A Timed B Method for Modelling Real Time Reactive Systems

Miloud RACHED, Jean-Paul BODEVEIX, Mamoun FILALI, Odile NASR

IRIT, Paul Sabatier University
118 Route de Narbonne, 31062 Toulouse FRANCE
{rached, bodeveix, filali, nasr}@irit.fr

Abstract. The purpose of this paper is to develop a timed B method for modelling real time reactive systems, relying on the extension of the MITL logic into EMITL (Event Metric Interval Temporal Logic). This method allows to specify and check functional and temporal properties of a system. For that, we have extended the property and the substitution languages of B. In the former we introduce a variant of a timed logic incorporating events and high level predicates, in order to insert EMITL formulas. In the latter we add a new substitution to express time progression.

1 Introduction

We present here a timed extension to the B method [5, 1]. It allows to specify and check functional and temporal properties for real time reactive systems. We introduce a variant of a timed logic incorporating events and high level predicates (during, since, period, ...), in order to insert the properties expressed by EMITL formulas (Event Metric Interval Temporal Logic) in the B method. These formulas express temporal properties on the environment, or timing constraints on the controller reactions. For instance, operators during, since and period occur in an operation precondition or an operation guard. We have also added a new substitution wait to B generalized substitutions. It allows to specify in a non-deterministic way the progression of time, when no state is modified and no event is enabled. Note that these operators model time without clocks.

The remainder of the paper is organized as follows. Section 2 introduces real time reactive systems. In section 3, the MITL timed logic and its EMITL extension needed to formalize timing constraints are presented. In section 4, we give an overview of the B method and we show how to specify time in B. In section 5, we propose the syntax and the semantics of a B timed extension. In section 6, we present the abstract timed machine allowing to specify real time systems and the semantics of this machine. The discussion and the conclusion are presented respectively in sections 7 and 8.

2 Analysis of real time reactive systems

A usual framework of real time systems is that where a controller and an environment interact through sensors and actuators [20]. In such a system, data flows are represented by the figure 1.
The controller can check the environment state by reading the sensors and it can modify this state by assigning new values to the actuators. We consider real time reactive systems where the controller must react to the events received from the environment. A received event is called uncontrollable or input event, and the reaction of the controller is called controllable (command) or exit event.

We can represent a real time reactive system using control and data flow diagrams, where input event flows are indicated by dashed lines and output command flows by solid lines (figure 2).

In general, a controller must react to the reception of environment events before a fixed delay or a given date. This reaction is credited only with a small temporal margin because the collected data have precise validity durations, and it is needed to enable or disable actions quickly and to send punctually answers or commands. A real time system can be represented by [10]:

- A cyclic generator which periodically reads the environment state by sampling, does computations, prepares actions and sends commands. Thus in this case, we must control the logical time.
- A reactive system which answers instantaneously the events coming from the environment according to the dynamics of this one. For example, we rely on event occurrence dates using clocks, or order/causality relations between these events. Therefore, control actions are done according to their durations and their deadlines. Here, it is a question of controlling the physical time.

**Timing constraints** In a real time reactive system, we distinguish two types of timing constraints, the environment constraints and the properties which the system must ensure (reaction constraints).

- Environment constraints: we suppose that the environment verifies some temporal properties necessary to the satisfaction of the controller timing
constraints. They are composed mainly of periodicity constraints and event interarrival times.

- Reaction constraints: the controller must react to any environment event detected by a sensor. The result of this reaction is the release of an event allowing to modify the state of an actuator. The validity of the controller reaction does not depend only on the computations accuracy, but also on their production time.

3 Property language

To take into account event and state properties of a system, we have extended the MITL logic to EMITL (Event Metric Interval Temporal Logic). This logic plays a very important role in the design of our timed B method.

3.1 The MITL temporal timed logic

The MITL logic (Metric Interval Temporal Logic) [4] is a timed extension of the linear temporal logic. It allows to express the timed behavior of real time reactive systems. Here, we have chosen this logic to express the order and the metric (duration) between two events of a reactive system. Note that in MITL, time progression is continuous, and clocks are not mentioned.

MITL syntax MITL formulas are built using atomic propositions on a set of state variables. The logical operator Until is indexed by an interval I which has one of the following forms: [a, b], [a, b], [a, ∞), (a, b], (a, b), (a, ∞) where a < b for a, b ∈ ℝ⁺ (ℝ⁺ nonnegative real numbers).

Lower and upper bounds of I are denoted respectively by l(I) and r(I). The dimension of this interval is defined by |I| = r(I) - l(I) which can be infinite if I is unbounded. The interval I should not be singular (reduced to a single point), i.e. of the form [a, a], so that the satisfiability is decidable [4].

MITL formulas are defined inductively by the following grammar on a set of propositions P:

φ ::= T | p | ¬φ | φ ∧ φ | φ U I φ

where I is a non-singular interval and p is a proposition of P.

MITL semantics MITL formulas are interpreted on timed executions of the form τ = (σ, T) where σ is a state sequence σ = σ₀σ₁σ₂... and T an interval sequence T = I₀I₁I₂... such that:

- for any i, the intervals Iᵢ and Iᵢ₊₁ are adjacent and disjoint.
- any time t ∈ ℝ⁺ belongs to an interval Iᵢ.

A timed execution (σ, T) expresses that in the interval Iᵢ the system is in the state σᵢ. For a MITL formula φ and a timed execution sequence τ = (σ, T), the satisfaction relation, τ satisfies φ denoted τ ⊧ φ, is semantically defined by:

- τ ⊧ φ iff (∑₀, 0) ⊧ φ.
\[ (\tau, t) \models \top. \]
\[ (\tau, t) \models p \text{ iff for some } k, t \in I_k, \text{ and } p \in \rho(\sigma_k), \text{ where } \rho \text{ is the valuation function associating to each state a set of propositions}. \]
\[ (\tau, t) \models \neg \varphi \text{ iff } (\tau, t) \not\models \varphi. \]
\[ (\tau, t) \models \varphi_1 \land \varphi_2 \text{ iff } (\tau, t) \models \varphi_1 \text{ and } (\tau, t) \models \varphi_2. \]
\[ (\tau, t) \models \varphi_1 U_I \varphi_2 \text{ iff for some } t' \in I + t, (\tau, t') \models \varphi_2, \text{ and for all } t'' \in [t, t'), \]
\[ (\tau, t'') \models \varphi_1. \]

**Logical operators** Unary operators \( \Diamond_I \) (eventually) and \( \Box_I \) (always) can be defined respectively using the operator \( U_I \) by: \( \Diamond_I \varphi = \top U_I \varphi \), and \( \Box_I \varphi = \neg \Diamond_I \neg \varphi \).

An execution sequence \( \tau \) satisfies the formula \( \Box_I \varphi \) (respectively \( \Diamond_I \varphi \)) at time \( t \), if and only if \( \varphi \) is satisfied at any time (respectively, at a certain time) of the interval \( t + I \).

**LTL operators**, \( \Diamond \), \( \Box \) and \( U \) of [18] can be represented using MITL operators:

\[ \Diamond \varphi = \Diamond_{[0, \infty)} \varphi \ ; \ \Box \varphi = \Box_{[0, \infty)} \varphi \ ; \ \varphi_1 U_I \varphi_2 = \varphi_1 U_{[0, \infty)} \varphi_2 \]

The operator \( \leadsto_I \) (leadsto) can be defined by the operators *always* and *eventually*: \( \varphi_1 \leadsto_I \varphi_2 = \Box (\varphi_1 \Rightarrow \Diamond_I \varphi_2) \).

### 3.2 Event timed logic EMITL

We extend the MITL grammar in order to specify reaction properties in a bounded time to events received from the environment. For instance, formulas of type \( e \leadsto_{[0, k]} p \) with \( e \) an event and \( p \) a proposition, are not supported by the MITL logic. This extension allows such formulas as well as the simultaneous occurrence of several events. Note that for the LTL temporal logic [17, 18], there is a state/event derivative defined in [8].

The grammar of EMITL is defined on a set of propositions \( P \) and a set of events \( \Sigma \) by:

\[ \varphi ::= \top \mid p \mid e \mid \neg \varphi \mid \varphi \land \varphi \mid \varphi U_I \varphi \]

with \( e \) an event of \( \Sigma \), \( p \) a proposition of \( P \) and \( I \) a non-singular interval.

**EMITL semantics** The semantics of this logic derives from that presented in section 3.1 after having extended the notion of timed execution to events.

An event timed execution \( \theta \) is a 3-tuple \( (\bar{\sigma}, E, I) \) where \( \bar{\sigma} \) is a state sequence, \( E \) is a sequence of event sets, and \( I \) is a sequence of disjoint and adjacent intervals whose union covers \( \mathbb{R}^+ \). We write also \( \theta \) by:

\[ \theta = E_0 \rightarrow (\sigma_0, I_0) E_1 \rightarrow (\sigma_1, I_1) \ldots E_i \rightarrow (\sigma_i, I_i) E_{i+1} \rightarrow (\sigma_{i+1}, I_{i+1}) \ldots \]

Such an execution expresses that the system is in the state \( \sigma_i \) in the interval \( I_i \), and that all events of \( E_i \) occur at time \( l(I_i) \). The simultaneous occurrence of events enables us to express the parallel composition of events such as for instance

\[ I + t = \{ t' \in \mathbb{R}^+ \text{ such as } t' - t \in I \} \]

184
the parallel operation calls in B. $E_i = \emptyset$ means that no event is attached to the
transition which corresponds to the internal events. It will be denoted $\text{skip}$.

The EMITL semantics is an extension of the MITL’s one. For a timed execution sequence $\theta = (\sigma, E, T)$ and an EMITL formula $\varphi$, the satisfiability relation, $\theta$ satisfies $\varphi$ denoted by $\theta \models \varphi$, is semantically defined by extending MITL semantics by the following rule:

$$(\theta, t) \models e \text{ iff for some } k, t = l(I_k), \text{ and } e \in E_k$$

**Example** Consider the example of the transaction mechanism described in [6]. Here, we are interested in checking the following liveness constraint:

"After the release of the event beginTransaction, events abortTransaction or commitTransaction must be enabled within 10 time units". This property is expressed in EMITL by:

$$\Box(\text{beginTransaction} \Rightarrow \Diamond_{[0,10]}(\text{commitTransaction} \lor \text{abortTransaction}))$$

### 3.3 The EMITL$_{past}$ logic

The past extension of the EMITL logic, denoted EMITL$_{past}$ (Past Event Metric Interval Temporal Logic), is represented by the following grammar:

$$\varphi ::= \top \mid p \mid e \mid \neg \varphi \mid \varphi \land \varphi \mid S_I \varphi$$

Given an event timed execution sequence $\theta$:

$$\theta = E_0, (\sigma_0, I_0) \xrightarrow{E_1} (\sigma_1, I_1) \ldots E_i, (\sigma_i, I_i) \xrightarrow{E_{i+1}} (\sigma_{i+1}, I_{i+1}) \ldots$$

The semantics of EMITL$_{past}$ is represented by:

- $\theta \models \varphi$ iff $(\theta, 0) \models \varphi$.
- $(\theta, t) \models e$ iff for some $k, t = l(I_k), \text{ and } e \in E_k$.
- $(\theta, t) \models p$ iff for some $k, t \in I_k, \text{ and } p \in \rho(\sigma_k)$, where $\rho$ is the valuation function associating to each state a set of propositions.
- $(\theta, t) \models \neg \varphi$ iff $(\theta, t) \not\models \varphi$.
- $(\theta, t) \models \top$.
- $(\theta, t) \models \varphi_1 \land \varphi_2$ iff $(\theta, t) \models \varphi_1$ and $(\theta, t) \models \varphi_2$.
- $(\theta, t) \models S_I \varphi_2$ iff $t - r(I) \geq 0$ and for some $t', t - r(I) \leq t' \leq t - l(I)$, $(\theta, t') \models \varphi_2$, and for every $t''$ such that $t' \leq t'' \leq t$, $(\theta, t'') \models \varphi_1$.

Operators $\Diamond$ et $\Box$ are defined respectively using the operator $S_I$ by: $\Box_I \varphi = \neg \Diamond_I \neg \varphi$, and $\Diamond_I \varphi = \top \land S_I \varphi$.

### 4 Specification language: the $B$ method

In B [1], a development begins with the construction of a model which expresses the requirement specifications. It specifies in abstract machines the main state variables of the system and the properties (invariants) that these variables must
satisfy at any time. Operations (services) allow to transform the machine
variables. The B model is then refined (specialized) until obtaining a complete imple-
mentation of the software system. Several refinements can satisfy a specification,
the choice of the solutions relies on various criteria such as the execution speed,
the computation precision or the simplicity of validation. Refinement can also
be used as a specification technique. It allows to include gradually the problem
details in the formal development. The formal specification is then realized gradu-
ally and not directly. The B method is supported by a tool called Atelier B [5].
It automatically generates proof obligations to be proven in order to verify that
the invariant is preserved by the operations of the machine.

4.1 Abstract machines

The abstract machine is the decomposition unit of the B method. This concept is
very close to the notions of modules, classes or abstract data types. It contains:

- a static part which specifies the system state. This specification defines the
  variables describing the state of system components and the invariants which
  are logical formulas expressing the static rules of the system,
- a dynamic part which expresses the initialization and the evolution of the
  system state through a set of operations.

The static rules that the variables must follow, are defined by predicates de-
scribed using mathematical concepts and constitute the invariant of the abstract
machine. The operations are defined by generalized substitutions, and each one
contains:

- a precondition represented by a predicate which expresses the conditions
  under which the operation can be called,
- an action represented by a substitution which expresses how the machine
  variables evolve.

Each operation must preserve the invariant of the machine. This constraint re-
results in a proof obligation to be proven for validating the machine. Note that
the B property language relies on the first-order logic and the set theory.

4.2 Refinement

The refinement mechanism consists in reformulating, by successive steps, the
variables and the operations of the abstract machine, so as to lead finally to
a module which constitutes a running program. The intermediate steps of re-
formulation are called refinements and the last refinement level is called the
implementation (M₁ ⊆ M₂ ⊆ ... ⊆ Implementation). Refinements must preserve
operation interfaces, such as they were defined in the abstract machine. Each B
component (abstract machine, refinement or implementation) is structured us-
ing a single language, the B language. The refinement of an operation is correct
if it establishes the same result as its abstraction.
4.3 Substitution semantics

B operations are specified by substitutions, for example, a preconditioned substitution **PRE P THEN S END** (the substitution S is to be applied only when P is satisfied), a guarded substitution **SELECT P THEN S END** (S remains blocked until P holds). The semantics of a substitution S is expressed in term of a predicate transformer denoted [S]. This semantics is based on Hoare’s triples [13] and the weakest precondition calculus: let P and Q be logical formulas, and S a substitution. In B, the weakest precondition of S is denoted [S]Q which represents the set of states from which the execution of S leads to a state verifying Q. To prove that {P}S{Q} is satisfied, we must prove the formula P ⇒ [S]Q. The postcondition Q is the invariant of the B machine.

The following table defines the rules for calculating the weakest precondition of the B generalized substitutions.

<table>
<thead>
<tr>
<th>S</th>
<th>[S]Q</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>skip</td>
<td>Q</td>
<td>“do nothing”</td>
</tr>
<tr>
<td>x:= e</td>
<td>Q[e/x]</td>
<td>affectation</td>
</tr>
<tr>
<td><strong>PRE P THEN S END</strong></td>
<td>P ∧ [S]Q</td>
<td>precondition</td>
</tr>
<tr>
<td><strong>SELECT P THEN S END</strong></td>
<td>P ⇒ [S]Q</td>
<td>guard</td>
</tr>
<tr>
<td>S:T</td>
<td>[S][T]Q</td>
<td>sequence</td>
</tr>
<tr>
<td><strong>ANY x WHERE P THEN S END</strong></td>
<td>∀x.P ⇒ [S]Q</td>
<td>unbounded choice</td>
</tr>
<tr>
<td>S</td>
<td></td>
<td>T</td>
</tr>
<tr>
<td><strong>CHOICE S₁ OR S₂ END</strong></td>
<td>[S₁]Q ∧ [S₂]Q</td>
<td>bounded choice</td>
</tr>
</tbody>
</table>

The B language defines the notions of feasibility fis and termination trm, and gives a relational point of view via the predicate prd.

**Termination**

We express that a substitution S terminates by the predicate trm(S). Predicates true and false being represented respectively by btrue (⊤) and bfalse (⊥), the semantics of trm(S) is defined by:

\[
\text{trm}(S) = [S]⊤
\]

This means that a substitution invoked outside its domain does not terminate.

**Feasibility**

We can meet some substitutions which do not have any meaning. This comes from the fact that their guards do not hold. This type of substitutions has the following form: ⊥ ⇒ S. So, whatever the postcondition Q, the substitution S establishes Q since: [⊥ ⇒ S]Q ⇔ ⊥ ⇒ [S]Q which is indeed always true. This substitution is called miraculous because it allows to establish any property. Thus it is unrealizable or non-feasible. The predicate fis(S) characterizes the values for which S is feasible. It is defined by:

\[
fis(S) = \neg[S]⊥
\]
Before-after predicate

The before-after predicate determines the relation between the values of a variable $x$ before and after the application of a substitution $S$. We denote the after value of this variable by $x'$. The before-after relation of the variable $x$ for the substitution $S$ is denoted by $\text{prd}_x(S)$. It means that $x'$ is one of the possible values of $x$ after the application of $S$. The semantics of $\text{prd}_x(S)$ is defined by:

$$\text{prd}_x(S) = \neg\neg[S](x' \neq x)$$

The double negation is necessary because substitutions can be non-deterministic, for instance:

$$\text{prd}_x(\text{CHOICE } x := x + 1 \text{ OR } x := x + 2 \text{ END }) = (x' = x + 1 \lor x' = x + 2)$$

4.4 Adding time to B

There are two aspects in time: an ordering notion allowing to classify events in time, and a metric notion allowing to express durations between events. The second aspect is necessary to the specification of timing constraints of real time reactive systems. To express these constraints, the EMITL logic was used.

Time is not primitive in B. A possible expression of time in B uses clock variables, which are discretized by integer values. An operation $\text{tick}$ increases them. We can meet the absence of equity problem in B, since an operation guarded by an indefinitely true guard, can be infinitely often crossed to the detriment of other operations whose guards are true at the same time. To solve this problem, it is necessary to interleave clock update operations with other machine operations.

5 B timed extension

This section presents a timed extension of the B method, relying on the EMITL logic. In order to express timing constraints on the environment and timed reactions of the controller in B, we have added respectively to B property and substitution languages, new temporal operators ($\text{during}$, $\text{since}$, $\text{period}$...) and a new substitution $\text{wait}$. Thus, operations with timed preconditions or guards expressed by $\text{during}$, $\text{since}$ and $\text{period}$ must preserve the predicates of $\text{TIMING_INARIANT}$ clause. This preservation is done under the properties of $\text{TIMING_CONSTRAINTS}$ clause. Those two clauses contain logical formulas expressing respectively system timing constraints and environment assumptions. We distinguish two types of temporal operators:

- past operators represented by $\text{during}$ and $\text{since}$,
- futur operators represented by $\text{always}$ and $\text{leadsto}$. They are used to express properties of $\text{TIMING_CONSTRAINTS}$ and $\text{TIMING_INARIANT}$.

5.1 Extension of the property language

Here, we present the syntax and the semantics used in the property language of the timed extension of B.
Syntax  We have extended the grammar of the B property language by introducing new high level temporal properties. According to logical operators, we have divided the syntax of this grammar into:

- Syntax for the temporal properties relying only on the current state of the system. It is defined by:
  \[ P : = \top \mid p \mid Op \mid P \Rightarrow P \mid P \land P \mid \neg P \mid \forall x.P \mid \exists x.P \mid \text{period}(d) \]

- Syntax for the temporal properties relying on the current or the future states of the system. It is defined by:
  \[ P^+ : = \top \mid p \mid Op \mid P^+ \Rightarrow P^+ \mid P^+ \land P^+ \mid \neg P^+ \mid \forall x.P^+ \mid \exists x.P^+ \mid \text{period}(d) \mid \text{always} P^+ \mid \text{always(I)} P^+ \mid P^+ \text{ leadsto(I)} P^+ \]

- Syntax for the temporal properties relying on the current or the past states of the system. It is defined by:
  \[ P^- : = \top \mid p \mid Op \mid P^- \Rightarrow P^- \mid P^- \land P^- \mid \neg P^- \mid \forall x.P^- \mid \exists x.P^- \mid \text{period}(d) \mid P^- \text{ during}(d) \mid P^- \text{ since}(I) P^- \]

where \( p \) is a proposition, \( x \) is a variable, \( Op \) is an operation call event, \( I \) is a non-singular interval and \( d \in \mathbb{N}^* \) is a constant duration.

As we will see in the weakest precondition calculus for the extended property language (Section 5.2), we need to distinguish between \( P, P^+ \) and \( P^- \). Note that the operators always, leadsto, during and since model time explicitly, but without using clocks.

Temporal predicates semantics  Given the temporal properties \( P, P_1 \) and \( P_2 \), we represent the temporal predicates semantics by:

- \text{period}(d): has the value \( \top \) each \( d \) time units. Given an event timed execution \( \theta = (\overline{a}, E, T) \) (section 3.2), the semantics of \( \text{period}(d) \) is defined by:
  \[(\theta, t) \models \text{period}(d) \text{ iff } t \mod d = 0\]

- \text{always} \( P \): \( P \) holds at each time; it is defined by: \( \square P \).

- \text{always(I)} \( P \): the semantics of this predicate is represented by the EMITL formula: \( \square_I P \).

- \( P_1 \text{ leadsto(I)} P_2 \): at each time, if the property \( P_1 \) holds, then in the \( d \in I \) time units, the property \( P_2 \) will be satisfied. The semantics of \( P_1 \text{ leadsto(I)} P_2 \) is given by: \( \square (P_1 \Rightarrow \Diamond_I P_2) \).

- \( P \text{ during}(d) \): \( P \) was satisfied during the last \( d \) time units. Semantically \( P \text{ during}(d) \) is equivalent to the EMITL_{past} formula: \( \square_{[0,d]} P \).

- \( P_1 \text{ since}(I) P_2 \): \( P_2 \) was satisfied there is \( d \in I \) time units, and \( P_1 \) is satisfied since. The semantics of this predicate is expressed by the following formula: \( P_1 S_I P_2 \).

We can classify these temporal properties into: environment timing constraints defined in the TIMING_CONSTRAINTS clause, machine temporal properties defined in the TIMING_INVARIANT clause and temporal preconditions or guards.
Environment timing constraints These properties are defined in the TIMING_CONSTRAINTS clause or by temporal preconditions. They express assumptions on the environment, needed for the satisfaction of the machine temporal properties. These constraints are generally expressed by the predicate always(I) P.

Example
Consider the example of the level crossing described by [11] and suppose the following timing constraint: "When the train approaches, it must not reach the barrier zone before at least K time units". It can be specified in the TIMING_CONSTRAINTS clause by:

always (approach ⇒ always([0..K])¬enter)

where the event enter is enabled when the train reaches the barrier zone. The same constraint can be specified by a temporal precondition in the operation enter, expressing that the event approach did not occur too recently:

enter = PRE ¬approach during(K) ∧ ... THEN ... END

Machine temporal properties These properties express the bounded reaction constraints that must be ensured by the system. They are represented by the predicates of the TIMING_INVARIANT clause. In general, these predicates are expressed using the leadsto operator.

Example
Consider the following timing constraint: "When the train approaches, the barrier must start closing within K time units". We can express this property in our timed extension of B by:

approach leadsto([0..K]) start_closing

where approach and start_closing are operation call events.

Temporal preconditions and guards The temporal preconditions or guards allow to express constraints on the operation occurrences. They are represented by the predicates P during(d), P1 since(d) P2 and period(d) which must be satisfied by the environment or the system. These properties that can be eventually checked at run time, should not depend on the future.

Example
Consider the following timing constraint: "When the train approaches, it must reach the barrier zone after at least K time units". It is an environment constraint which the controller supposes to be satisfied. This constraint is expressed by a temporal precondition on the operation enter:

enter = PRE (train_state = before) during(K) ∧ ... THEN ... END

It means that the operation enter should be called only if the train remained at least K time units in the state before.
5.2 Extension of the substitution language

We have incorporated into the B substitution language, a new substitution $\text{wait}(I)$ expressing a non-deterministic wait of $d \in I$ time units. The semantics of this substitution is represented by:

<table>
<thead>
<tr>
<th>$S$</th>
<th>$[S]Q$</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{wait}(I)$</td>
<td>$\Box_I Q$</td>
<td>non-deterministic wait of $d \in I$</td>
</tr>
</tbody>
</table>

This definition means that if $Q$ is true at any time in $I$, then after executing $\text{wait}(I)$, $Q$ will be true. We can use the substitution $\text{wait}(I)$ in two ways according to how time is managed. Let $S$ be a substitution:

- If the actions are not atomic and thus have execution durations, we can write $S \parallel \text{wait}(I)$. This means that the substitution $S$ lasts between $l(I)$ (lower bound) and $r(I)$ (upper bound) time units.
- If the actions of the machine have null durations, either we apply the substitution $S$ to change the state, or we leave the time progress by executing $\text{wait}(I)$.

Before presenting the syntax and the semantics of the timed machine, we define the predicate transformer $[S]$ associated to a substitution $S$ using the same rules as classical B, except for the multi-assignment rule. It is defined over temporal constructors of the fragment $P^+$ of future properties by:

- $[V_1,..,V_n := v_1,..,v_n]_I Q = [V_1,..,V_n := v_1,..,v_n]_I Q$
- $[V_1,..,V_n := v_1,..,v_n]_{\Diamond I} Q = [V_1,..,V_n := v_1,..,v_n]_{\Diamond I} Q$

In the following section, we will present the syntax and the semantics of a timed B machine using our B extension. This machine allows to specify temporal and functional properties of a real time reactive system.

6 Abstract timed machine

The invariant of a B machine describes a property to be preserved after the execution of the machine operations. The conservation of this invariant is verified under environment assumptions, which are expressed by the preconditions or the CONSTRAINTS clause. In the real time context, we have given the possibility of specifying behavioral constraints on the environment using the new clause TIMING_CONSTRAINTS. Likewise, the new clause TIMING_INVARIANT allows to express temporal properties which must be preserved by the operations. These clauses contain the predicates which use the temporal operators always(I) and leadsto(I).

In timed B, the operations of a real time reactive system are specified according to their nature (uncontrollable or controllable).

- Uncontrollable operations can be modelled in B by the preconditioned substitution:

  \begin{verbatim}
  PRE P THEN S END
  \end{verbatim}

  the precondition $P$ is supposed to be true when the operation is called.
Controllable operations can be specified in B by the substitution:

\[
\text{SELECT } P \text{ THEN } S \text{ END}
\]

these operations remain blocked until the guard \( P \) is satisfied. The system must ensure the correct scheduling of controllable operation calls.

In our timed extension of B, the real time reactive systems can be represented by the following timed machine:

\begin{verbatim}
MACHINE \( M_t(X) \) /* X are the parameters of the machine */
TIMING_CONSTRAINTS \( C_t(X) \) /* Environment hypotheses */
CONSTANTS \( C, d_1, \ldots, d_k \) /* Constant durations */
PROPERTIES \( \text{Prop}(C) \land d_1 \ldots d_k \in \text{NATURAL} \)
VARIABLES \( V \)
INVARIANT \( I(V) \)
TIMING_INVARIANT \( I_t(V) \) /* Controller reactions */
INITIALISATION \( \text{init}(V) \) /* Initialization of variables \( V \) */
OPERATIONS
/* Specification of the uncontrollable operations */
\( \text{Op}_i = \text{PRE } P_i \text{ THEN } S_i \text{ END; } \)
\( \ldots \)
\( \text{Op}_i = \text{PRE } P_i \text{ during}(d_i) \text{ THEN } S_i \text{ END; } \)
/* Specification of the periodic operations */
\( \text{Op}_j = \text{PRE } P_j \land \text{period}(d_j) \text{ THEN } S_j \text{ END; } \)
/* Specification of the controllable operations */
\( \text{Op}_k = \text{SELECT } P_k \text{ during}(d_k) \text{ THEN } S_k \text{ END; } \)
\( \ldots \)
\( \text{Op}_n = \text{SELECT } P_n \text{ THEN } S_n \text{ END; } \)
/* Specification of the time progression */
\( \text{tick} = \text{ANY } l, r \text{ WHERE } \)
\( l \in \text{NATURAL} \land r \in \text{NATURAL} \land 1 \leq r \land \text{always}(0..r)(I(V) \land I_t(V)) \)
\( \text{THEN } \text{wait}(l..r) /* l and r are respectively the lower and upper bounds */ \)
END
\end{verbatim}

Time progression is modelled by the operation \( \text{tick} \) using the substitution \( \text{wait} \). This progression is done if and only if \( \text{always}(0..r)(I(V) \land I_t(V)) \) is satisfied, i.e., the temporal and the machine invariants must be preserved at any time \( t \in [0, r] \). Moreover, in order to define the semantics of a such machine on timed execution traces, we suppose that two operations of the machine \( M_t \) are not called at the same date; so we must execute \( \text{tick} \) before each operation of \( M_t \).

6.1 Preserving the invariant by operation calls

If we ignore \( \text{SETS}, \text{CONSTANT} \) and \( \text{PROPERTIES} \) clauses, we can denote the previous machine \( M_t \) by: \( M_t = (\Sigma_{Op}, C_t, V, I_t, I_{init}, < M_{Op} >_{Op \in \Sigma_{Op}}) \), where:

- \( \Sigma_{Op} \) is a finite set of operation names (including \( \text{tick} \)).
- \( C_t \) are the environment properties.
- \( V \) is a tuple of variables.
- $I$ is the machine invariant.
- $I_t$ is the machine timing invariant.
- $M_{init}$ is the initialization of state variables of the machine.
- $<M_{Op} >_{op \in \Sigma_{Op}}$ are the machine operations.

The semantics of operation calls must allow to re-use the proofs concerning the operations definition. In B, if an operation definition preserves the invariant $I$, then the operation calls will preserve the invariant. Consequently, the operation calls $Op_i \in \Sigma_{Op}$ of the timed machine $M_t$ must verify the following property:

$$C_t \land I \land I_t \land P_i \Rightarrow [S_i][I \land I_t]$$

with $S_i$ the substitution of the operation $Op_i$, and $P_i$ the guard or precondition of $Op_i$.

### 6.2 Timed machine semantics

Let $M_t = (\Sigma_{Op}, C_t, V, I, I_t, M_{init}, <M_{Op} >_{op \in \Sigma_{Op}})$ be a timed B machine and $Exec_t(M_t)$ a set of timed executions of this machine. A timed execution $\theta \in Exec_t(M_t)$ is a triple of same length sequences $(\bar{\sigma}, \bar{O_p}, \bar{T})$, where $\bar{\sigma} = \sigma_0 \sigma_1 \sigma_2 \ldots$ is a state sequence, $\bar{O_p} = Op_0 Op_1 Op_2 \ldots$ is a sequence of operation calls with $Op_0 = M_{init}$ and $\bar{T}$ is a sequence of adjacent and disjoint intervals. $\theta$ can also be represented by:

$$\theta = (\sigma_0, I_0) \xrightarrow{Op_1} (\sigma_1, I_1) \xrightarrow{Op_2} (\sigma_2, I_2) \xrightarrow{Op_3} \ldots \xrightarrow{Op_i} (\sigma_i, I_i) \xrightarrow{Op_{i+1}} \ldots$$

It means that the state $\sigma_i$ can be obtained by applying the operation $Op_i \in \Sigma_{Op}$ on the state $\sigma_{i-1}$. We remain in the state $\sigma_i$ during the interval $I_i$ before calling $Op_{i+1}$. If we ignore timing aspects, this can be expressed by the following proof obligation, with $i \geq 1$:

$$V = \sigma_{i-1} \Rightarrow \neg[Op_i](V \neq \sigma_i) \land \text{trm}(Op_i)$$

which can be written as:

$$V = \sigma_{i-1} \Rightarrow \neg[Op_i](V \neq \sigma_i) \land \text{trm}(Op_i)$$

This means that $Op_i$ always terminates in a desired state.

Since in the machine $M_t$ the expression $\neg[Op_i](V \neq \sigma_i)$ can contain a past EMITL formula, this obligation is generalized into:

$$(\xrightarrow{Op_0} (\sigma_0, I_0) \xrightarrow{Op_1} (\sigma_1, I_1) \xrightarrow{Op_2} \ldots \xrightarrow{Op_{i-1}} (\sigma_{i-1}, I_{i-1}), l(I_i)) \Rightarrow \neg[Op_i](V \neq \sigma_i) \land \text{trm}(Op_i)$$

Relying on the prefix definition of the execution $\theta$, we can represent the satisfaction relation by:

$$(\tilde{\sigma}, \tilde{O_p}, \tilde{T}, l(I_i)) \vdash \text{prdy}(Op_i) \land \text{trm}(Op_i)$$

where $\tilde{\sigma} = \sigma_0 \sigma_1 \ldots \sigma_{i-1}$, $\tilde{O_p} = Op_0 Op_1 \ldots Op_{i-1}$, $\tilde{T} = I_0 I_1 \ldots I_{i-1}$ and $l(I_i)$ is the lower bound of $I_i$.

We must also check time progression during the given intervals. For this purpose, we consider the date associated to the lower bound of an interval and check that we can wait during the length of the interval:

$$(\tilde{\sigma}, \tilde{O_p}, \tilde{T}, l(I_i)) \vdash [\text{wait}(|I_i|, |I_i|)] \top$$

193
7 Discussion

In [15, 14, 16], Kevin Lano proposes a technique allowing to express time explicitly in B. It consists in each operation, to advance the clock in a non-deterministic way, for a duration belonging to a bounded interval. The bounds of this interval represent execution durations in the best and the worst case. In our method, we suppose that operations are atomic with null durations, which corresponds to the synchronous model. Time progression is done between operations. This choice allows the direct modelling of a timed automaton in B.

The methods of [7] have a similar semantics to timed automata of [3, 2]. Here the authors express time by discretizing it on the set of nonnegative integers. They also use clock variables to record the enabling date of operations. The timed machine is a B machine integrating a set of clocks and an operation tick. This operation makes all the clocks progress in a synchronous way by one time unit, if and only if the system timing constraints are preserved until the next time. This operation does not modify any other variable of the machine. Other operations of the timed machine can read or reinitialize these clocks. Our approach is more abstract since clock variables are not explicitly expressed in a timed B machine. Furthermore, we impose that operations occur at different times in order to give to a timed B machine a semantics compatible with that of EMITL.

The authors of [9] propose a timed extension of B using the duration calculus [12]. Operations are assigned a duration and a new weakest precondition calculus is defined. However, the duration calculus does not contain events. It would be interesting to study an extension of this proposal with operation names as events.

8 Conclusion

In this study, we have presented the timed B method allowing to express and to check temporal and functional properties of real time reactive systems. It relies on extending the MITL logic to EMITL, and enriching the B method in order to express timing constraints over the environment and to specify controller reactions. We have added to B machines the new clauses TIMING_CONSTRAINTS and TIMING_INVARIANT containing respectively the environment properties and the machine timing constraints. Therefore, we have introduced high level predicates expressed by the temporal operators during, since, and period, to operation preconditions and guards. The operations have to preserve the predicates of the new TIMING_INVARIANT clause. We have also introduced the wait substitution to specify time progression within an interval.

A way to continue this work would be to develop a B translator for EMITL formulas, allowing to translate an EMITL formula into a timed B machine with an explicit expression of clocks through timed automata [19]. We aim also to prove that this machine is a refinement of the machine expressed by our timed B method.
References