Livrable T2-12-B
V1 - 01/2011
Partenaires : LAAS+IRIT+ONERA

DEFINITION DU LANGAGE DE PROPRIETES RT-FIACRE (Quarteft T2-12-B)

Auteurs : Nouha Abid[1,4], Bernard Berthomieu[1,4], Jean-Paul Bodeveix[2,4], Silvano Dal Zilio[1,4], Mamoun Filali[2,4], Didier Le Botlan[1,4], Francois Vernadat[1,4]

1 – CNRS ; LAAS ; 7 avenue colonel Roche, F-31077 Toulouse, France
2 – CNRS ; IRIT ; 118 route de Narbonne, F-31062 Toulouse, France
4 – Université de Toulouse ; UPS, INSA, INP, ISAE, UT1, UTM ;F-31062 Toulouse, France

Quarteft est un projet financé par la FNRAE dans le cadre de son sixième appel à projets.
Fondation de Recherche pour l'Aéronautique et l'Espace
http://www.fnrae.org

Contents

1 Introduction

In this report, we define a high-level language for expressing properties over system expressed using Fiacre. Our approach is based on the definition of a set of property patterns inspired by properties that occur commonly during the specification of concurrent and reactive systems [Pat].

In the current state of the toolchain, properties are checked at the level of the generated state space – that is at the level of the TTS target language – using the SELT model-checker. This has several drawbacks:

- system events that are used in a property should refer to the names as they appear at the compiled level, which means that the name of processes, variables and communication ports used in a property are the one that appear in the TTS, after compilation from Fiacre. Hence this solution is not portable, since the naming convention is contingent on the compiler implementation;
- properties need to be defined using a temporal logic formalism – such as LTL for verification with SELT, or CTL is we use the MUSE model-checker – which are difficult to grasp by non-specialists;
- there are currently no tool supporting timed extension of temporal logic, and the known algorithms have a very high complexity. Therefore it is not possible to check real-time properties of the system using this method.

In the scope of the Quarteft project, we propose several additions to the Fiacre toolchain in order to overcome these problems. Each proposal is defined in its own Section.

In Sect. 1, we define a new **event language** in order to unambiguously refer to the element of a Fiacre model: components; processes; states; communication ports and variables; and to the system events that are associated to these elements, namely a transition to a given state, a communication over a given port, an assignment to a shared variable, etc. We also define a direct mapping from this event language into the system at the TTS level.

In Sect. 3, we define a set of **fundamental properties** for components and processes, such as liveness, safety, etc. The meaning of these fundamental properties is defined by defining the semantics of every operator using an interpretation in LTL.

Finally, after introducing the formal framework required for our definition in Sect. 4 and 5, we define a set of **properties patterns** in the style of [Pat]. The originality of our proposal is to allow the declaration of **timed patterns**, that can express temporal constraints such as the compliance to deadline, worst-case execution time, etc. For every pattern, we give a precise definition based on three different formalism: algebraic formulas over timed traces; diagrams based on a formal, non-ambiguous, graphical language; and a timed extension of LTL named MITL. (All these formalisms, as well as the semantics used to express the set of observable actions of a system, are defined in Sect. 4 and 5.)

Concerning the verification of timed patterns, we propose a method based on the use of *observers*. In a nutshell, to check a pattern P on a Fiacre program M, we compose an observer O_P (that is a TTS specification) with the TTS obtained from the compilation of M and check a given LTL formula ϕ_P on the resulting system. As might be expected, observers are defined so that they do not interfere with the system under observation. This approach has been implemented in a prototype extension to the *frac* compiler: the compiler accept the declaration of patterns directly inside a Fiacre program, automatically generates the corresponding observers and compose them with the specified system.

To conclude, we present another possible enhancement to the definition of properties for Fiacre programs based on the notion of **probes**. In this context, a probe is a look into the global state of a system at runtime that can be retrieved without interfering with the system. This extension could be useful to simplify the definition of observers directly inside a Fiacre specification, that is to allow a user to define his owns observers.

2 Naming of Events in Fiacre

2.1 Naming of Processes and Components

Properties checked in the formal verification of systems typically express constraints on the occurence of *observable events* – and essentially restrict the order and the date of occurrence of these events. In the context of a Fiacre specification, an event is an *instantaneous action* involved in the evolution of the system: it can be a process that moves its current state (more specifically the event of entering into or leaving a given state); a transition in the system (e.g. due to a communication); or a modification of a shared variable.

Remark: in this first version of the property language definition for RT-Fiacre we will not cover events related to shared variables.

The *frac* compiler, used for the conversion of Fiacre into TTS, sets a unambiguous naming convention for the states of a Fiacre program. Each state a – defined in a process P in a Fiacre source code – may correspond to several “state instances”, where each state instance corresponds to a different process instantiation. Indeed, each invocation of a process (respectively a component) in a component leads to the generation of a process (resp. component) instance. It is possible to display the process and component instances of program using the `-arch` command-line option of the frac compiler.

A valid Fiacre specification must declare the main component, say $\bar{C}$. Each process/component instance may be be connected to this initial component in a hierarchical way. To disambiguate two instances of the same process/component, declared in the same context, we use indices. Thus, the *qualified name* $A, B, \ldots$ of a process/component is either: $\bar{C}$, also denoted $//$, for the main component; or the instance i of the process/component C under the context A, that is denoted $A/C[i]$. In the remainder of this text – and based on an analogy with the naming convention used for UNIX file systems – we speak of "absolute path" and "relative path" to identify a process instance relative to the main component or against a context.

Example In the following example, we show how processes and components of a Fiacre program are named with our approach. We have added in front of each identifier, between brackets, the indices computed by frac when generating the TTS file.

```
process A is ...

process B is ...

component D is
  par A || B end

component C is
  par A || D || D  end

component Main is
  par C || D || A  end

Main
```

```
Main–C(1)–A(1)
     |    |–D(1)-A(2)
     |    |      |-B(1)
     |    |–D(2)-A(3)
     |           |-B(2)
     |–D(3)–A(4)
     |      |–B(3)
     |–A(5)
```

- Process //D[1]/A[1] – with our notation – corresponds to process A_4 in the TTS generated from frac.
- State //D[1]/A[1]/init corresponds to the state A_4_sinit in TTS.

2.2 Basic Events

We use uppercase letters $L, M, N, \ldots$ to denote an observable event in Fiacre, and the symbol $[\![L]\!]$ for the interpretation of L as an **event set**. A state a in the process $A[i]$ is denoted $A[i]/a$. Likewise, a communication port p defined in $A[i]$ is denoted $CA[i]/p$. Since transitions are not directly exposed in Fiacre (and are “encoded” inside the macro-transition associated to each state), we do not offer the ability to name them. In the following, we use $t_0, t_1, \ldots$ for the name of transitions and we say $t \in A$ if t is a transition associated to the process A. In the case where the transition is linked with to the same transition. We denote $t \in A/p$ the fact that transition t corresponds to a communication over the port A/p.

- the most basic event is the move of a process, say A, in a new state, say a. This event is denoted $A/a\check{}$;
- conversely, the event associated with the exit of the process A from state a is denoted $A/a\hat{}$;
- the event A/p is used to refer to all the transitions associated with a synchronisation over the port A/p.

2.3 Observable Events

Let A be an instance of a process and B an instance of a component. We use the following notations:

Event L	$Interpretation of L(\llbracket L \rrbracket)$
A/a	$\{A/a\check{}\}$
$A/a\hat{}$	$\{A/a\hat{}\}$
B/p	$\{t \mid t \in B/p\}$
A	$\{t \mid t \in A\}$
B	$\bigcup\{\llbracket A \rrbracket \mid A$ instance dans $B\}$

In the following Section, we define a first set of properties based on our notation for observable events.

3 Fundamental Properties

We define a list of general properties that can be used to express fundamental characteristics of a system, like the absence of a deadlock for example. For each fundamental properties – whenever possible – we give their interpretation in LTL and CTL. In the following table, the symbol L is used for an observable event of the system (or a set of events), as defined in Sect. 2.2 and 2.3.

Property	LTL	CTL
NoGlobalDeadlock	`[]-dead`	`AG-dead`
Never L	`[]-`$\llbracket L \rrbracket$	`AG-`$\llbracket L \rrbracket$
Eventually L	`<>`$\llbracket L \rrbracket$	`AF`$\llbracket L \rrbracket$
InfinitelyOften L	`[]<>`$\llbracket L \rrbracket$	(*not expressible*)
Mortal L	`<>[]-`$\llbracket L \rrbracket$	`AF EG-`$\llbracket L \rrbracket$(*weaker definition*)
Live L	(*not expressible*)	`AG EF`$\llbracket L \rrbracket$

We may give an intuitive definition for each of these basic properties.

NoGlobalDeadlock is true for a system that has no deadlocks or interlocking, that is a state where the system is stuck and cannot evolve.

Never If L is a state (or set of states) then Never L means that the system may never reach reach the states in $\llbracket L \rrbracket$. If L is a reference to a process, then Never L means that the process may never execute $\llbracket L \rrbracket$.

Eventually Is used to express a property on every possible execution of the system. If L is a state (or set of states) then Eventually L means that the system will inevitably reach the states in $\llbracket L \rrbracket$ at least once. If L is a reference to a process, then Eventually L means that the process will inevitably be executed.

InfinitelyOften Similar to Eventually with the additional constraint that the state (resp. process) L should be visited (resp. executed) infinitely often.

Mortal Is used to express that, after some finite time, an event will never occur. If L is a state (or set of states) then Mortal L means that the states in $[\![L]\!]$ will inevitably become inaccessible. It is also possible to propose a weaker version of this operator (based on the CTL definition of the pattern) that is equivalent to the property that it is always possible to reach a state where L will never be reachable.

Live Is used to express liveness properties. If L is a state (or set of states) then Live L means that it is always possible to reach a state in $[\![L]\!]$. If L is a reference to a process, then Live L means that the process may always potentially be executed. (This property cannot be expressed in LTL).

Before describing the rest of our property patterns – and especially the timed patterns – we set the formal framework required for the definition and, in particular, the notion of timed trace.

4 Execution Traces

Properties describe expected behaviours of Fiacre systems. Formally, they are defined over timed traces, which we define now.

Definition 1 (Timed Trace) *Given a set of events Ω, a timed trace σ is a possibly infinite sequence of events $A \in \Omega$ and durations $d(t)$ with $t \in \mathbb{R}^+$. Formally, σ is a mapping from either $\mathbb{N}$ or $[0;n]$ for some $n \in \mathbb{N}$ to $\Omega \cup \{d(t) \mid t \in \mathbb{R}^+\}$.*

Thus, traces can be finite or infinite. The subset of finite traces could be equivalently defined with :
Finite timed trace $\sigma_f ::= \epsilon \mid \sigma_f.d(t) \mid \sigma_f.A$ with $t \in \mathbb{R}^+$, $A \in \Omega$, and ϵ stands for the empty trace.

The set of events Ω is not necessarily finite. In particular, we assume that events contain information about the system's current state. For instance, an event could be a pair of a transition name and the new system state.

Concatenation and duration Given a finite trace σ_1 and a (possibly infinite) trace σ_2, we note $\sigma_1 \cdot \sigma_2$ the concatenation of σ_1 and σ_2 (we omit the details). The concatenation operator is associative. Given a finite trace σ, the duration of σ, written $\Delta(\sigma)$ is defined inductively as :

$$\Delta(\epsilon) = 0 \qquad \Delta(\sigma.d(t)) = \Delta(\sigma) + t \qquad \Delta(\sigma.A) = \Delta(\sigma)$$

Definition 2 (Equivalence over timed traces) *The equivalence relation over timed traces, written $\equiv$, is the smallest equivalence relation satisfying $\sigma.d(0) \cdot \sigma' \equiv \sigma \cdot \sigma'$ and $\sigma.d(t).d(t') \cdot \sigma' \equiv \sigma.d(t+t') \cdot \sigma'$.*

We note that $\sigma_1 \equiv \sigma_2$ implies $\Delta(\sigma_1) = \Delta(\sigma_2)$.

Well-formed traces In the following, we only consider well-formed traces, that is, traces that let time ellapse. Formally all traces σ must satisfy

$$\sigma \text{ infinite} \Rightarrow \forall t > 0\ \exists \sigma_1, \sigma_2\ .\ \sigma = \sigma_1\sigma_2 \wedge \Delta(\sigma_1) > t$$

5 Formalisms

Properties over Fiacre systems are formally defined and illustrated next. To ease the understanding of these properties, we describe them using different means, including a graphical notation (named TGIL) and a timed temporal logic (MITL), both of which are introduced now.

5.1 TGIL: Timed Graphical Interval Logic

Some users may understand more easily a graphical notation rather than a textual definition. For this reason, we define a graphical language, named TGIL, which is mostly inspired by GIL [DKM+94], a graphical language for temporal logic. Note, however, that our notations may differ a bit from GIL's notations.

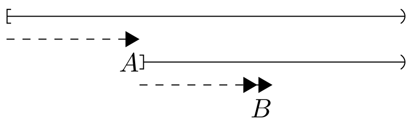
Notation We write with letters A, B, $\ldots$, predicates over events, that is, functions from Ω to booleans. For example, a predicate may check the system's state (since the system's state is part of the event). Or, another predicate may check the occurrence of a given transition.

In the following, we say that an event ω is an occurrence of A if and only if $A(\omega)$ is true.

Execution contexts All TGIL diagrams are defined under an *execution context*, depicted as $[\!\longrightarrow)$ which, by default, represents the whole trace. When a starting point is specified, as in $\overset{A}{]}\!\longrightarrow)$ (for instance A can be the result of a search primitive), then this execution context represents the part of the trace occurring strictly after A. The closed version, including the occurrence of A, is drawn as $\overset{A}{[}\!\longrightarrow)$. Additionally, the ending point can be specified (typically by a search point), as in $[\!\longrightarrow\overset{A}{]}$ or $[\!\longrightarrow\overset{A}{[}$.

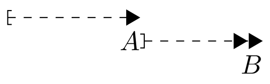
Searches Given an execution context, we define a *weak search* primitive depicted as a dashed arrow $\underset{\mathrm{A}}{\text{- - - - - -}\blacktriangleright}$ which represents the first occurrence of predicate A, if any (it is not an error if no such occurrence exists). The *strong* variant, depicted as $\underset{\mathrm{A}}{\text{- - - - -}\blacktriangleright\!\blacktriangleright}$, requires the occurrence to exist.

Searches results can be used to define new execution contexts, as in this example, to be read from top to bottom :



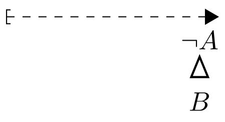
In the default context (above), find the first occurrence of A (which does not necessarily exist), and then find the first occurrence of B located strictly after A (such a B must exist)[1].

We omit execution contexts when they can be unambiguously inferred from the diagram. Thus, the example above is equivalently drawn as :

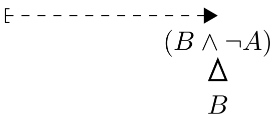


In the following diagram, the triangle below $\neg A$ indicates that predicate B must hold at the very point returned by the search (if any). Thus, the whole diagram means that either A holds on the whole trace, either B holds at the first point where A is false.

This pattern is known as A until B (weak version), usually written A **W** B.

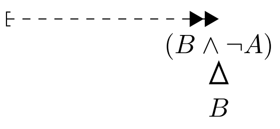


As an optimization, the search could stop as soon as B is found. The following diagram also happens to be equivalent to A **W** B.

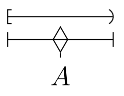


[1]By abuse of language, we just write A instead of *"the occurrence of A"*. Similarly, for B.

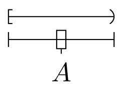
Finally, here is a diagram standing for the strong variant of A until B, written $A\ \mathbf{U}\ B$, which requires B to hold eventually :



Quantifiers Given a context and a predicate A, we can state that A holds at some point in the context, or that A holds for the whole context.

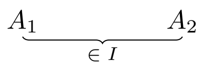


A holds at some point in the context

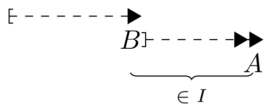


A always holds in the context

If A_1 and A_2 are occurrences of predicates (for example, a result of a *search*, as introduced above, or an event satisfying a predicate), we can state that A_2 occurs after A_1 within a given interval I, that is, $t(A_2) - t(A_1) \in I$, where $t(A)$ is the time of the occurrence of A. This is depicted as follows :

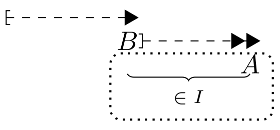


Thus, for example, the following diagram states that the first occurrence of A after the first occurrence of B (if any), is such that the delay between both occurrences belongs to interval I.



This pattern will be introduced later as present A after B within I.

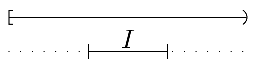
Grouping Grouping is just a graphical notation for parentheses. Compare the following picture with the previous one.



The only difference, the dotted rectangle, groups together predicate A with the time constraint. As a consequence, the search primitive looks for the first occurence of A such that $(t(A) - t(B) \in I)$ (other occurrences of A can exist after B and before this one). In contrast, in the previous example, the first occurence of A had to satisfy $(t(A) - t(B) \in I)$.

Remember that a TGIL diagram should be read from top to bottom, and can be roughly understood as a sequential algorithm.

Time context Given a context, one can restrict it to a time interval I (relatively to the starting point of the context), drawn as follows :



Note that if the given initial context is bounded, the resulting interval might be empty. It should not be considered as an error at this point. Yet, all searches performed within this interval will certainly fail (which is an error for strong searches).

If the given initial context is bounded, we can write $-I$ for the time interval in order to indicate that I is relative to the ending point of the context. Thus $-[1, 2]$ means between 1 and 2 units of time before the ending point.

5.2 MITL: Metric Interval Temporal Logic

Another way to describe formal properties is to use MITL [AFH96], which is a dense-time extension of LTL. MITL core syntax is defined by the grammar : $\phi ::= p \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \phi_1\ \mathbf{U}\ \phi_2 \mid \phi_1\ \mathbf{U}_{[a,b]}\ \phi_2$ where p is a proposition, and $[a,b]$ an interval of $\mathbb{R}^+$. Intuitively, the property $\phi_1\ \mathbf{U}_{[a,b]}\ \phi_2$ means that ϕ_1 must hold until ϕ_2 becomes true, the latter of which must happen between a and b units of time from the current time (for more details, read the precise semantics of MITL [AFH96]), Thanks to these operators, one can derive other standard operators including the time-constrained *eventually* and *always* operators : $\Diamond_{[a,b]}\phi$, defined as True $\mathbf{U}_{[a,b]}\ \phi$, and $\Box_{[a,b]}\phi$, defined as $\neg\Diamond_{[a,b]}\neg\phi$.

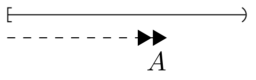
6 Property Patterns

We follow the classification introduced at [Pat], that is, we consider existence patterns, absence patterns, universality patterns, precedence patterns, and response patterns.
For some patterns, we propose a timed version. Note that it is usually the case that we chose one timed extension among several extensions that seemed natural for a given untimed pattern.

Notation We use letters A and B for predicates over Ω, I for time intervals, that is, intervals of $\mathbb{R}^+$, and D for durations (in $\mathbb{R}^+$). The notation $\sigma = \sigma_1\omega_A\sigma_2$ means that there exists ω such that $\sigma = \sigma_1\omega\sigma_2$ and $A(\omega)$ is true. When we do not need to refer to the variable ω, we just write $\sigma = \sigma_1 A\sigma_2$ instead of $\sigma = \sigma_1\omega_A\sigma_2$.
We write $A \in \sigma$ instead of $\exists\sigma_1, \sigma_2\ .\ \sigma = \sigma_1 A\sigma_2$. Given a predicate over events A, we consider it as a predicate overs traces, such that $A(\sigma)$ holds if and only if $\forall\omega \in \sigma\ ,\ A(\omega)$ holds (meaning that A holds over the whole trace σ).

6.1 Existence

Present A

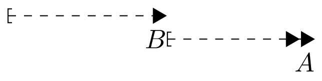


There is an occurrence of A.

LTL definition : $\Diamond A$

Definition over the timed-trace σ : $A \in \sigma$

Present A **after** B

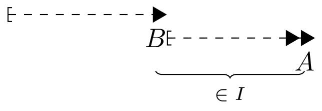


There is an occurrence of A after the first occurrence of B, if any.

LTL definition : $(\neg B)\ \mathbf{W}\ (B \wedge \Diamond A)$

Definition over the timed-trace σ :
$\forall\sigma_1, \sigma_2\ .\ (\sigma = \sigma_1\omega_B\sigma_2 \wedge B \notin \sigma_1) \quad\Rightarrow\quad A \in \omega\sigma_2$

Present A **after** B **within** I



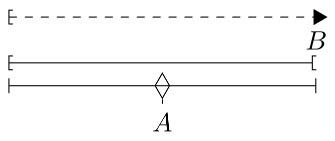
The first occurrence of A holds within I after the first occurrence of B, if any.

MITL definition : $(\neg B)\ \mathbf{W}\ (B \wedge (\neg A)\ \mathbf{U}_I\ A)$

Definition over the timed-trace σ :
$\forall\sigma_1, \sigma_2\ .\ (\sigma = \sigma_1\omega_B\sigma_2 \wedge B \notin \sigma_1) \quad\Rightarrow\quad \exists\sigma_3, \sigma_4\ .\ \omega\sigma_2 = \sigma_3 A\sigma_4 \wedge A \notin \sigma_3 \wedge \Delta(\sigma_3) \in I$

Variants of this pattern may be defined. We can relax the constraints so as to look for one occurrence of A within I instead of the first.

Present A before B

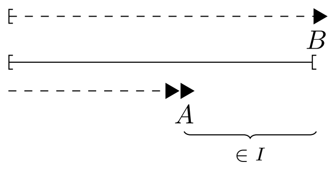


There is an occurrence of A strictly before the first occurrence of B, if any.

LTL definition : $\neg B\ \mathbf{W}\ (A \wedge \neg B)$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \quad \Rightarrow \quad A \in \sigma_1$

Present A before B within I



The first occurrence of A holds within I before the first occurrence of B, if any.

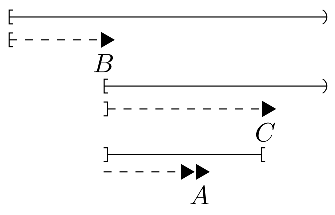
MITL definition : $(\Diamond B) \Rightarrow (\,(\neg A \wedge \neg B)\ \mathbf{U}\ (A \wedge \neg B \wedge (\neg B\ \mathbf{U}_I\ B))\,)$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \quad \Rightarrow \quad \exists \sigma_3, \sigma_4\ .\ \sigma_1 = \sigma_3 A \sigma_4$
$\wedge\ A \notin \sigma_3 \wedge \Delta(\sigma_4) \in I$

In this pattern, we have chosen to consider the first occurrence of A. A similar pattern could be defined by taking the last occurrence of A. Additionally, considering any occurrence of A seems feasible (could be implemented with an observer).

Warning : the two following patterns have definitions that differ from the ones found at [Pat].

Present A between B and C



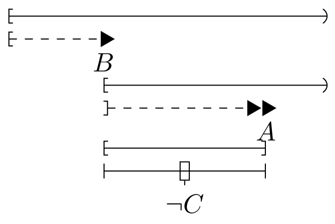
After the first occurrence of B, if any, and before the first subsequent occurrence of C, if any, there must be an occurence of A.

Requirement : B and C must be in exclusion, that is $\neg(B(\omega) \wedge C(\omega))$ holds for every event ω occuring in σ.

LTL definition : $(\neg B)\ \mathbf{W}\ (B \wedge ((\neg C)\ \mathbf{W}\ (A \wedge \neg C)))$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2, \sigma_3\ .\ (\sigma = \sigma_1 B \sigma_2 C \sigma_3 \wedge B \notin \sigma_1 \wedge C \notin \sigma_2) \quad \Rightarrow \quad A \in \sigma_2$

Present A after B until C



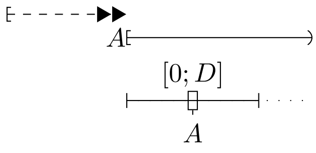
After the first occurrence of B, if any, there must be an occurrence of A that holds before the first subsequent occurrence of C, if any.

Requirement : B and C must be in exclusion (see above).

LTL definition : $(\neg B)\ \mathbf{W}\ (B \wedge ((\neg C)\ \mathbf{U}\ (A \wedge \neg C)))$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \quad \Rightarrow \quad \exists \sigma_3, \sigma_4, \omega\ .\ \sigma_2 = \sigma_3 \omega_A \sigma_4$
$\wedge\ C \notin \sigma_3 \omega$

Present A lasting D



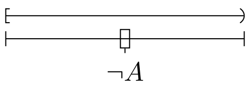
From the first occurrence of A, the predicate A remains true for at least duration D.

MITL definition : $(\neg A)\ \mathbf{U}\ (\Box_{[0,D]} A)$

Definition over the timed-trace σ :
$\exists \sigma_1, \sigma_2, \sigma_3\ .\ \sigma = \sigma_1\sigma_2\sigma_3 \wedge A \notin \sigma_1 \wedge \Delta(\sigma_2) \geqslant D \wedge A(\sigma_2)$

6.2 Absence

Absent A

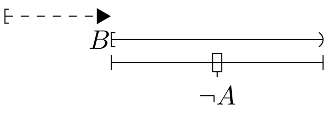


There is no occurrence of A

LTL definition : $\Box\neg A$

Definition over the timed-trace σ : $A \notin \sigma$

Absent A after B

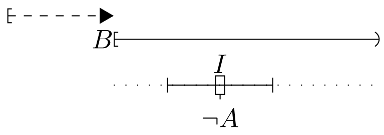


There is no occurence of A after the first occurrence of B, if any.

LTL definition : $\Box(B \Rightarrow \Box(\neg A))$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \quad \Rightarrow \quad A \notin \sigma_2$

Absent A after B for interval I

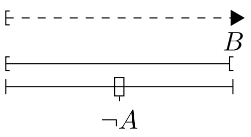


After the first occurrence of B, if any, no A can occur within I

MITL definition : $\neg B\ \mathbf{W}\ (B \wedge \Box_I \neg A)$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2, \sigma_3, \omega\ .\ (\sigma = \sigma_1 B \sigma_2 \omega \sigma_3 \wedge B \notin \sigma_1 \wedge \Delta(\sigma_2) \in I) \quad \Rightarrow \quad \neg A(\omega)$

Absent A before B

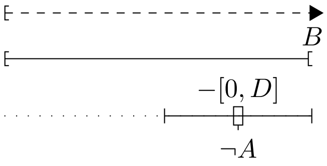


No A can occur before the first occurrence of B, if any.

LTL definition : $\Diamond B \Rightarrow (\neg A\ \mathbf{U}\ B)$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2 \wedge B \notin \sigma_1) \quad \Rightarrow \quad A \notin \sigma_1$

Absent A before B for duration D

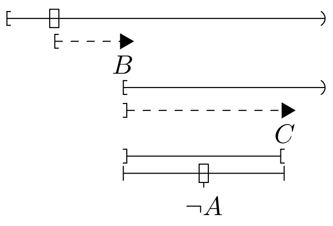


No A can occur less than D unit of time before the first occurrence of B, if any

MITL definition : $\Diamond B \Rightarrow (A \Rightarrow (\Box_{[0;D]} \neg B))\ \mathbf{U}\ B$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2, \sigma_3, \omega\ .\ (\sigma = \sigma_1\sigma_2 B \sigma_3 \wedge B \notin (\sigma_1\sigma_2) \wedge \Delta(\sigma_2) \leqslant D) \quad \Rightarrow \quad A \notin \sigma_2$

Absent A between B and C



After every occurrence of B, if there is a subsequent occurrence of C, then A cannot occur (strictly) inbetween.

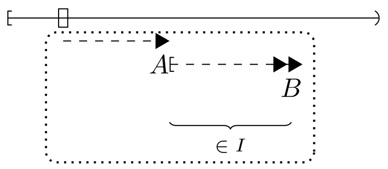
Requirement : B and C must be in exclusion (see above).

LTL definition : $\Box((B \wedge \Diamond C) \Rightarrow (\neg A \ \mathbf{U}\ C))$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2, \sigma_3 \ . \ (\sigma = \sigma_1 B \sigma_2 C \sigma_3 \wedge C \notin \sigma_2) \quad \Rightarrow \quad A \notin \sigma_2$

6.3 Response

A leadsto first B within I

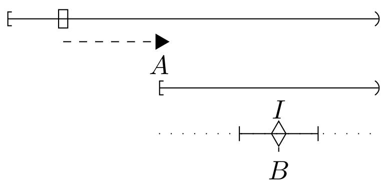


After each occurrence of A, there is an occurrence of B, the first of which occurs within I

MITL definition : $\Box(A \Rightarrow (\neg B)\ \mathbf{U}_I\ B)$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2 \ . \ (\sigma = \sigma_1 \cdot A \cdot \sigma_2) \quad \Rightarrow \quad \exists \sigma_3, \sigma_4 \ . \ \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in I \wedge B \notin \sigma_3$

A leadsto B within I

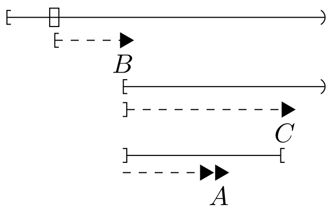


After each occurrence of A, there is an occurrence of B that occurs within I

MITL definition : $\Box(A \Rightarrow \Diamond_I B)$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2 \ . \ (\sigma = \sigma_1 A \sigma_2) \quad \Rightarrow \quad \exists \sigma_3, \sigma_4 \ . \ \sigma_2 = \sigma_3 B \sigma_4 \wedge \Delta(\sigma_3) \in I$

B leadsto A before C



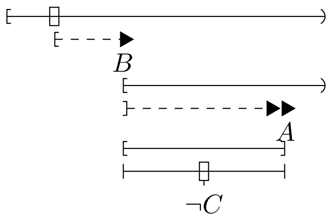
After every occurrence of B, if there is a subsequent occurrence of C, then A occurs (strictly) inbetween.

Requirement : B and C must be in exclusion, that is $\neg(B(\omega) \wedge C(\omega))$ holds for every event ω occuring in σ.

LTL definition : $\Box(B \Rightarrow (\neg C\ \mathbf{W}\ (A \wedge \neg C)))$

Definition over the timed-trace σ :
$\forall \sigma_1, \sigma_2, \sigma_3 \ . \ (\sigma = \sigma_1 B \sigma_2 C \sigma_3 \wedge C \notin \sigma_2) \quad \Rightarrow \quad A \in \sigma_2$

B **leadsto** A **until** C



After every occurrence of B, there must be an occurrence of A, which must hold before the first subsequent occurrence of C, if any.

Requirement : B and C must be in exclusion (see above).

LTL definition : $\Box(B \Rightarrow (\neg C\ \mathbf{U}\ (A \wedge \neg C)))$

Definition over the timed-trace σ :

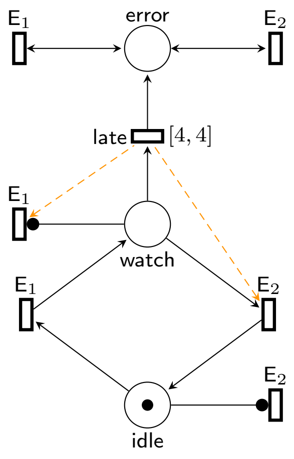
$$\forall \sigma_1, \sigma_2\ .\ (\sigma = \sigma_1 B \sigma_2) \quad \Rightarrow \quad \exists \sigma_3, \sigma_4\ .\ \sigma_2 = \sigma_3 \omega_A \sigma_4 \wedge C \notin \sigma_3 \omega$$

7 Implementing Patterns with Observers

While the basic patterns expressed in Sect. 3 can be translated into a temporal logic framework (and hence be checked using the model-checking tools that accept TTS inputs), it is not possible to do the same with the timed patterns. To check these more expressive patterns, we propose to use an approach based on observers that will be defined at the level of TTS.
We also describe an approach for implementing observers directly in Fiacre through the use of *probes*. While we already provide some tooling that support the definition of patterns in Fiacre, the formal definition of a probe language is scheduled for the next version of this deliverable.

7.1 Observer at the TTS Level

We give an example of observer associated to the property E_1 leadsto E_2 within $[0, 4]$, that is *"every event E_1 is followed by an event E_2 at least before 4 unit of times"* . This observer is expressed in TTS format and only synchronizes with the events E_1 and E_2, which are assumed to be transition of the observed system. Therefore, since we do not make use of data in this case, we can "draw" the observer using the graphical syntax for Timed Petri Net extended with priorities (pictured as dashed line between transitions in the following diagram). We give more information on the behavior of observers in the next Section.

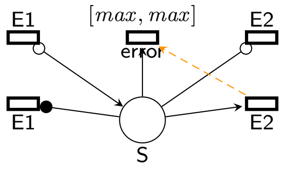


In this example, transition late has priority over E$_1$ and E$_2$. (We also make use of *read arcs*, depicted with rounded ends.) Then, to check that the property is valid, it is enough to compose the TTS obtained from the system with the observer and to check a simple LTL property. In this case that the state error cannot be reach, i.e. Never(error). With the help of observers, we essentially reduce the model-checking of a timed property to the model-checking of a simpler LTL property on an extended system.

7.2 Current Implementation Within Fiacre

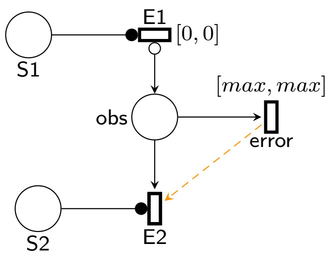
We have defined a set of observers for some of the patterns defined in Sect. 6. In our experiments, and in this Section, we focus our attention on the leadsto pattern.
The target language of the Fiacre language – the TTS format – is quite rich, as it includes both: concurrency constructs (through places and transitions as in Petri Nets); shared variables (through the inclusion of data); and priorities. Therefore we may implement the observer associated to a given pattern in several ways as there is no "natural paradigm". We propose three different classes: a first approach based on the use of transitions (this is the style of observers that has been used in the example of Sect. 7.1); a second based on the observation of places; and a third approach based on the observation of shared, boolean variables added to the system. While the use of transitions are typical for observing Petri Nets behaviors, the use of an *observer on data* is quite new. The results of our experiments show that, in practice, this is the best way to implement an observer.
For educational purpose, we give an example of observer for each of these three different classes for the real-time pattern: E_1 leadsto E_2 within $[0, max]$, meaning that whenever $E1$ occurs, then eventually E_2 must occur in less than max unit of times.

Observer based on transitions: in this example, we assume that $E1$ and $E2$ are labels used in the transitions of the system. We propose a possible implementation based on a *deterministic observer*, that examines all occurrences of E_1 for the failure of a deadline. We only give a simplified version of the correct observer, that needs to be extended if one of the transition $E1$ or $E2$ bears a non-trivial time constraint (that is, different from $[0, \infty]$) in the system. By virtue of the definition of Petri Net composition, the observer duplicates each transition labeled E_1 (respectively E_2): one copy can be fired if place S is empty – as a result of the inhibitor arc – wile the other can be fired only if place S is full.



It is also possible to define a non-deterministic observer, such that occurences of E_1 may be disregarded. This approach is safe – since, using model-checking, all cases will be perused – and close to the treatment obtained when model-checking the equivalent property using an LTL formula. Experiments have shown that the deterministic observer is more efficient, which underlines the benefit of singling out particular properties when looking for optimizations.

Observer based on states: in this approach, we assume that events E_1 and E_2 are associated to the system entering some given states S1 and S2. We can easily adapt this net to observe events associated to transitions in the system. In our case, the observer will be composed with the system through its places S1 and S2 (we can extend the classic composition of Petri Nets over transitions to composition over places). In our observer, we use a transition labeled E1 whenever a token is placed in S1. Note that TTS obtained from Fiacre are 1-safe, meaning that at most one token may be in S1 at any given time. Likewise, we use a transition E2 for observing that the system is in state S2. The error transition is fired if the observer stays in state obs for more than max unit of times, as needed.



Observer based on data: the use of data in the TTS model is quite simple: a TTS has a set of shared variables, its environment (values and variables are typed and there is a rich type system); every transition t may be conditioned by the values of the shared variables (we call it the precondition of t) and, upon firing, a transition may update the value of the environment (we call this the action of t). In this last class of observers, the idea is to condition the error transition of the observer on the value of a private boolean shared variable, say FLAG. Hence the precondition of error is the predicate (FLAG == false). We assume that the initial value of FLAG is false. To connect the observer to the system, we add to every transition in E_1 the action (FLAG := true) and to every transition in E_2 the action (FLAG := false).

In the context of the Quarteft project, we have implemented an extension of the *frac* compiler that understands the syntax of patterns and automatically compose a system with the necessary observer. In the case where the pattern is not valid, we obtain a counter-example, that is a sequence of events (with time information) that leads to a problematic state.
In the following listing, we illustrate the use of patterns in an actual Fiacre specification that models the operation of a simple manufacturing plant. A *factory* assembles products from two *command lines*, L1 and L2, composed of a set of machines. Two *workers* and one *technician* operate the lines, with worker W1 on line L1 and W2 on line L2.
The factory is subject to operational and legal requirements: (1) workers should operate on work cycles of less than 35 minutes, separated by 5 minutes pauses; (2) the duration of a machine task is between 5 and 10 minutes; (3) machines should be maintained after 15 task cycles. The leadsto pattern may be used to check these requirements. For instance, the listing show the declaration of a property that corresponds to requirement that each Worker must pause after at most 35 minutes of work[2].

Listing 1: A factory example in Fiacre

```
property Worker_1_sWork leadsto Worker_1_sPause within [0, 35]

process Machine [StartMachine: machinename, EndMachine: machinename,
                 StartMachineMnt: machinename, EndMachineMnt: machinename]
                 (name: machinename) is

   states Idle, Work, Mnt

   var x: linename, y: machinename, cycle: nat:=0

   from Idle StartMachine?name; to Work

   from Work
   if cycle<= 15 then EndMachine!name; cycle:= cycle+1; to Idle
   else StartMachineMnt!name; to Mnt end

   from Mnt EndMachineMnt?y where y=name; cycle:=0; to Idle


process Worker[Startline: linename, Endline: linename] (line: linename) is

    states Idle, Work, Pause

    var y: linename

    from Idle Startline?y where y= line; to Work

    from Work Endline?y where y=line; to Pause

    from Pause wait [5,5]; to Idle
```

We can define the *complexity* of an observer as its impact on the augmentation of the state space of the observed system. Hence the complexity of an observer O is a function C_O of the system, say S, such that $C_O(S)$ is the value ${}^{size(S \otimes O)}/_{size(S)}$. For verification, we take the size of the state space of a system as the size of a system. This notion of complexity is useful to explain the impact of an observer during verification, if we assume that model-checking a property on a system S is linear in *size*(S). Unfortunately, it is not possible to give an analytical definition of C_O outside of the most simple cases.

[2]Our tool does not currently understand the syntax of events defined in Section 1 of this deliverable.

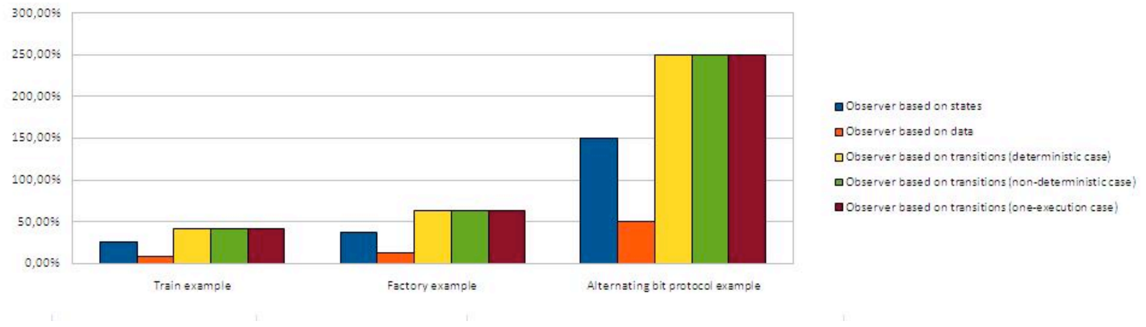


Figure 1: Increase in systems transitions number in the case of verified properties

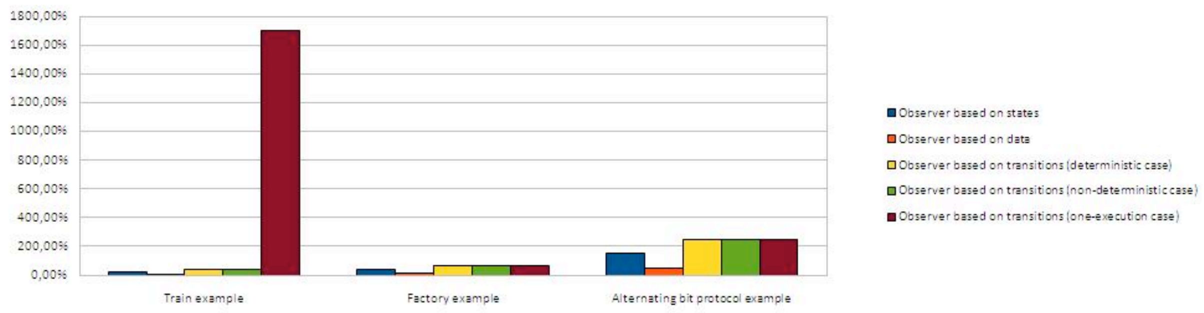


Figure 2: Increase in systems transitions number in the case of unverified properties

We have used our prototype compiler to experiment with different implementations for the observers. The goal is to find the most efficient observer "in practice", that is the observer with the lowest complexity. To this end, we have run several experiments to compare the complexity of different implementations on a fixed set of representative examples. Even if this solution is not perfect, we believe that this approach is better than using randomly generated systems and events. We show in the following Figures two sets of results obtained for the leadsto pattern. In each case, we take the precaution to consider both the cases of valid and invalid patterns. In our experiments, we have consistently observed that observers based on data are the most efficient.

7.3 Observers at the Level of Fiacre

In future version of the system, we envisage to enrich the Fiacre language with a notion of *probes* which can be viewed as special ports that allow to observe the internal state of a process without perturbing it. We do not plan to give more information at this stage but, with this simple analogy, it is already possible to show how the observer defined in the previous Section could be directly expressed in Fiacre .

Listing 2: A factory example in Fiacre

```
process max [E1,E2 : probe]  is
   states idle, watch, error
   from idle
      select
        E1; to watch
   [] E2; loop
   end
   from watch
        select
        E1; loop
   [] E2; to idle
   unless
           wait [4,4]; to error
   end
```

```
    from error
        select
          E1; to error
        [] E2; to error
        end

component obsmax is
    port E1 : probe is Main.1.3.state:A.enter
    port E2 : probe is Main.1.3.state:B.enter
    par max [E1,E2] end
```

Then, to "connect" an observer with the system, we need to extend the declarations of Fiacre with the keyword **probe** that states that the ports E_1 et E_2 must be synchronized with the observer non-intrusively. The complete probe mechanism will be defined in the next milestone of the project.

8 Conclusion

We define a high-level language for expressing properties over system expressed using Fiacre. Our approach is based on the definition of a set of property patterns inspired by properties that occur commonly during the specification of concurrent and reactive systems (see [Pat]). The originality of our proposal lies in in the definition of *timed property patterns* – that is, such that timed constraints can be declared on the occurences of events – and on the definition of a tailored *observer-based verification method.*

The definition of a timed extension for property patterns is quite new, especially when coupled with a static verification method, such as model-checking. While the choice of a particular set of patterns (and a particular interpretation for each pattern) may appear arbitrary, we have based our selection on a set of well-accepted "atemporal" patterns, together with the constraint that validation should be decidable. As a result, we obtain a set of properties that is not comparable with the set of formulas defined in the decidable fragment of timed temporal logics, such as MITL, or to the set of timed properties that may be checked " on-the-fly". For instance, we are able to express properties over unbounded or punctual time intervals.

Concerning our choice of verification technique; while observer-based approaches are well-known, they often rely on observers that are expressed in the same language than the specification (usually without a proof of their " innocuousness") and boils down to checking that the observer will never enter into a failure state. Instead, in our approach, we use observer to convert a timed property into the verification of a plain LTL formula over the composition of the system with an observer. (Hence we can express properties beyond reachability.) To the best of our knowledge, this approach has never been explored before. Also, we define observers in a low-level language, where we can apply more powerfull optimization techniques and where the safety of the oberver is easier to enforce.

References

[AFH96] Rajeev Alur, Tomás Feder, and Thomas A. Henzinger. The benefits of relaxing punctuality. *J. ACM*, 43:116–146, January 1996.

[DKM+94] L. K. Dillon, G. Kutty, L. E. Moser, P. M. Melliar-Smith, and Y. S. Ramakrishna. A graphical interval logic for specifying concurrent systems. *ACM Transactions on Software Engineering and Methodology*, 3:131–165, 1994.

[Pat] Patterns. http://patterns.projects.cis.ksu.edu/.