#$ and e pair) and if $\forall e \in E \bullet c(e.id, x) < e.u$.

For each event $e \in E$, $x \in e.f\_ind \to$ 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 towards their upper bounds.

Scheduling So far, we have constrained executions so that the state changes in controlled ways. However, to ensure that a given execution does not stop making progress, we need to assume fairness. The current TTM tool supports four possible scheduling assumptions.

1. Spontaneous event. When no fairness keyword is given, and the UTB is given as * or unspecified, then even when the event is enabled, it might never be taken.

2. Just event scheduling (a.k.a. weak fairness [9]). This is assumed when the event is declared with the keyword `just` and when the upper time bound is * or unspecified. For any execution $\sigma \in \Sigma_{\mathscr{L}}$, if an event e eventually becomes continuously enabled, then it occurs infinitely many times: $\sigma \models (\forall x \bullet \Diamond\Box e.en(x) \to \Box\Diamond(\exists y \bullet e(x,y)))$, where x ranges over e's fair indices and y its demonic indices.

This highlights the key distinction between fair and demonic indices. The fairness assumption guarantees that $e(x,\_)$ is treated fairly for every single value of x. For example, if x is a process identifier, making it a fair index means that as long as it is active, each process is eventually given CPU time. In contrast, if x is treated as a demonic index, then it is possible that infinitely often the same process will be given CPU time.

3. Compassionate event scheduling (a.k.a. strong fairness [9]). This is assumed when the event is declared with the keyword `compassionate` and when the upper time bound is * or unspecified. For any execution $\sigma \in \Sigma_{\mathscr{L}}$, if an event e becomes enabled infinitely many times, it has to occur infinitely many times. More precisely: $\sigma \models (\forall x \bullet \Box\Diamond e.en(x) \to \Box\Diamond(\exists y \bullet e(x,y))$.

4. Real-time event scheduling. The finite UTB $e.u$ of the event e is taken as a deadline: it has to occur within u units of time after $e.grd$ 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).

2.3 Semantics of Module Composition

So far we have specified the semantics of individual module instances. However, the TTM notation includes a composition. The semantics of systems comprising many instances is defined through flattening, i.e. by providing a single instance which, by definition, has the same semantics as the whole system.

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

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

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 $f2$.

Composition The composition $m1||m2$ is an associative and commutative function on two module instances. To flatten the composition, we rename the local variables and events (by prepending the module instance name) so that they are system-wide unique. We then proceed to create the composite instance. Its local variables are the (disjoint) union of the local variables of the two instances. Its interface variables are the (possibly non-disjoint) union of the interface variables of both instances with their mode (`in`, `out`, `share`) adjusted properly [10, Table 1, p. 38] (e.g., variable `in x` in $m1$ and variable `out x` in $m2$ result in an `out` variable in the composite instance).

The simplest case of composition results in the union of the set of events of both instances. However, events from separate instances can be executed synchronously. This can be specified using the notation of synchronous events. As an illustration, consider a case where the plant and controller act synchronously.

```
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
```

We say module *CONTROLLER* depends on module *PLANT*. At the module level (e.g., *CTRL*), 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 next-state values. For example, the *respond* event uses the next-state value of the input variable *x* (i.e., *x'*) to compute the next-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 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 controlled variable *b*.

Specifying *depends* clauses (at the module level) and *sync* clauses (at the event level) results in one or more compound events whose actions are composed of those involved in the synchronization. We discuss the process of merging event actions below. For how event time bounds and fairness assumptions are merged in synchronization, refer to [10, p. 40].

The use of synchronous events results in three kinds of dependency graphs[2].

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.

In each connected component of the module dependency graph, we construct a *synchronous event set* (e.g., $\{PLANT.generate, CONTROLLER.respond\}$) by including each event *e*, where *e* declares a *sync* clause, and all events under *e*'s *sync* clause.

[2]Assume that *MOD* denotes the set of declared modules, *EVT* the set of declared events qualified by their containing modules, e.g., *PLANT.generate*, and *VAR* the set of interface and local variables

2. An *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 the same synchronous event set and e_2 is declared under the *sync* clause of e_1.

3. An *Action Graph* is constructed from each synchronous event set. We write VAR_s to denote variables that are involved in actions of events in a synchronous event set s. 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 (or next state) value depends on that of v_2. There are two cases to consider: 1) in an equation where v_2 appears on the RHS and v_1' on the LHS (i.e., v_1' = ... v_2 ...); and 2) in an assignment where v_2 appears on the RHS and v_1 on the LHS (i.e., v_1 := ... v_2 ...).

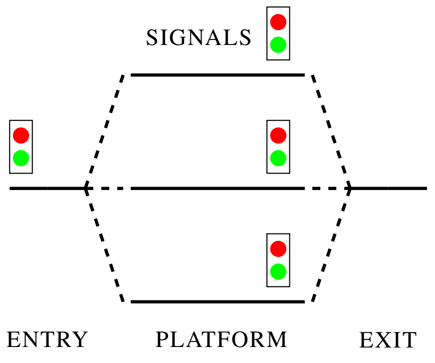
We perform a topological sort on each action graph to calculate the order of variable assignments, from which we calculate a sequence of variable projections. The *projection* for each variable v is a pair (v, act), where *act* is either an unconditional assignment (i.e., v := *exp*), or an conditional assignment (i.e., ***if*** b_1 ***then*** v := exp_1 ***elseif*** b_2 ***then*** v := exp_2 ... ***else*** ...). The latter case is resulted from the fact that changes on v (either through assignments or the primed notation) occur inside nested if-statements. Finally, the produced sequence of variable projections is adopted as the action of the compound event.

To ensure consistency, the TTM tool reports an error when, e.g., one of the above graphs contains a cycle, or a flattened (or compound) event assigns multiples values to the same variable.

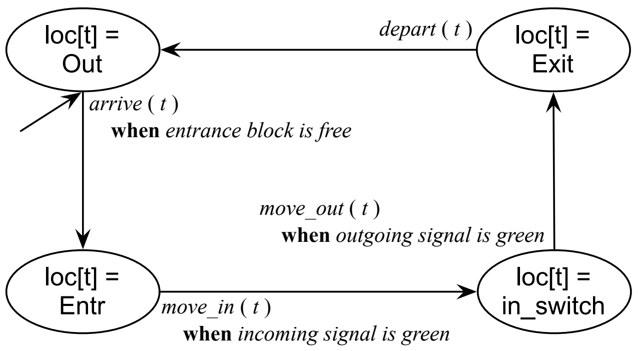
Iterated Composition. Iterated composition allows us to compose an indexed set of similar instances. For example, in the case of a network of processes, we may specify the common process behaviour as a module once, and instantiate them from the set *PID* of process identifiers: $system = \, || \; pid : PID \; @ \; Process(\textbf{in} \; pid)$.

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 event 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.



(a) Topology


loc[t] = Out
loc[t] = Exit
loc[t] = Entr
loc[t] = in_switch
depart (t)
arrive (t) when entrance block is free
move_out (t) when outgoing signal is green
move_in (t) when incoming signal is green

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

Figure 1: A Train Control System

Fig. 1a shows the topology of the train control system [3]. There is an entry block (*Entr*) and an exit block (*Exit*) on both ends of the station. Between the entry and exit blocks is a set *PLF* 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, depending on the availability of platforms. On each platform $p \in PLF$, there is a signal $osgn[p]$ regulating the outgoing train, depending on the availability of the exit block. Fig. 1b illustrate the common behaviour of all trains. Each train is initially travelling outside the station. The train may first arrive at the entry block, 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 the station.

Trains must never collide in the train station. Also, once a train arrives, it should be eventually scheduled to depart from the station.

$$(\forall t1, t2 : TRAIN \bullet \ (\ t1 \neq t2 \wedge loc[t1] \neq Out \wedge loc[t2] \neq Out\ ) \Rightarrow \ loc[t1] \neq loc[t2]) \tag{3}$$

$$\Box(\ loc[t] = Entr \Rightarrow \Diamond(loc[t] = Out)\ ) \tag{4}$$

We consider two versions of TTM that satisfy both Eq. 3 and 4. Fig. 2a presents the TTM interface of an abstract version, where monitored and controlled variables are separated. 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 Fig. 3a) and of signal control (e.g., event *ctrl_platform_signal* in Fig. 4a). On the other hand, Fig. 2b presents the interface of a refined version, which distinguishes between one monitored variable (i.e., *occ* for the set of occupied platforms) and three controlled variables (i.e, *isgn* for an incoming train, *in_switch* for platform currently connected to the entrance block, and *osgn* for outgoing trains). Consistently, the behaviour of the controller and that of the trains are factored in separate events and placed in separate modules. The monitored variable (with modifier ***in***) is owned by the *STATION* module and read-only for the *CONTROLLER* module.

module *STATION*
interface
  loc : **out ARRAY**[*OPT_BLOCK*]
  isgn : **out BOOL**
  osgn : **out ARRAY**[**BOOL**]
  in_switch : **out** *BLOCK*

(a) Abstract

module *STATION*
interface
  occ : **out ARRAY**[**BOOL**]
  isgn : **in BOOL**
  osgn : **in ARRAY**[**BOOL**]
  in_switch : **in** *BLOCK*
local
  loc : **ARRAY**[*OPT_BLOCK*]

share initialization
  qe : <*Queue*>
end
module *CONTROLLER*
interface
  occ : **in ARRAY**[**BOOL**]
  isgn : **out BOOL**
  osgn : **out ARRAY**[**BOOL**]
  in_switch : **out** *BLOCK*

(b) Refined: Separate Station & Controller Events

Figure 2: Train Control System in TTM: Interfaces

The refined version of TTM changes the representation of the data used by control events. In the abstract version (Fig. 2a), 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 Fig. 3a) 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 Fig. 4a) 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 (Fig. 2b), 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 (Fig. 4b). Moreover, train events in the environment (e.g., Fig. 3b) 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 that a block is occupied if and only if it corresponds to the location of some train.

```
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 3: Train Control System in TTM: the *move_out* Event in Module *STATION*

The two versions of TTMs are different in scheduling the green signals that control the passage from the platforms to the exit block. While the abstract model is non-deterministic about the order in which trains gain access to the exit block, the concrete model specifies the order uniquely. The signals are controlled by event *ctrl_platform_signal*. In the abstract version (Fig. 4a), 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 4, 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. Consequently, in the refined version (Fig. 4b), we use a C# FIFO *Queue*[3] to specify the order of train departure. The reduced non-determinism allows us to remove the fair index on trains and weaken the fairness assumption (i.e., the event becomes *just*).

```
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 4: Train Control System in TTM: Controller Events

4 Example: Tabular Requirement of a Nuclear Shutdown System

We illustrate the use of synchronous events on parts of the software requirements of a shutdown system for the Darlington Nuclear Generating Station. We present two versions of the system. The first version presents a high-level requirements [11] where the controller responds instantaneously to environment changes. We synchronize the environment and controller events to model such instantaneity, and check

[3]Using a C# data object, implementation details of operations such as *Enqueue* are all encapsulated, resulting in a model simpler than one using a native TTM array.

it via an invariant property. The refined version illustrates how the response allowance [12] can be incorporated as event time bounds (i.e, the controller responds fast enough to environment changes). We 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) [4]. Figure 5 exemplifies tabular requirements for two units: Neutron OverPower (NOP) Parameter Trip (Figure 5a) and Sensor Trips (Figure 5b). 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 input conditions) and the disjointness (i.e., no input conditions satisfied simultaneously) of our requirements [4].

	Result
Condition	*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

	Result
Condition	*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 sensor i, $i \in 0..17$ (monitoring *calibrated_nop_signal*[*i*])

Figure 5: Tabular Requirement for the Neutron Overpower (NOP) Trip Unit

The NOP Parameter Trip unit (the NOP controller) depends on 18 instances of the Sensor Trip units (the NOP sensors). There are two monitored variables for each NOP sensor i: (1) a floating-point calibrated NOP signal value *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 can be one of the 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. The hysteresis region $(f\_NOPsp - k\_NOPhys, f\_NOPsp)$ is an open interval. When the monitored signal falls within this region, then the new value of *f_NOPsentrip* remains as that in the previous state, denoted as $f\_NOPsentrip_{-1}$. 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 6), to ensure that the system satisfies the tabular specification in Figure 5, the NOP controller must have completed its very first response (denote as predicate $\neg init\_response$).

The requirements model in Figure 5 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 [12]. 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, 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[4] 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.

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 6 illustrates the structure of synchronization. The dashed box in Figure 6 indicates the set of synchronized modules instances: plant p, controller *nop*, and 18 sensors *sensor_i* ($i \in 0..17$).

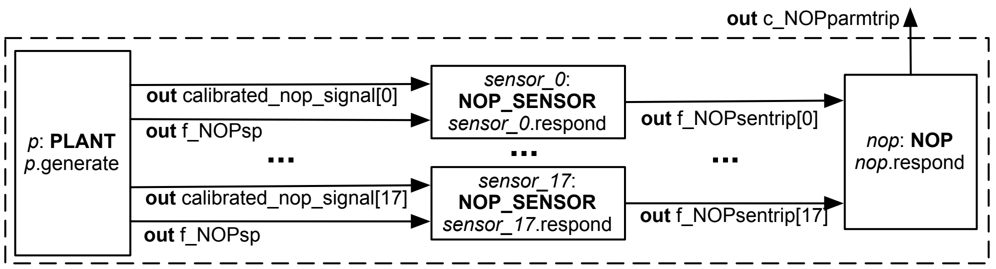


Figure 6: Neutron Overpower (NOP): Abstract Version – Synchronized Plant and Controller

Figure 8 (p. 95) presents the complete[5] 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 performed via the demonic assignment *calibrated_nop_signals* :: ***ARRAY***[*cal_nop*](*N*) (Lines 5 – 6). 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

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

[5]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.

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 5a and Figure 5b, 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 dependent sensors have written to the shared array *f_NOPsentrip*, the NOP controller responds instantaneously.

$$\Box\left(\begin{array}{ll} & (\ \exists i : 0\,..\,N \bullet f\_NOPsentrip[i] = e\_Trip\ ) \Rightarrow c\_NOPparmtrip = e\_Trip \\ \wedge & (\ \forall i : 0\,..\,N \bullet f\_NOPsentrip[i] = e\_NotTrip\ ) \Rightarrow c\_NOPparmtrip = e\_NotTrip \end{array}\right) \tag{5}$$

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*.

$$\Box\left(\begin{array}{l} \left(\begin{array}{ll} & \neg\ init\_response \\ \wedge & f\_NOPsp = k\_NOPLPsp \\ \wedge & k\_NOPLPsp \leq calibrated\_nop\_signal[0] \leq k\_CalNOPHiLimit \end{array}\right) \\ \qquad \Rightarrow c\_NOPparmtrip = e\_Trip \end{array}\right) \tag{6}$$

However, the satisfaction of Equation 6 is an idealized behaviour without the realistic concern of some allowance on the controller's response [12]. 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 8 by decoupling actions of the controller[6] and its plant. Figure 7 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.

The resulting system would fail to satisfy Equation 6, as we introduce some allowance on the response time (termed *response allowance* in [12]) of the NOP controller to environment changes. On

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

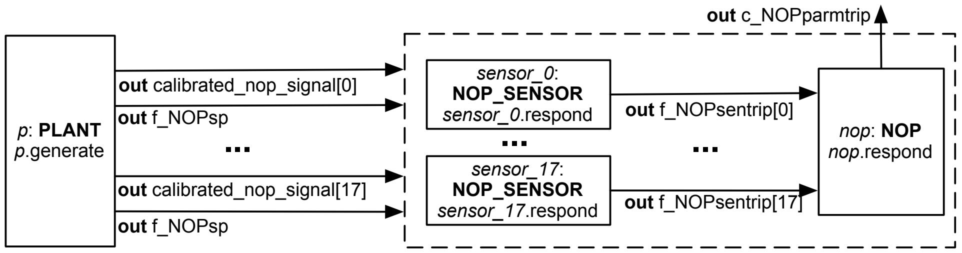

out c_NOPparmtrip
p: PLANT
p.generate
out calibrated_nop_signal[0]
out f_NOPsp
out calibrated_nop_signal[17]
out f_NOPsp
sensor_0: NOP_SENSOR sensor_0.respond
sensor_17: NOP_SENSOR sensor_17.respond
out f_NOPsentrip[0]
out f_NOPsentrip[17]
nop: NOP
nop.respond

Figure 7: Neutron Overpower (NOP): Refined Version – Separate Plant and Controller

the other hand, as we still consider the controller's response actions, once initiated, take effect instantaneously, the resulting system should still satisfy Equation 5.

We apply the following changes to produce the refined TTM (Figure 8). First, in module *PLANT*, we revise time bounds of the *generate* event to $[2, *]$, which encodes the assumption that the controller (whose *respond* event has time bounds $[1, 1]$) responds fast enough to the environment changes. Second, 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). Third, 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*. Fourth, 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:

$$\Box\left(\begin{pmatrix} & f\_NOPsp = k\_NOPLPsp \\ \wedge & k\_NOPLPsp \le calibrated\_nop\_signal[0] \le k\_CalNOPHiLimit \\ \wedge & t = 0 \end{pmatrix} \atop \Rightarrow mono(t)\ \mathbf{U}\ (\ c\_NOPparmtrip = e\_Trip \wedge t < 2\ )\right) \qquad (7)$$

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.

5 Discussion

Our new TTM notations facilitate the formal validation of cyber-physical system requirements. In the train control system (Sec. 3), the indexing construct allows us to select a specific actor (e.g., a train, a process, etc.) and specify a temporal property for that actor. Synchronous events, together with primed variables, allow us to check (real-time) response properties of the tabular requirements of a nuclear shutdown system (Sec. 4).

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) [7] and its tool [2] support multi-way synchronization by matching event names 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.) [8] allow events to be attached with state updates. However, their semantics and tool support do not allow events that are attached with updates to be synchronized. The UPPAAL model checker and its language of timed automata [5] 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.

For indexed events, the verification tool support for both conventional CSP [7] and UPPAAL [5] does not allow for fairness assumptions. For UPAAL, it is likely to manually construct an observer, but this is likely to result in convoluted encoding in larger systems and thus is prone to errors. On the other hand, the PAT tool allows users to choose fairness assumptions at the event, process, or global level [9] for verifying the un-timed CSP# and stateful timed CSP [8]. However, our notion of indexed events are of finer-grained for imposing fairness assumptions, as we allow the declaration of event indices as fair.

References

[1] Patricia Derler, Edward A. Lee & Alberto Sangiovanni-Vincentelli (2012): *Modeling Cyber-Physical Systems*. Proceedings of the IEEE (special issue on CPS) 100(1), pp. 13 – 28. Available at `http://dx.doi.org/10.1109/JPROC.2011.2160929`.

[2] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov & AndrewW. Roscoe (2014): *FDR3 – A Modern Refinement Checker for CSP*. In: TACAS, LNCS 8413, Springer, pp. 187–201. Available at `http://dx.doi.org/10.1007/978-3-642-54862-8_13`.

[3] Simon Hudon & ThaiSon Hoang (2013): *Systems Design Guided by Progress Concerns*. In: Integrated Formal Methods, LNCS 7940, Springer, pp. 16–30. Available at `http://dx.doi.org/10.1007/978-3-642-38613-8_2`.

[4] Ryszard Janicki, DavidLorge Parnas & Jeffery Zucker (1997): *Tabular Representations in Relational Documents*. In: Relational Methods in Computer Science, Advances in Computing Sciences, Springer Vienna, pp. 184–196. Available at `http://dx.doi.org/10.1007/978-3-7091-6510-2_12`.

[5] Kim G. Larsen, Paul Pettersson & Wang Yi (1997): *UPPAAL in a Nutshell*. International Journal on Software Tools for Technology Transfer 1(1–2), pp. 134–152. Available at `http://dx.doi.org/10.1007/s100090050010`.

[6] JonathanS. Ostroff, Chen-Wei Wang, Simon Hudon, Yang Liu & Jun Sun (2014): *TTM/PAT: Specifying and Verifying Timed Transition Models*. In: FTSCS, Communications in Computer and Information Science 419, Springer, pp. 107–124. Available at `http://dx.doi.org/10.1007/978-3-319-05416-2_8`.

[7] A.W. Roscoe (2010): *Understanding Concurrent Systems*, 1st edition. Springer. Available at `http://dx.doi.org/10.1007/978-1-84882-258-0`.

[8] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi & Étienne André (2013): *Modeling and verifying hierarchical real-time systems using stateful timed CSP*. ACM Trans. Softw. Eng. Methodol. 22(1), pp. 3:1–3:29. Available at `http://dx.doi.org/10.1145/2430536.2430537`.

[9] Jun Sun, Yang Liu, Jin Song Dong & Jun Pang (2009): *PAT: Towards Flexible Verification under Fairness*. In: CAV, LNCS 5643, pp. 709 – 714. Available at `http://dx.doi.org/10.1007/978-3-642-02658-4_59`.

[10] C.-W. Wang, J. S. Ostroff & S. Hudon (2014): *Using Indexed and Synchronous Events to Model and Validate Cyber-Physical Systems*. Tech Report EECS-2014-03, York University.

[11] A. Wassyng & M. Lawford (2006): *Software tools for safety-critical software development*. STTT 8(4-5), pp. 337–354. Available at `http://dx.doi.org/10.1007/s10009-005-0209-6`.

[12] A. Wassyng, M. Lawford & X. Hu (2005): *Timing Tolerances in Safety-Critical Software*. In: FM, pp. 157–172. Available at `http://dx.doi.org/10.1007/11526841_12`.

```
module PLANT      /* Template for Nuclear Reactor */
  interface
    f_NOPsp : out INT = k_NOPLPsp
    calibrated_nop_signal : out ARRAY[cal_nop](NUM_SENSORS) = [k_CalNOPLoLimit (NUM_SENSORS)]
  events generate[1, 1]
          do calibrated_nop_signal :: ARRAY[cal_nop](NUM_SENSORS), f_NOPsp := k_NOPLPsp end
end
module NOP      /* Template for Neutron Overpower Controller */
  depends
    env : PLANT
    sensor_0 : NOP_SENSOR
  interface
    f_NOPsentrip : share ARRAY[y_trip](NUM_SENSORS)      /* shared, but read only */
    c_NOPparmtrip : out y_trip = e_Trip
  local after_init_response : BOOL = false
  events
    respond[1, 1] sync env.generate, sensor_0.respond as respond
    do
      after_init_response := true,
      if (|| j: 0..(NUM_SENSORS−1) @ f_NOPsentrip'[j] == e_Trip) then
        c_NOPparmtrip := e_Trip
      elseif (&& k: 0..(NUM_SENSORS−1) @ f_NOPsentrip'[k] == e_NotTrip) then
        c_NOPparmtrip := e_NotTrip
      else skip fi
    end
end
module NOP_SENSOR      /* Template for Sensors */
  interface
    f_NOPsp : in INT
    calibrated_nop_signal_i : in cal_nop
    f_NOPsentrip_i : share y_trip      /* shared, but write only */
  local f_NOPsentrip_i_old : y_trip = e_Trip
  events
    respond[1, 1]
    do
      f_NOPsentrip_i_old' = f_NOPsentrip_i,
      if f_NOPsp' <= calibrated_nop_signal_i' then
        f_NOPsentrip_i := e_Trip
      elseif (f_NOPsp' − k_NOPhys < calibrated_nop_signal_i') && (calibrated_nop_signal_i' < f_NOPsp') then
        f_NOPsentrip_i := f_NOPsentrip_i_old
      elseif calibrated_nop_signal_i' <= f_NOPsp' − k_NOPhys then
        f_NOPsentrip_i := e_NotTrip
      else skip fi
    end
end
instances
  env = PLANT (out f_NOPsp, out calibrated_nop_signal)
  sensor_0 = NOP_SENSOR(in f_NOPsp, in calibrated_nop_signal[0], share f_NOPsentrip[0])
  nop = NOP(share f_NOPsentrip, out c_NOPparmtrip) with env := env, sensor_0 := sensor_0 end
  sys ::= env || sensor_0 || nop      /* named synchronous instance */
end
composition system = sys end
```

Figure 8: Requirement of NOP in TTM: Synchronized Plant and Controller