INDEX

1. ITCL (Interval Temporal Checking Logic)

We present a real-time Interval logic to be used in determining if the execution of a real-time distributed program, as characterized by a captured execution trace, is consistent with a formal description of the program behavior. Our work is intended to yield practical tools for software testers. Therefore, we emphasize the ease of expressing the complex timing and relational properties of real-time distributed software.

1.1. Basic concepts

While temporal logic provides good low-level mechanisms for expressing sequencing behavior, reasoning about an entire computation is often awkward and convoluted. Given the focus on checking and debugging, ITCL is a logic based on actions and system status that are defined as intervals. Reasoning about intervals allows us to define in an easy way timing and relational properties of real-time distributed systems, such as periodic behavior or temporal constrains.

1.1.1. Events

An event ϕ in the context of this document is defined as a log entry in the trace file (see Figure 3). These log entries record relevant changes to the system, including the start and end of significant actions, changes to state variables, or perceived changes in the environment. The information recorded for each log entry contains the name, a timestamp and a set of variables depending on the kind of log entry.

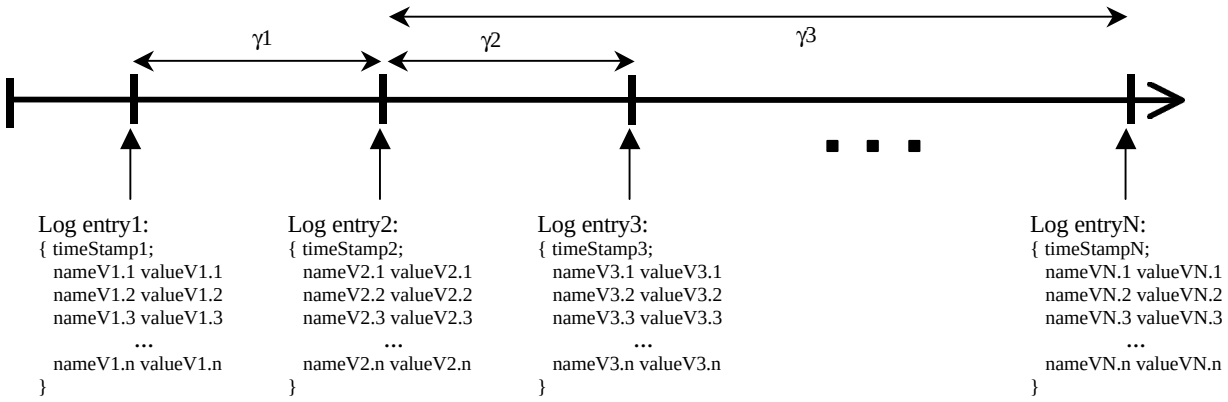

Figure 3. Intervals are defined by events.

Events can also be defined as a log entry *type* (rather than a single log entry *instance*). An event defined in this way can have several occurrences in the trace file. We use the term "*event set*" (represented as Φ here) to denote all of them.

1.1.2. Intervals

As show in figure 3, an execution trace can be represented by a set of intervals (γ), delimited by pairs of events ϕ**1** and ϕ**2**. The first event is included in the interval and the second event is not (this is, **[**ϕ**1,** ϕ**2)**). Therefore, the starting and ending event have to be different.

A *minimum interval* is the interval between two consecutive events. An interval can also span several consecutive minimum intervals. The whole trace itself is also considered as an interval.

Intervals can also be defined as pairs of log entry *types* (rather than single log entry *instances*). An interval defined in this way can have several occurrences in the trace file. We use the term "*interval set*" (represented as Γ here) to denote all of them. When we test for a condition over such a set, the condition is checked for all the occurrences.

1.1.3. Specifications

Specifications or rules to be checked on the trace file can be defined and evaluated with respect to intervals. They consist of propositions or logical expressions composed for different kind of expressions defined according to the RTIL (real time interval logic) defined below.

1.2. Syntax

1.2.1. Making reference to data in the log file.

As described in section 2.1, some events include data with a name and a type. This data can be referenced in the logic using the name of the log entry (event) and the name of the data with a dot in the middle. For example, the data **speed** for the log entry name **train1** would be referenced in the logic as "**train1.speed".** Sometimes is necessary to reference a log entry, in this case we will use the name of the log entry followed by a dot to differentiate it from a regular variable name. For example, "**train1."** in the case of the **train1** event.

1.2.2. Events

As we have defined before, an event ϕ in the context of this document is defined as a log entry in the trace file. Events can be defined grammatically as:

$$\phi \equiv |\uparrow\gamma \mid \downarrow\gamma \mid \phi\rightarrow\mathbf{t} \mid \mathbf{t} \leftarrow \phi$$

Where γ is an interval. The operators $\uparrow$ and $\downarrow$ appearing before an interval represent the beginning and ending of the interval. Figure 4 shows some examples.

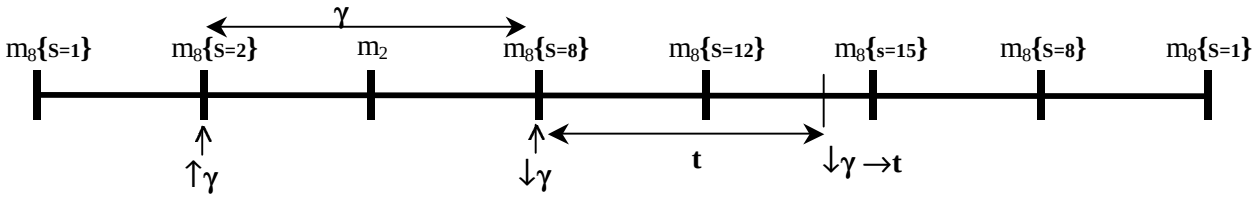

Figure 4. Events.

Finally, $\phi\rightarrow\mathbf{t}$ and $\mathbf{t}\leftarrow\phi$ represent the events **t** time units after ϕ and **t** time units before ϕ respectively. ϕ can be the name of an event variable or the definition of an event as the start or end of an interval.

Event expressions can also be represented between brackets, for example, $(\phi\rightarrow\mathbf{t})$ or $(\downarrow\gamma)\rightarrow\mathbf{t}$. However, these brackets can also be omitted without changing the meaning of the expression.

1.2.3. Intervals

An interval is defined by the starting and ending event. The formal basic definition of an interval is as follows:

$$\gamma := \phi_1 \Rightarrow \phi_5 \Rightarrow \ldots \mid \ldots \Leftarrow \phi_1 \Leftarrow \phi_2 \mid \perp$$

The search operators ($\Rightarrow$,$\Leftarrow$) extend the interval from a starting event searching forward ($\Rightarrow$) or backward ($\Leftarrow$) to an ending event. As always, the ending event is not included. Multiple search operators can be included in the same interval definition, but they must all be of the same direction. If no event is specified, search starts from the beginning $\Rightarrow \phi_2$ or end time $\phi_2 \Leftarrow$ of the log file. Thus, $\Rightarrow$ represents the interval including the whole execution trace. $\perp$ is used to represent the null interval, that is, there is no interval for which the definition holds. Figure 5 shows some examples of intervals defined using the search operators.

Same as events, interval expressions can also be represented between brackets, for example, $(\Rightarrow \phi_2)$. However, they can also be omitted without changing the meaning of the expression.

Operators $\Rightarrow$ and $\Leftarrow$, start the search right after the starting event occurs leading always to intervals with duration greater than zero.

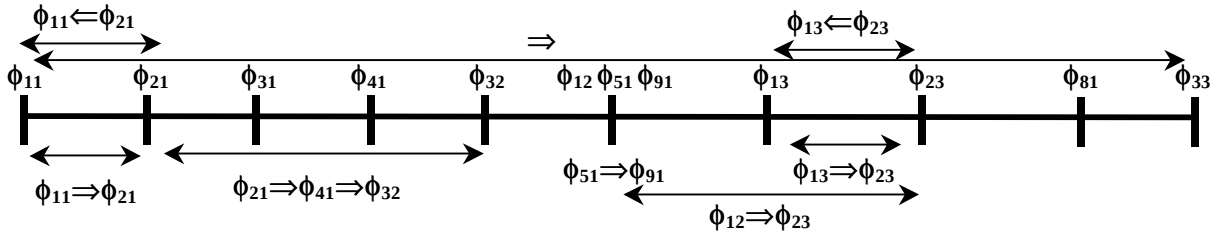

Figure 5. Intervals.

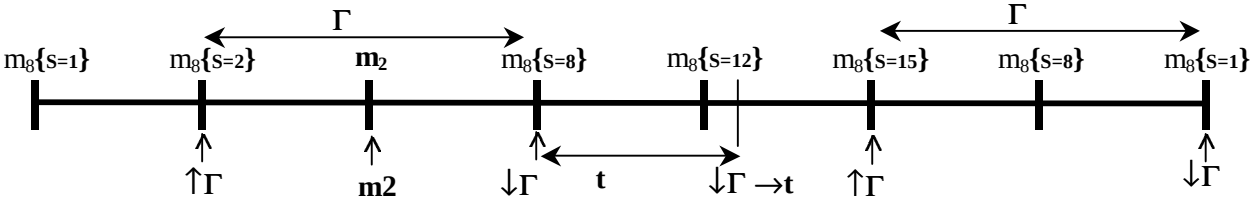

Figure 6. Event sets.

1.2.4. Event sets

As we have defined before, an event set **Φ** in the context of this document is defined as a set of events. Event sets can be defined grammatically as:

$$\Phi \equiv m \mid \uparrow\Gamma \mid \downarrow\Gamma \mid \Phi \rightarrow t \mid t \leftarrow \Phi$$

Where **m** is a log entry from the trace file. The operators ↑ and ↓ appearing before an interval set (**Γ**) represent the beginning and ending of the intervals that belong to **Γ**. Figure 6 shows some examples.

As events and intervals, event set expressions can also be represented between brackets, for example, **(↓Γ→t).**

1.2.5. Interval sets

The formal basic definition of an interval set is as follows:

Γ: = [γ] | [P] | $\Phi_1 \Rightarrow \Phi_2 \Rightarrow \Phi_5 \Rightarrow \ldots$ | $\ldots \Leftarrow \Phi_1 \Leftarrow \Phi_2$ | x:Γ st P| T ∪ T | T ∩ T | T & T | T |T

Where **P** is a condition or logical expression (section 1.3.6) and **T** can be an interval **γ** or an interval set **Γ**. An interval into brackets **([γ])** represents the casting operator that converts the interval **γ** into an interval set that contains only one interval (**γ**).

Next subsections describe some more complex ways to define interval sets from a condition **[P]**, using the **search** operator (⇒,⇐), **conditional** interval sets (**x:Γ st P**) and operations between intervals and interval sets as **union** (∪), **substraction** (∩), **disjunction** (|) and **conjunction** (**&**). Interval set expressions can also be embraced with brackets**.**

1.2.5.1. Defining intervals from conditions

An interval set can also be defined as the intervals of time during which a condition **P** holds using the syntax **[P].** For example, in Figure 7 we have defined an interval where the variable **S** of the log entry $\mathbf{m_8}$ is greater than 10.

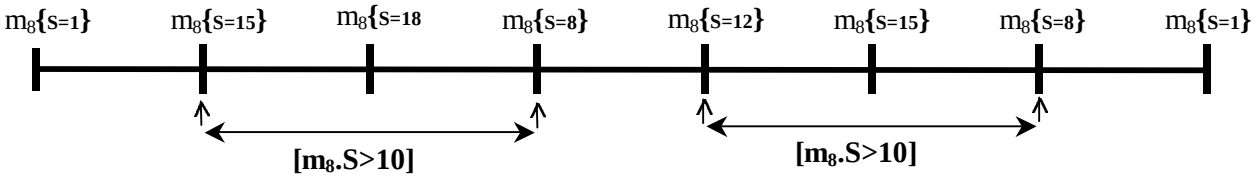

Figure 7. Definition of intervals from conditions.

1.2.5.2. Search operators

As in the intervals case, the search operators (⇒,⇐) extend the interval from a starting event searching forward (⇒) or backward (⇐) to an ending event. Multiple search operators can be included in the same interval definition, but they must all be of the same direction.

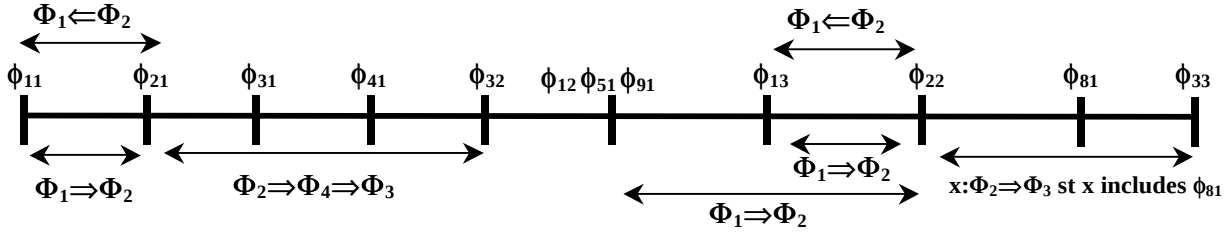

Figure 8. Interval sets.

In Figure 8, all the events ϕ_{1x} belong to the event set Φ_1. We can see that there are three intervals in the interval set $\Phi_1 \Rightarrow \Phi_2$ but only two in $\Phi_1 \Leftarrow \Phi_2$. Also, the interval set $\Phi_9 \Rightarrow \Phi_5$ will

return an empty interval set because there is no ϕ_5 after ϕ_9 and we should remember that the search starts after the starting search point (ϕ_9 in this case).

1.2.5.3. Conditional interval sets

The syntax **x:Γst P** represents the subset of the interval set **Γ** for which the condition **P** holds. For example, **x**:$\Phi_2 \Rightarrow \Phi_3$ **st x include** ϕ_{81} (figure 8) represents the subset of intervals from $\Phi_2 \Rightarrow \Phi_3$ that include the event ϕ_{81}. (**include** operator is defined in section 2.2.6.4)

1.2.5.4. Operations between interval sets and/or intervals

We include four operators to combine intervals and interval sets. The operands can be intervals or interval sets but since an interval in terms of the operation can be considered as an interval set with only one interval, we will discuss here the most general case of operations between interval sets.

1.- The **union** of intervals or interval sets:

$$\Gamma \equiv \Gamma 1 \cup \Gamma 2$$

This means that **Γ** includes all the intervals in **Γ1** and **Γ2.** Note that the individual intervals are not combined[1]. **Γ** is a set that contains all the intervals in the sets **Γ1** and **Γ2.**

Let's see some examples shown in figure 9:

$$\Gamma 1 \equiv \Phi_1 \Rightarrow \Phi_2$$
$$\Gamma 2 \equiv x : \Phi_1 \Rightarrow \Phi_3 \text{ st x include } \phi_8$$
$$\Gamma 3 \equiv \Gamma 1 \cup \Gamma 2$$
$$\Gamma 4 \equiv \Phi_3 \Rightarrow \Phi_4 \cup \Gamma 2$$

or

$$\Gamma 4 \equiv (\Phi_3 \Rightarrow \Phi_4) \cup \Gamma 2$$

2.- **Subtraction** of two interval sets:

$$\Gamma \equiv \Gamma 1 \cap \Gamma 2$$

This defines a new interval set in which the intervals in **Γ2** are subtracted from the areas (time intervals) that are included in **Γ1**[2]. Therefore, the number of intervals resulting can be more or less than the number of intervals in **Γ2** and **Γ2**. For example, $\Rightarrow$ and $(\Phi_3 \Rightarrow \Phi_4)$ have only one interval while **Γ6** that is the subtraction has two intervals.

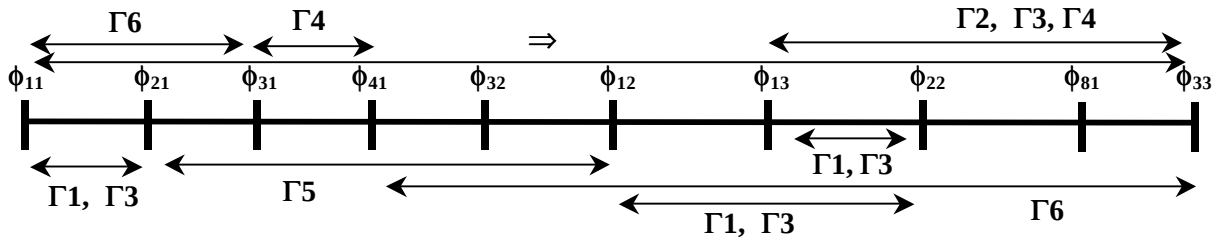

Figure 9. Operations with intervals.

[1] The purpose of this operation is to represent expressions like: "*after doing actions A_1, A_2 or A_3, the communications with earth have to be restarted*".

[2] This helps with expressions like: "*Communications with earth have to be on except while doing actions A_1, A_2 or A_3.*".

As in RTIL, ⊥ is returned when there is no interval for which the definition holds.

Let's see some examples shown in figure 9:

Γ1 ≡ $\Phi_1 \Rightarrow \Phi_2$

Γ2 ≡ x : $\Phi_1 \Rightarrow \Phi_3$ st x include ϕ_{81}

Γ5 ≡ ⇒ ∩ Γ3

Γ6 ≡ ⇒ ∩ $\Phi_3 \Rightarrow \Phi_4$

Figure 9 shows the different intervals. We can see that **Γ3** includes all the intervals that belong to **Γ1** and **Γ2**. On the other hand, **Γ5** are all the intervals in ⇒ when subtracting **Γ3**.

3.- Conjunction

Γ ≡ Γ1 & Γ2

In this case, a time t is in an interval of **Γ** if this time is in an interval of **Γ1** and also in an interval of **Γ2**. Also, consecutive times belong to the same interval so that there are no two intervals in **Γ** that overlap.

Let's see some examples shown in figure 10:

Given **Γ1, Γ2** *and* **Γ3** *as in figure 10***.**

Γ4 ≡ Γ1 & Γ2

Γ5 ≡ Γ1 & Γ3 = empty set

Γ6 ≡ Γ2 & Γ3

4.- Disjunction

Γ ≡ Γ1 | Γ2

Is similar to the case 3 (conjunction) but now a time t is in an interval of **Γ** if this time is in an interval of **Γ1** or in an interval of **Γ2**. Also, consecutive times belong to the same interval so that there are no two intervals in **Γ** that overlap.

Given **Γ1, Γ2** *and* **Γ3** *as in figure 10***.**

Γ7 ≡ Γ1 | Γ2

Γ8 ≡ Γ1 | Γ3

Γ9 ≡ Γ2 | Γ3

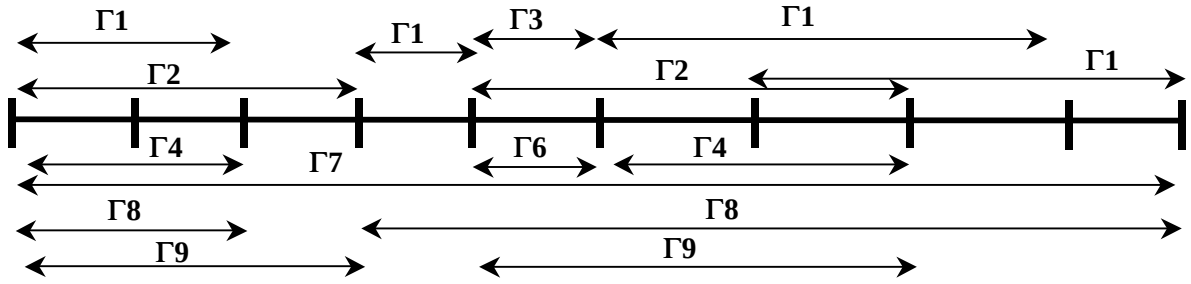

Figure 10. Logical operations with intervals.

1.2.6. Defining specifications.

The specifications are the rules or conditions to be evaluated in the trace file and are formed for one or several logical expressions (**P**) joined by the Boolean operators ∧ (and), ∨ (or) and → (imply) ended with colon. These logical expressions can also have the ¬ (not) operator preceding them. For example, we can have specifications as:

- ✓ **P;**
- ✓ **¬P;**
- ✓ **(¬P1) ∧ P2 ∨ P3;**

Logical expressions (**P**) can be relational expressions, iteration expressions and temporal relations. Each one of them are described in the following subsections but first we must define the operands we use for the different expressions.

1.2.6.1. Operands

We use three main types of data: numeric (integer or float[3]) strings, intervals and events. Intervals and Events have already been defined while strings are delimited by " to avoid ambiguity with the names of the variables. For example, "green" is considered a string.

Time is considered as a numeric value and we provide two functions: ***time****(event_name)* and ***duration****(interval_name)* that return the time of an event and duration of an interval respectively.

The data in the tracefile referenced as <***event_name***>.<***data_name***> can be numeric, string or Boolean. It is important to note that in this case, we reference an array of data. Therefore, we can also work with arrays as we will show in next sections. We can reference an item of an array as <***variable_name***>***[<index>]*** or <***event_name***>.***[<index>]***.<***data_name***> in the case of data associated to an event.

All this type of data and also Boolean values can be stored in variables.

Arithmetic expressions use operators '+', '-', '*' and '/' and can be used as numeric data in the expressions described in the following sections. Operators '*' and '/' have precedence over '+', '-' but it is possible to use brackets to change the order of operations since they have the highest priority.

1.2.6.2. Relational expressions

The result of these expressions is always a Boolean and they use relational operators (<, ≤, >, ≥, =, ≠) that compare two operands. The operands can be numeric (integer or float) a sting, an interval or an event[4]. However, in every relation both operands must have the same type and is not possible to compare for example an interval with a string.

1.2.6.3. Iteration expressions

As we will see in section 3 since we need to define temporal restrictions between two interval sets or event sets rather than two interval or events, we use iteration expressions. We have implemented two different iteration operators ∀ (for all) and ∃ (exists) that can iterate the elements of an event or interval set:

∀ <iterator_name> ∈ <set_name> {<logical_expression>}

The return value of this expression will be true if for each element of the set, the logical_expression evaluates true, otherwise it will return false.

---

[3] Time is also included as numeric since it can be expressed as a float.

[4] With events and intervals only the operators = and ≠ can be used.

∃ **<iterator_name> ∈ <set_name> {<logical_expression>}**
Also defined as ¬∀ **<iterator_name> ∈ <set_name> {¬ <logical_expression>}**

The return value of this expression will be true if there is at least one element of the set for which the logical_expression evaluates true, otherwise it will return false.

These loops can be nested as needed, for example:

```
∀ ϕ22 ∈ Φ2{
        ∃ ϕ11 ∈ Φ1 {
                ↓ϕ11 isbefore [,] ↑ϕ22
        }
}
```

1.2.6.4. Temporal relations

The basic temporal relations have already been defined in section 2.3.6.2 since time is considered as a numeric expression and we can compare two numeric expressions. Since time restrictions between events is used quite often, we added the macros **intersects, inside**, **include** and **isbefore**. We could use the Allen primitives but in most of the expressions we also need quantitative time restrictions. For example, we could need to know if an interval happens always T time units before another interval.

1.- **γ1 intersects γ2**

Two intervals intersect if there are some time points that belong to both.

2.- **γ1 inside γ2**

ϕ inside γ2

One event **ϕ** (interval **γ1**) is inside an interval **γ2** if **γ2** contains **ϕ** (**γ1**). **Include** is the inverse of **inside**.

3.- ϕ_1 isbefore[T1, T2] ϕ_2

Meaning that ϕ_2 happens within the interval [T1, T2] after ϕ_1. T1 can be omitted (**[, T2]**) obtaining by default a zero value (**[0, T2]**). In the same way, T2 can be omitted (**[T1,]**) getting by default ∞ value (**[T1, ∞]**). Also, both can be omitted (**[,]**) which is equivalent to **[0, ∞]**. The interval can be open or closed as shown in table 1.

Table 1. event time relations

MACRO	EQUIVALENCE
γ1 intersects γ2	**time(↑γ1) < time(↓γ2) ∧ time(↓γ1) > time(↑γ2)**
γ1 include ϕ	**time(↑γ1) ≤ time(ϕ) ∧ time(↓γ1) > time(ϕ)**
γ1 include γ2	**time(↑γ1) ≤ time(↑γ2) ∧ time(↓γ1) ≥ time(↓γ2)**
γ1 inside γ2	**time(↑γ1) ≥ time(↑γ2) ∧ time(↓γ1) ≤ time(↓γ2)**
ϕ inside γ2	**time(↑γ2) ≤ time(ϕ) ∧ time(↓γ2) > time(ϕ)**
ϕ_1 isbefore[T1, T2] ϕ_2	**time(ϕ_1) + T1 ≤ time(ϕ_2) ∧ time(ϕ_1) + T2 ≥ time(ϕ_2)**
ϕ_1 isbefore(T1, T2] ϕ_2	**time(ϕ_1) + T1 < time(ϕ_2) ∧ time(ϕ_1) + T2 ≥ time(ϕ_2)**
ϕ_1 isbefore[T1, T2) ϕ_2	**time(ϕ_1) + T1 ≤ time(ϕ_2) ∧ time(ϕ_1) + T2 > time(ϕ_2)**
ϕ_1 isbefore(T1, T2) ϕ_2	**time(ϕ_1) + T1 < time(ϕ_2) ∧ time(ϕ_1) + T2 > time(ϕ_2)**

1.2.6.5. Where are the logical expressions evaluated?

Logical expressions (**P**) can be evaluated with respect to intervals. However, since an interval can be composed of several "minimum intervals" and some expressions are function of data associated to events, an expression can have different values depending on which subinterval it is evaluated. Therefore, we use the following symbols to represent different minimum intervals of an interval **γ** (see Figure 11).

P≡ γΔP | γβP | γ∇P | γα P

- ✓ **γ Δ P** **P** is evaluated in the first minimum interval of **γ**.
- ✓ **γ ∇ P** the last minimum interval of **γ**.
- ✓ **γ β P** the minimum interval following **γ**.
- ✓ **γ α P** the minimum interval preceding **γ**.

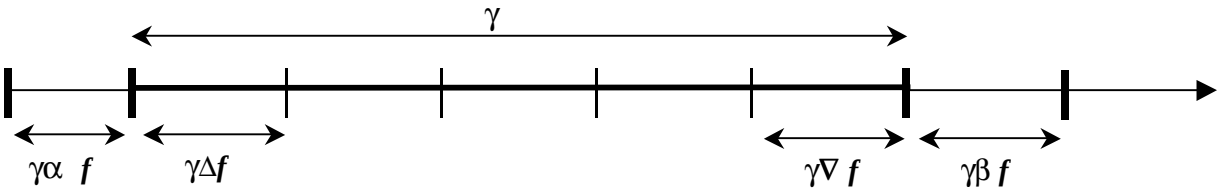

Figure 11. Reference to different parts of an interval.

For example **γ(s<2)** is true if **s** is less than 2 during the first minimum interval inside **γ**.

γ β (s<2) is true if the minimum interval following the interval **γ** has the value of **s** less than 2. **γα (s<2)** is true if the value of **s** is less than 2 in the minimum interval before **γ**.

We also use the temporal operators always (⊗) where **γ ⊗ P** is true if **P** is true during all the intervals within **γ** and eventually (◊) **γ ◊P** as **γ ¬⊗¬P**.

If the interval not exists, for example using the preceding or following operators and there is not preceding or following interval, by default, the logical expression evaluates to true.

1.2.6.6. Variables

Basic variables can store numeric, string, intervals, events, interval sets and event sets or Boolean values. However, typing it is not necessary. Valid names of variables are any sequence of digits and letters starting with a letter.

A variable is defined when a value is assigned in two different expressions. The first one is using the assignation operator (≡) for example **x ≡ 2 + y;**. The second is when they are used as indexes in the iteration expressions as we have seen in section 1.3.6.3.

1.3. Written specifications using ITCL

The Interval Temporal Checking Logic we have presented uses a set of symbols. This set of symbols, as with many other temporal logic languages, use characters that cannot be easily typed from a keyboard. Therefore, we present here the equivalent symbols that we use in our implementation of ITCL.

1.3.1. Mapping symbols

Next table shows the equivalence symbols and their meaning.

Table 2. Symbol equivalence

ITCL	Program	Short description
Arithmetic operators		
+	+	**plus**
-	-	**minus**
*	*	**times**
/	/	**division**
Logic operators and constant values		
¬	!	**Negation**
∧	&&	**And**
∨	||	**Or**
true	true	**True**
false	false	**False**
Relational operators		
=	==	**Equal**
≠	!=	**Different**
≤	<=	**Less or equal**
≥	>=	**Greater or equal**
Iteration operators		
∀	forall	**For all (intervals or events)**
∃	exists	**There is one (interval or events)**
∈	:	**Belongs to a interval or event set**
Specific event and interval operators		
⇒	->	**Search forward for next event**
⇐	<-	**Search backward for next event**
→	~>	**Define new event as time after event**
←	<~	**Define new event as time before event**
∪	union	**Union of two interval sets**
∩	--	**Subtraction of two interval sets**
↑	start	**Beginning a interval or condition**
↓	end	**Ending a interval or condition**
Δ	begin	**First subinterval of a interval**

∇	**last**	**Final subinterval of a interval**
β	**after**	**Interval after**
α	**before**	**Interval before**
⊗	**always**	**Always**
◊	**eventually**	**Eventually**
⊥	**none**	**Null interval**
st	**st**	**Restricts interval set with condition.**
time ϕ	**time(ϕ)**	**Time when an event occurs**
|Γ/Φ|	**cardinal(Γ/Φ)**	**Number of intervals/events in a set**
Miscellany symbols		
≡	**=**	**Assignation**
→	**=>**	**imply: {a=>b} ↔ {!a ||b}**
γ intersects γ1	**γ intersects γ1**	**intersects expression, see table 1.**
γ include ϕ	**γ include ϕ**	**include expression, see table 1.**
ϕ inside γ	**ϕ inside γ**	**inside expression, see table 1.**
duration γ	**duration(γ)**	**duration expression, see table 1.**
isbefore	**is_before**	**isbefore expression, see table 1.**

1.3.2. Syntax description using kind of BNF form

```
identifier:
        IDENTTOK
        ;

numeric_value:
         basic_numeric
        | extend_numeric
        ;

basic_numeric:
        INTTOK
        | FLOATTOK
        | CLOCKTOK
        ;

extend_numeric:
         CARDINALTOK '('interval_set_value ')'
        | CARDINALTOK '('all_event_set_value ')'
        | CARDINALTOK '(' identifier ')'
        | TIMETOK '(' all_event')'
        | DURATIONTOK '(' interval ')'
        ;

message_single_value:
        MSGTOK '['arithmetic_expr']''.'identifier
        | identifier '[' arithmetic_expr ']'
        ;

message_value:
        MSGTOK identifier
        ;

string_value:
         STRINGTOK
        ;
```

```
arithmetic_expr:
        identifier
        | arithmetic_expr_value
        | message_single_value
        | message_value
        ;

arithmetic_expr_value:
        numeric_value
        | arithmetic_expr '+' arithmetic_expr
        | arithmetic_expr '-' arithmetic_expr
        | arithmetic_expr '*' arithmetic_expr
        | arithmetic_expr '/' arithmetic_expr
        | '(' arithmetic_expr_value ')'
        ;

relational_expr_value:
        unknown_type_relational_expr
        | numeric_relational_expr
        | string_relational_expr
        | new_types_relational_expr
        ;

unknown_types_term:
        identifier
        | unknown_types_term_value
        ;

unknown_types_term_value:
        message_single_value
        | message_value
        ;

unknown_type_relational_expr:
        unknown_types_term '<' unknown_types_term
        | unknown_types_term '>' unknown_types_term
        | unknown_types_term LETOK unknown_types_term
        | unknown_types_term GETOK unknown_types_term
        | unknown_types_term_value EQTOK unknown_types_term
        | unknown_types_term_value NETOK unknown_types_term
        ;

/* this can be simplified with unknown_type_relational_expr
   but not  string_relational_expr*/
numeric_relational_expr:
        unknown_types_term '<' arithmetic_expr_value
        | arithmetic_expr_value '<' arithmetic_expr_value
        | arithmetic_expr_value '<' unknown_types_term
        | unknown_types_term '>' arithmetic_expr_value
        | arithmetic_expr_value '>' arithmetic_expr_value
        | arithmetic_expr_value '>' unknown_types_term
        | unknown_types_term LETOK arithmetic_expr_value
        | arithmetic_expr_value LETOK arithmetic_expr_value
        | arithmetic_expr_value LETOK unknown_types_term
        | unknown_types_term GETOK arithmetic_expr_value
        | arithmetic_expr_value GETOK arithmetic_expr_value
        | arithmetic_expr_value GETOK unknown_types_term
        | identifier EQTOK arithmetic_expr_value
        | unknown_types_term_value EQTOK arithmetic_expr_value
        | arithmetic_expr_value EQTOK arithmetic_expr_value
        | arithmetic_expr_value EQTOK unknown_types_term
        | unknown_types_term_value NETOK arithmetic_expr_value
        | identifier NETOK arithmetic_expr_value
        | arithmetic_expr_value NETOK arithmetic_expr_value
        | arithmetic_expr_value NETOK unknown_types_term
        ;
```

```
string_relational_expr:
        unknown_types_term '<' string_value
        | string_value '<' string_value
        | string_value '<' unknown_types_term
        | unknown_types_term '>' string_value
        | string_value '>' string_value
        | string_value '>' unknown_types_term
        | unknown_types_term LETOK string_value
        | string_value LETOK string_value
        | string_value LETOK unknown_types_term
        | unknown_types_term GETOK string_value
        | string_value GETOK string_value
        | string_value GETOK unknown_types_term
        | unknown_types_term_value EQTOK string_value
        | identifier EQTOK string_value
        | string_value EQTOK string_value
        | string_value EQTOK unknown_types_term
        | unknown_types_term_value NETOK string_value
        | identifier NETOK string_value
        | string_value NETOK string_value
        | string_value NETOK unknown_types_term
        ;

new_types_relational_expr:
       /* ############## intervals ########## */
        all_interval_value EQTOK all_interval
        | identifier EQTOK all_interval_value
        | all_interval_value NETOK all_interval
        | identifier NETOK all_interval_value
       /* ############## events ########## */
        | all_event_value EQTOK all_event
        | identifier EQTOK all_event_value
        | all_event_value NETOK all_event
        | identifier NETOK all_event_value
        ;

logical_expr:
        identifier
       | logical_expr_value
        ;

logical_expr_value:
       basic_logical_expr
       | iteration_expr
       | new_types_logical_expr
       | '(' logical_expr_value ')'
        ;

basic_logical_expr:
       relational_expr_value
        | '!' logical_expr
        | logical_expr ANDTOK logical_expr
        | logical_expr ORTOK logical_expr
        | TRUETOK
        | FALSETOK
        ;

arithmetic_or_null:
       arithmetic_expr
        | /* NULL */
        ;

new_types_logical_expr:
       all_interval INCLUDETOK single_new_types
        | all_interval INTERSECTTOK single_new_types
        | single_new_types INSIDETOK all_interval
        | all_interval ALWAYSTOK logical_expr
        | all_interval EVENTUALLYTOK logical_expr
        | all_interval BEFORETOK logical_expr
        | all_interval BEGINTOK logical_expr
        | all_interval LASTTOK logical_expr
```

```
        | all_interval AFTERTOK logical_expr
        | logical_expr IMPLYTOK logical_expr
        | all_event ISBEFORETOK'['arithmetic_or_null','
                                 arithmetic_or_null ']' all_event
        | all_event ISBEFORETOK '[' arithmetic_or_null ','
                                 arithmetic_or_null ')' all_event
        | all_event ISBEFORETOK '(' arithmetic_or_null ','
                                 arithmetic_or_null ']' all_event
        | all_event ISBEFORETOK '(' arithmetic_or_null ','
                                 arithmetic_or_null ')' all_event
        ;


iteration_expr:
        FORALLTOK identifier ':' identifier '{' logical_expr '}'
        | FORALLTOK identifier ':' all_interval_set_value '{' logical_expr '}'
        | FORALLTOK identifier ':' all_event_set_value '{' logical_expr '}'
        | EXISTSTOK identifier ':' identifier '{' logical_expr '}'
        | EXISTSTOK identifier ':' all_interval_set_value '{' logical_expr '}'
        | EXISTSTOK identifier ':' all_event_set_value '{' logical_expr '}'
         /* JLF: extensions */
        | FORALLTOK identifier  ':' arithmetic_expr_value '{' logical_expr '}'
        | EXISTSTOK identifier ':' arithmetic_expr_value '{' logical_expr '}'
        ;

statement:
         expression_statement
        | compound_statement
        | assignment_statement
        | EXITTOK
        ;

expression_statement:
        ';'
        | logical_expr ';'
        ;

assignment_expr:
        identifier
        | event_value
        | event_set_value
        | ev_se_value
        | interval_value
        | interval_set_value
        | in_se_value
        | arithmetic_expr_value
        | logical_expr_value
        | message_single_value
        | message_value
        ;

assignment_statement:
          identifier '=' assignment_expr ';'
        ;

compound_statement:
         '{' '}'
        | '{' statement_list '}'
        ;

statement_list:
         statement
        | statement_list  statement
        ;

program:
        function
        ;
```

```
function:
          function statement
        | function EOFTOK
        | /* NULL */
        ;

single_new_types:
        identifier
        | event_value
        | ev_se_value
        | interval_value
        | in_se_value
        ;

/***************************************
            EVENT and EVENT SETS
***************************************/
event_expr:
         identifier
        | event_expr_value
        ;

event_expr_value:
         event_value
        | event_set_value
        | ev_se_value
        ;
/***************************************
            EVENT SET DEFINITION
***************************************/
all_event_set_value:
        event_set_value
        | ev_se_value

event_set_value:
        base_event_set
        | extend_event_set
        | '(' event_set_value ')'
        ;

base_event_set:
         MSGTOK
        | STARTTOK '(' interval_set_value ')'
        | ENDTOK '(' interval_set_value ')'
        ;

extend_event_set:
        extend_forward_event_set
        | extend_backward_event_set
        ;

extend_forward_event_set:
        base_event_set EXTEND_FTOK arithmetic_expr
        ;


extend_backward_event_set:
         arithmetic_expr EXTEND_BTOK base_event_set
        ;

/***************************************
            EVENT DEFINITION
***************************************/
all_event:
        identifier
        | all_event_value
        ;

all_event_value:
        event_value
        | ev_se_value
```

```
event_value:
        base_event
        | extend_event
        | '(' event_value ')'
        ;

base_event:
        STARTTOK '(' interval_value ')'
        | ENDTOK '(' interval_value ')'
        ;

extend_event:
        extend_forward_event
        | extend_backward_event
        ;

extend_forward_event:
        base_event EXTEND_FTOK arithmetic_expr
        ;

extend_backward_event:
         arithmetic_expr EXTEND_BTOK base_event
        ;

/*****************************************
  DEFINITIONS that can be an EVENT or EVENT SETS
*****************************************/
ev_se_value:
        base_ev_se
        | extend_ev_se
        | '(' ev_se_value ')'
        ;

base_ev_se:
        STARTTOK '(' identifier ')'
        | ENDTOK '(' identifier ')'
        ;

extend_ev_se:
        extend_forward_ev_se
        | extend_backward_ev_se
        ;

extend_forward_ev_se:
        forward_ev_se EXTEND_FTOK arithmetic_expr
        ;

forward_ev_se:
        base_ev_se
        | identifier
        ;

extend_backward_ev_se:
         arithmetic_expr EXTEND_BTOK backward_ev_se
        ;

backward_ev_se:
         base_ev_se
        | identifier
        ;

/*************************************
            INTERVAL and INTERVAL SETS
*************************************/

interval_expr:
         interval_set
        | interval_value
        | in_se_value
        ;
```

```
/****************************************
            INTERVAL SET DEFINITION
****************************************/

all_interval_set:
        identifier
        | all_interval_set_value
        ;

interval_set:
        identifier
        | interval_set_value
        ;

all_interval_set_value:
        interval_set_value
        | in_se_value


interval_set_value:
        base_interval_set
        | search_interval_set
        | conditional_interval_set
        | interval_set_expr
        | '(' interval_set_value ')'
        ;

base_interval_set:
        CAST_INTVTOK '(' all_interval ')'
        | '[' logical_expr_value ']'
         ;

search_interval_set:
        search_forward_interval_set
        | search_backward_interval_set
        ;

search_forward_interval_set:
        forward_interval_set SEARCH_FTOK event_expr
        | forward_interval_set SEARCH_FTOK
        ;

forward_interval_set:
        search_forward_interval_set
        | base_interval_set
        | event_set_value
        ;

search_backward_interval_set:
         event_expr SEARCH_BTOK backward_interval_set
        | SEARCH_BTOK backward_interval_set
        ;

backward_interval_set:
        search_backward_interval_set
        | base_interval_set
        | event_set_value
        ;

conditional_interval_set:
        identifier ':' interval_set IFFTOK logical_expr
        ;

interval_set_expr:
        interval_expr UNIONTOK interval_expr
        | interval_expr ORITOK interval_expr
        | interval_expr ANDITOK interval_expr
        | interval_expr SUBSTOK interval_expr
        ;
```

```
/****************************************
            INTERVAL DEFINITION
****************************************/
all_interval:
        identifier
        | all_interval_value
        ;

interval:
        identifier
        | interval_value
        ;

all_interval_value:
        interval_value
        | in_se_value

interval_value:
        base_interval
        | search_interval
        | '(' interval_value ')'
        ;

base_interval:
         NONETOK
        ;

search_interval:
        search_forward_interval
        | search_backward_interval
        ;

search_forward_interval:
        forward_interval SEARCH_FTOK event_expr
        | forward_interval SEARCH_FTOK
        | SEARCH_FTOK event_expr
        | SEARCH_FTOK
        ;

forward_interval:
        search_forward_interval
        | base_interval
        | event_value
        ;

search_backward_interval:
         event_expr SEARCH_BTOK backward_interval
        | event_expr SEARCH_BTOK
        | SEARCH_BTOK backward_interval
        | SEARCH_BTOK
        ;

backward_interval:
        search_backward_interval
        | base_interval
        | event_value
        ;

/****************************************
  DEFINITIONS that can be an INTERVAL or INTERVAL SETS
****************************************/
in_se_value:
         search_in_se
        | '(' in_se_value ')'
        ;

search_in_se:
        search_forward_in_se
        | search_backward_in_se
        ;
```

```
search_forward_in_se:
        forward_in_se SEARCH_FTOK event_expr
        | identifier SEARCH_FTOK event_expr
        | forward_in_se SEARCH_FTOK
        | identifier SEARCH_FTOK
        ;

forward_in_se:
        search_forward_in_se
        | ev_se_value
        ;

search_backward_in_se:
         event_expr SEARCH_BTOK backward_in_se
        | SEARCH_BTOK backward_in_se
        ;

backward_in_se:
        search_backward_in_se
        | identifier
        | ev_se_value
        ;
```

Identifiers

An identifier is the name of a variable (storing a numeric or string value, event or interval), log entry (event) or name of data associated to the log entry. Can be defined as in C as a sequence of characters and digits where the first character has to be a letter.