Verifying quantitative real-time properties of synchronous programs

M. Jourdan, F. Maraninchi and A. Olivero
{jourdan|maraninchi|olivero}@imag.fr

VERIMAG, B.P. 53X 38041 Grenoble Cedex - FRANCE

Abstract. We propose to apply the verification techniques available for Timed Graphs [ACD90], and particularly the symbolic model-checking algorithm of [HNSY92], to the Argos [Mar92] synchronous language. We extend the language with a single delay construct that allows to express watchdogs and timeouts, at any level in the parallel or hierarchic structure of a program. We define the semantics of this extended language in terms of Timed Graphs, and show that it is a "convenient" extension of the pure Argos synchronous semantics. Indeed, for discrete time, the two semantics coincide. The Timed Graph semantics can be viewed as a continuous time semantics for Argos. We extend the Argos compiler and connect it to the KRONOS tool which implements the abovementioned model-checking algorithm.

1 Introduction

The synchronous languages for real-time systems, like Lustre [CHPP87], Esterel [BG88, BG92], Signal [GBBG85], Statecharts [Har87] and Argos [Mar92] traditionally deal with *multiform time*. In particular, time is seen as an ordinary input of the system, which may count meters as well as seconds. As a consequence, a timeout structure implies counting the occurrences of a particular input event. The synchronous languages are compiled into labeled transition systems. Verification may be performed on this compiled form, applying model checking techniques, or abstractions and comparisons modulo a well chosen equivalence relation. However, the kind of property that may be verified in this context deals with *qualitative* aspects of time only, i.e. with ordering of inputs and outputs. Nothing can be expressed about the actual values of the delays, for instance. Indeed, there are essentially two ways of compiling a timeout structure: the different values of the counter, up to the bound used in the program, can be expanded into states; or the counter may be compiled into a runtime variable. The first solution gives rise to state explosion due to the values of the delay bounds; the second solution solves this problem, but the usual verification techniques cannot deal with runtime variables, and can only be used to verify time-independent properties on the control structure of the program.

On the other hand, *Timed Graphs* [ACD90] have been introduced, that allow to represent time in automata, in a very concise way. A Timed Graph is an automaton, extended with a finite set of real-valued clocks. Verification may be performed on Timed Graphs using model checking techniques. An algorithm is proposed in [ACD90]. A *symbolic* model checking algorithm for TCTL [ACD90] (a real-time extension of the branching-time logic CTL) is described in [HNSY92].

Timed Graphs cannot be used directly as a specification formalism, and it is interesting to define the translation of a real-time language or formalism into Timed Graphs. [NSY91] describes the translation of the algebra ATP.

In this paper, we propose to translate Argos into Timed Graphs. This language has a small set of powerful constructs, that allows the description of a complex system as a composition of automata. In particular, a single construct called *refinement* is used to express exceptions, interruptions, suspensions, etc. This construct introduces hierarchy in state/transition-based descriptions of reactive systems, and is related to the notion of macro-state in Statecharts. The ability of this single construct to describe all these control structures strongly relies on the *synchronous* communication mechanism between the components involved in a refinement.

By allowing *delays* to be attached to the states of the basic automaton components, we allow the expression of timeouts and watchdogs and a readable merge of these constructs with the other control structures of the language.

The delay construction could be introduced as a macro-notation, which means that its semantics would be defined formally by the syntactic translation of a temporized Argos program into a "pure" Argos program. This could be done by adding a *timer* component for each temporized state. A timer component is an automaton which has one state for each possible value of the counter (between 0 and the bound which appears in the program). The original program has to be modified, in order to synchronize the timer components with the entering and leaving of the temporized states. A timer counts the occurrences of a special input event. But this makes sense only if we consider *discrete time*.

The aim of this paper is to define the structural translation of a temporized Argos program into a Timed Graph, applying a slightly modified version of the usual Argos semantics. The idea is to define the translation of a basic automaton component with temporized states into a Timed Graph, and then to define the semantics of the Argos constructs as operators on timed graphs. The Timed Graph is built in such a way that a discrete interpretation of it would give a labeled transition system B_2, and B_2 is bisimilar to B_1 produced by the usual Argos semantics for the pure program where the macro-notations are expanded.

Forgetting about the discrete time semantics of Argos, we can see the macro-notation as a built-in feature, and take the semantics in terms of Timed Graphs as the definition of the language.

Section 2 recalls the constructs of the Argos language. Section 3 gives the formal semantics of pure Argos, the semantics in terms of Timed Graphs, and the comparison between the two. Section 4 deals with the verification of timing properties. The Argos compiler has been modified to allow the generation of Timed Graphs, and we use the KRONOS tool [NSY92] based upon the ideas of [HNSY92] to verify timing properties expressed in TCTL. Section 5 is the conclusion.

2 The Argos constructs

In this section, we recall the constructs of the Argos language, by commenting a simplified version of the reflex game system. The complete description of the example may be found in [BG92]. A more detailed presentation of the Argos language may

be found in [Mar92]. Detailing the behaviour of the program and the semantics of the Argos constructs is beyond the scope of this paper. Section 3 gives a simplified version of the Argos semantics (without error detection).

The program of Fig. 1 describes a system whose inputs are: `onoff`, `coin`, `ready`, `stop`, `go` and whose outputs are: `lampON`, `lampOFF`, `flashTILT`, `FlashGO`, `inG`, `outG`. The machine has to be started with input `onoff`. Then a coin must be inserted (input `coin`). Then two users may play a reflex game: when the first player (P1) is ready, he presses `ready`. (If he does not press it within L time units, the game ends). A lamp at the back of the machine is switched on, and the second player (P2) has to press the `go` button. (If he does not press it within `L'` time units, the machine reacts as if he had pressed it). When P2 presses the go button, another lamp flashes at the front of the machine, and P1 must press the `stop` button as fast as he can. Then the game ends. (It also ends if P1 does not press `stop` within `L''` time units).

If P1 presses `stop` *before* the `go` lamp flashes, or exactly at the same instant, the game ends and the `tilt` lamp flashes. In the Argos program of figure 1, the reader should first ignore the events `inG`, `outG`, which will be explained in 4. Moreover, the program does not describe the *reflex time measure* itself, but only the control structure of the game.

The main structure of the program is a two-state automaton: `OFF` and `ON`. The `ON` state is *refined* by another two-state automaton: `GameOver` and `GameOn`. Finally, the `GameOn` state is refined by the *parallel composition* of two automata. These two components represent the two players. `start`, `end`, `exit` and `error` are *local* to the process refining `ON`.

The transition labels are of the form: input/output. Negation of events is denoted by overlining. The Argos communication mechanism is the *synchronous broadcast* of Esterel. When the system reacts to an external input, it may give rise to several transitions, in different automata, which are simultaneous. The components of a parallel composition or refinement operation may communicate by synchronous broadcast, if the output of one of them is the input of another one.

When the system enters a refined state, the refining subprogram is started in its initial state (denoted with an arrow without source). Leaving a refined state `X` kills the refining program `P`. It may be done by an *interruption*, or by the *termination* of `P`. In this case, `P` emits an event to which the automaton reacts by leaving `X`.

The `ON` state can only be left with input `onoff`. This event behaves as an *interruption* of the whole `ON` behaviour.

The `GameOn` state may be left in three ways: either the game ends normally, after P1 has pressed `stop`: one of the parallel components emits `exit`; or the same component emits `error`, in one of the error situations listed above. In these two cases, the `GameOver/GameOn` automaton leaves state `GameOn` and emits `lampOFF` to switch the lamp off. In case of an error, it also emits `flashTILT`. Finally, `GameOn` may be left because the machine is switched off with input `onoff`.

The timed construct: the state `WR` (resp. `WS`, `Wait`) is *temporized* by the delay `L` (resp. `L''`, `L'`) written between brackets. It has an outgoing transition whose label is replaced by a square box. This is the *timeout* transition. The intuitive semantics is as follows: once a temporized state is entered, it *must* be left before the indicated amount of time has elapsed. The program can leave the state by taking a "normal" transition, or has to take the special *timeout* transition, when the delay expires. The

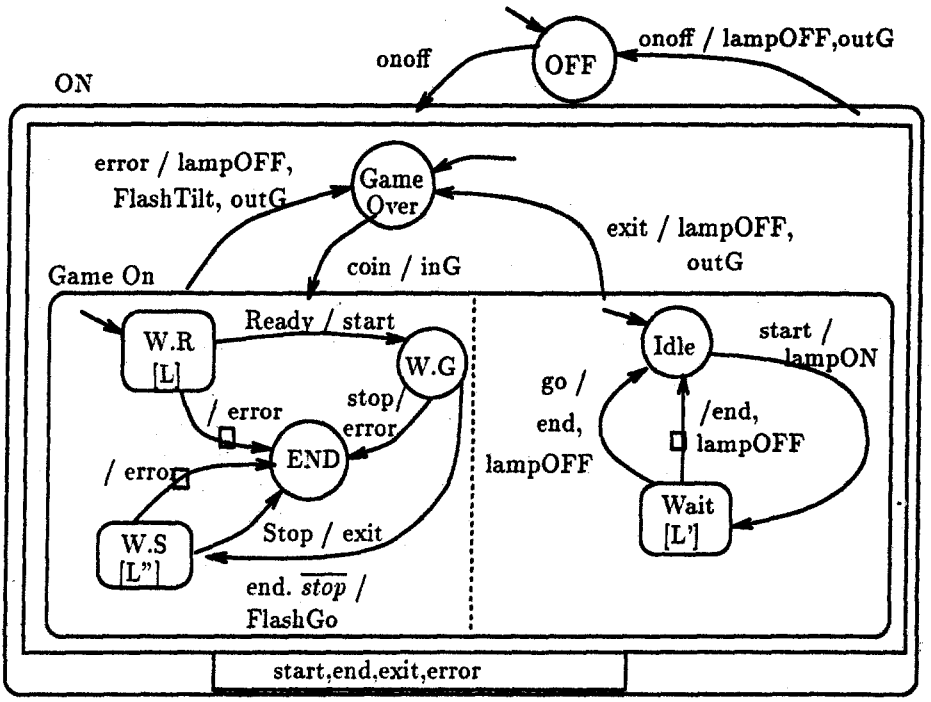


Fig. 1. The reflex game

example shows *timeouts* only but the timed construct may be used to implement a variety of timed constructs taken from other languages. A *watchdog* of delay d on a subprogram P is described by refining a state X by P and temporizing X by d.

3 From temporized Argos programs to Timed Graphs

We first give the semantics of "pure" Argos, in terms of input/output-labeled transition systems. Then we extend this model with *timers*. We obtain a form of Timed Graphs where the transition labels have additional *actions*, as in [NSY91]. These actions are used to apply the semantics of the Argos contructs to Timed Graphs.

3.1 Syntax

We consider a global set of *events* $\alpha = \{a, b, c, ...\}$. $\mathcal{M}(\alpha)$ is the set of *monomials* with variables in α. Negation is denoted by overlining and, for a monomial $m \in \mathcal{M}(\alpha) = a_1 \wedge a_2 \wedge ... \wedge a_n \wedge \overline{b_1} \wedge \overline{b_2} \wedge ... \wedge \overline{b_p}$ we denote by m^+ the set $\{a_1, a_2, ...a_n\}$ and by

m^- the set $\{b_1, b_2, ..., b_p\}$. We write $m \neq \texttt{false}$ iff $m^+ \cap m^- = \emptyset$. The *restriction* $m[X]$ of a monomial m to a subset $X \subseteq \alpha$ is defined by: $(m[X])^+ = m^+ \cap X$ and $(m[X])^- = m^- \cap X$.

"Pure" Argos programs

An *automaton* is a tuple $A = (Q, q_0, T)$ where Q is a set of *boxes*, q_0 is the *initial* box and $T \subseteq Q \times \mathcal{M}(\alpha) \times 2^\alpha \times Q$ is the set of *transitions*. A program $P \in \mathcal{P}$ is of the form: $P ::= \mathbf{R}_A(R_1, ..., R_n) \mid P \| P \mid \overline{P^Y}$. $R ::= P \mid$ NIL. Each automaton $A = (Q, q_0, T)$ defines a parameterized refinement operator $\mathbf{R}_A()$, whose arity is $|Q|$. To relate a box in Q to its refining program, we identify the set Q to the interval $[1..|Q|]$. Then $\mathbf{R}_A(R_1, ..., R_n)$ is a program made with the automaton A, the box i being refined by R_i. The R_i may be programs or the special value NIL, which means that the box is not refined. $\|$ denotes parallel composition; overlining denotes local event declaration ($Y \subseteq \alpha$).

Temporized Argos programs

The automata of a temporized Argos program are of the form: $(Q, q_0, T, \mathcal{T})$, where Q is the set of boxes, q_0 is the initial box, $T \in Q \times \mathcal{N}(\alpha) \times 2^\alpha \times Q$ is the set of transitions. The input part of the label is an element of $\mathcal{N}(\alpha) = \mathcal{M}(\alpha) \cup \{\texttt{to}\}$ (to for *timeout*). $\mathcal{T} : Q \longrightarrow \mathbb{N}^* \cup \{+\infty\}$ is the function which expresses the temporization of boxes. $\mathcal{T}(q) = +\infty$ if the box is not temporized. The programs have the same syntax as before.

3.2 The semantics of pure Argos programs

We consider a version of ARGOS without error detection. The models we produce may be nondeterministic and/or nonreactive. However, for correct programs, the semantics given below coincide with the one detailed in [Mar92]. The semantics of pure ARGOS programs is given by the function $\mathcal{S}$: $\mathcal{P} \longrightarrow \mathcal{I}olts$ where $\mathcal{I}olts$ is the set of *input/output-labeled transition systems*. Such a system is of the form (Q, q_0, T) where Q is a set of *states*, $q_0 \in Q$ is the *initial* state, $T \in Q \times \mathcal{M}(\alpha) \times 2^\alpha \times Q$ is the set of *transitions*. A transition $t = (q_s, m, o, q_t)$ will be denoted by $q_s \xrightarrow{m/o} q_t$ in the following. $\mathcal{S}$ is defined in a structural way. It is extended to NIL. For a correct program P, the transition system $\mathcal{S}(P)$ is *reactive*, which means: $\forall q \in Q, \forall m \in \mathcal{M}(\alpha), \exists q', o$ s.t. $q \xrightarrow{m/o} q'$ (sometimes $q' = q$ and $o = \emptyset$).

Parallel composition

Let $\mathcal{S}(P_i) = (Q_i, q_{0i}, T_i)$ be the semantics of program P_i, for $i \in \{1, 2\}$.
Then $\mathcal{S}(P_1 \| P_2) = (Q_1 \| Q_2, q_{01} \| q_{02}, T')$ with $Q_1 \| Q_2 = \{q_1 \| q_2, q_1 \in Q_1, q_2 \in Q_2\}$ and T' is defined from T_1, T_2 by the following rule:

$$\frac{q_1 \xrightarrow{m_1/o_1} q_1', \quad q_2 \xrightarrow{m_2/o_2} q_2', \quad m_1 \wedge m_2 \neq \texttt{false}}{q_1 \| q_2 \xrightarrow{m_1 \wedge m_2 / o_1 \cup o_2} q_1' \| q_2'} \qquad \text{[P pure]}$$

The two components always react together. But one of them may take an empty-output loop. Note that, if $\mathcal{S}(P_1)$ and $\mathcal{S}(P_2)$ are reactive, then $\mathcal{S}(P_1 \| P_2)$ is too.

Encapsulation

Let $\mathcal{S}(P) = (Q, q_0, T)$ be the semantics of a program P. Then $\mathcal{S}(\overline{P^Y}) = (\overline{Q^Y}, \overline{q_0^Y}, T')$ with $\overline{Q^Y} = \{\overline{q^Y}, q \in Q\}$ and T' is defined from T by the following rule:

$$\frac{q \xrightarrow{m/o} q', \quad m^+ \cap Y \subseteq o \wedge m^- \cap Y \cap o = \emptyset}{\overline{q^Y} \xrightarrow{m[\alpha-Y]/o-Y} \overline{q'^Y}} \qquad \text{[I pure]}$$

This rule expresses the synchronous broadcast mechanism. Taking a program P whose transition labels contains the elements of Y both as inputs and outputs, it keeps only those transitions which correspond to correct cooperations between the components: the events that are positively tested in the input part have to be emitted ($m^+ \cap Y \subseteq o$) and the events that are negatively tested have not to be emitted ($m^- \cap Y \cap o = \emptyset$). Moreover the elements of Y are hidden.

Refinement

Let $\mathcal{S}(R_i) = (Q_i, q_{0i}, T_i)$ be the semantics of programs $R_1, ..., R_n$.
Then $\mathcal{S}(\mathbf{R}_{(Q,q_0,T)}(R_1, ..., R_2)) = (Q', q', T')$ with:

$$Q' = \{\mathbf{R}_{(Q,q_0,x,T)}(y_1, ..., y_n), x \in Q, y_i \in Q_i \text{ for } i \in [1,n]\}$$
$$q' = \mathbf{R}_{(Q,q_0,q_0,T)}(q_{01}, ..., q_{0n})$$

and T' is defined from $T, T_1, ..., T_n$ by the following rules:

$$\frac{(q_c, m_1, o_1, q_d) \in T, \quad r_c \xrightarrow{m_2/o_2} r_c', \quad m_1 \wedge m_2 \neq \texttt{false}}{\mathbf{R}_{(Q,q_0,q_c,T)}(r_1, ..., r_c, ..., r_n) \xrightarrow{m_1 \wedge m_2/o_1 \cup o_2} \mathbf{R}_{(Q,q_0,q_d,T)}(q_{01}, ..., q_{0n})} \qquad \text{[R1 pure]}$$

$$\frac{\text{nonreact}(q_c, m_2, T), \quad r_c \xrightarrow{m_2/o_2} r_c'}{\mathbf{R}_{(Q,q_0,q_c,T)}(r_1, ..., r_c, ..., r_n) \xrightarrow{m_2/o_2} \mathbf{R}_{(Q,q_0,q_c,T)}(r_1, ..., r_c', ..., r_n)} \qquad \text{[R2 pure]}$$

where: $\text{nonreact}(q_c, m_2, T) \iff \nexists\ m_1$ s.t. $(\exists q_d, o_1, (q_c, m_1, o_1, q_d) \in T$ and $m_1 \wedge m_2 \neq \texttt{false})$ and $\mathcal{S}(\text{NIL}) = (\{\text{NIL}\}, \text{NIL}, T)$ where T is defined by: $\text{NIL} \xrightarrow{m/\emptyset} \text{NIL}$.

A state of the behaviour has the form: $q ::= q \| q \mid \overline{q^Y} \mid \mathbf{R}_{(Q,q_0,q_c,T)}(r_1, ..., r_n)$.
$r ::= q \mid \text{NIL}$. A box B is said to be *active* in a state q iff $active(B, q)$:

$$\begin{aligned}
active(B, q_1 \| q_2) &= active(B, q_1) \vee active(B, q_2) \\
active(B, \overline{q^Y}) &= active(B, q) \\
active(B, \mathbf{R}_{(Q,q_0,q_c,T)}(r_1, ..., r_c, ..., r_n)) &= (B = q_c) \vee active(B, r_c) \\
active(B, \text{NIL}) &= \texttt{false}
\end{aligned}$$

All states of the behaviour are such that the subprograms refining inactive states are set to their initial state. (this is due to the general reinitialization of the refining subprograms in [R1 pure]). Hence the states are built in a canonical form, and the behaviour is minimal in this sense.

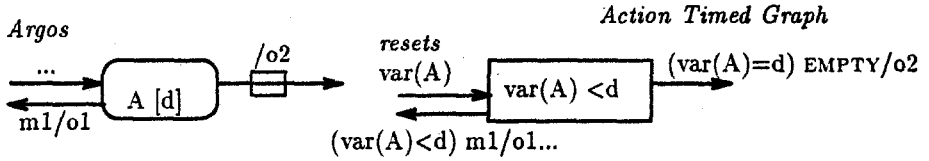
3.3 The semantics of temporized Argos programs in terms of Timed Graphs

The semantics of temporized ARGOS programs in terms of Timed Graphs is given by the function $\mathcal{S}'$: $\mathcal{P}_t \longrightarrow \mathcal{ATG}$ where $\mathcal{P}_t$ is the set of *temporized Argos programs* and $\mathcal{ATG}$ is the set of *Action Timed Graphs*. The *Action Timed Graphs* we use here are of the form $(Q, q_0, T, \mathcal{F}, X)$ where Q is a set of *nodes*, $q_0 \in Q$ is the *initial* node, X is a set of integer variables called *timers*, $T \subseteq Q \times \mathcal{C}(X) \times \mathcal{M}(\alpha) \cup \{\text{EMPTY}\} \times 2^{\alpha} \times \mathcal{RA}(X) \times Q$ is the set of *transitions*. $\mathcal{C}(X)$ is the set of boolean expressions built from the variables of X following the grammar: $c ::= x < k \mid x = k$ where k is an nonnegative integer constant and $x \in X$ (timed graphs allow more general conditions). $\mathcal{RA}(X)$ is the set of *reset actions*. A reset action $r \in \mathcal{RA}(X)$ is a subset of X, which contains the variables to be reset when the transition is taken. The other variables are unchanged. A transition $t = (q_s, C, m, o, r, q_t)$ will be denoted by $q_s \xrightarrow{C.\ m/o\ .r} q_t$ in the following. $\mathcal{F} : Q \longrightarrow \mathcal{C}(X)$ gives the labeling of nodes by *invariant* properties.

Principle of the translation

We consider a global set of real variables $\mathcal{X}$. We suppose that all the boxes of the basic automaton components are distinct and that there exists a one-to-one function *var* which associates an element of $\mathcal{X}$ to each automaton box. Conversely, $box(x)$, for $x \in \mathcal{X}$. gives the automaton box to which the timer x is attached. (When a box is not temporized, the bound is $+\infty$. This will introduce conditions of the form $x < +\infty$ (resp. $x = +\infty$) in the timed graph. These conditions are replaced by `true` (resp. `false`) and the variable x never appears in the timed graph.)

The idea is to do the following, for each temporized box:



The input part of a transition label may be $m \in \mathcal{M}(\alpha)$ or a special input denoted by EMPTY. In the sequel, a transition is said to be *temporal* if the input part of its label is EMPTY, *nontemporal* otherwise. The EMPTY input is used as a marker to identify the timeout transitions which are taken without external input, and to avoid combining them with nontemporal transitions, when applying the semantics of the Argos constructs (see [P tempo] and [R* tempo] below). Note that a temporal transition may have outputs. The $\mathcal{S}'$ semantic function is defined in such a way that $\mathcal{S}(\mathcal{E}xp(P_t)) \equiv \mathcal{D}(\mathcal{S}'(P_t))$ where:

- $\mathcal{E}xp$ is the expansion of the delay construct, considered as a macro-notation: $\mathcal{E}xp(P_t)$ is a *pure* Argos program containing a special "time" event χ;
- We need to compare $\mathcal{S}(\mathcal{E}xp(P_t))$ — which is a labeled transition system containing χ, to $\mathcal{S}'(P_t)$ — which is a timed graph, where time is implicit. For this purpose, we use a *"discrete time"* semantics $\mathcal{D}$ of the Timed Graphs we obtain as models of the temporized Argos programs. For a timed graph atg, $\mathcal{D}(atg)$ will be a labeled

transition system containing χ.

- $\equiv$ is the strong bisimulation.

The expansion of the macro-notation

The macro-notation may be expanded in such a way that the pure Argos program corresponding to P_t is of the form: $\overline{P_m \| C_1 \| C_2 \| ... \| C_n}^{Y}$ where the C_i are *timer components* (one for each temporized box in P_t) and P_m is obtained from P_t by adding some communication with the timer components (the boxes and transitions are not modified. The transition *labels* are modified). The timer component C_X for a box $X[d]$ is the automaton $(Q = [0, d-1], q_0 = 0, T)$ where:
$T = \{(i, \chi \wedge \overline{\texttt{reset}_X}/\emptyset, i+1), (i, \texttt{reset}_X/\emptyset, 0), i < d-1\} \quad \cup$
$\{(d-1, \texttt{reset} \wedge \overline{\chi}/\emptyset, 0), (d-1, \chi/\texttt{to}_X, 0)\}$.
$Y = \{\texttt{reset}_u, \texttt{to}_u, u \in [1, n]\}$. χ is the external event introduced to model discrete time; we consider that the time event never occurs simultaneously with another input event: in the global behaviour, the input part of the transitions is χ *or* $m \in \mathcal{M}(\alpha)$. The timer sends $\texttt{to}_X$ to P_m when it expires. It is reset (with $\texttt{reset}_X$) when the box X to which it is attached becomes *active*. When this box is not active, the timer is not set to an idle value: it loops, counting time events modulo its bound, sending timeout events which are ignored.

The discrete semantics of Action Timed Graphs

We denote by $\vec{X} \in \mathbb{N}^n$ a valuation of the variables in $X = \{x_1, ...x_n\}$. $\vec{X}[i]$ is the value of x_i in $\vec{X}$. For $k \in \mathbb{N}$, $\vec{k} \in \mathbb{N}^n$ is defined by $\vec{k}[i] = k, \forall i \in [1, n]$.
$\mathcal{D}((Q, q_0, T, \mathcal{F}, X)) = (Q \times \mathbb{N}^n, (q_0, \vec{0}), T')$ and T' (denoted by $\longrightarrow$) is defined by:

$$\frac{\mathcal{F}(q)(\vec{X} + \vec{1})}{(q, \vec{X}) \xrightarrow{\chi/\emptyset} (q, \vec{X} + \vec{1})} \qquad \text{[D1]}$$

$$\frac{q \xRightarrow{C.\,\text{EMPTY}/o\,.r} q', C(\vec{X} + \vec{1})}{(q, \vec{X}) \xrightarrow{\chi/o} (q', r(\vec{X} + \vec{1}))} \qquad \text{[D2]}$$

$$\frac{q \xRightarrow{C.\,m/o\,.r} q', m \neq \text{EMPTY}, \quad C(\vec{X})}{(q, \vec{X}) \xrightarrow{m/o} (q', r(\vec{X}))} \qquad \text{[D3]}$$

where $r(\vec{Y})[i] = \begin{cases} 0 \text{ if } x_i \in r \\ \vec{Y}[i] \text{ otherwise} \end{cases}$

The bisimulation

With this macro-expansion and this discrete semantics for timed graphs, we have: $\mathcal{S}(\mathcal{E}xp(P_t)) \equiv \mathcal{D}(\mathcal{S}'(P_t)))$ where $\equiv$ is a *strong bisimulation*. We write $\mathcal{S}(\mathcal{E}xp(P_t)) = (Q_1, q_{01}, T_1)$, $\mathcal{D}(\mathcal{S}'(P_t)) = (Q_2, q_{02}, T_2)$ Recall that $P' = \mathcal{E}xp(P_t)$ has the form $\overline{P_m \| C_1 \| C_2 \| ... \| C_n}^{Y}$, where P_m and P_t have the same *structure*. So the states in Q_1 have the form $\overline{q \| c_1 \| c_2 \| ... \| c_n}^{Y}$ where q can be considered as a configuration of

boxes in P_t. On the other hand, the nodes of $atg = \mathcal{S}'(P_t)$ are the configurations of boxes in P_t. Hence a state in Q_2 has the form $(q, \vec{X})$ where $\vec{X}$ is a valuation of the timers and q is a configuration of P_t. Let $\mathcal{R} \subseteq Q_1 \times Q_2$ be defined by:

$$\mathcal{R}(\overline{q\|c_1\|c_2\|...\|c_n}^{Y}, (q', \vec{X})) \iff q = q' \wedge (active(box(x_i), q) \implies c_i = \vec{X}[i])$$

which means that the states correspond to the same configuration q of P_t and that the valuation $\vec{X}$ and the timer components $C_1, ..., C_n$ coincide on the *relevant* timers, i.e. the ones attached to the active boxes of q. $\mathcal{S}'$ is built in such a way that $\mathcal{R}$ is a bisimulation, i.e. $(q_{01}, q_{02}) \in \mathcal{R}$ and

$$(q_1, q_2) \in \mathcal{R} \implies \begin{cases} q_1 \xrightarrow{m/o} q_1' \implies \exists q_2' \text{ s.t. } q_2 \xrightarrow{m/o} q_2' \text{ and } (q_1', q_2') \in \mathcal{R}. \\ q_2 \xrightarrow{m/o} q_2' \implies \exists q_1' \text{ s.t. } q_1 \xrightarrow{m/o} q_1' \text{ and } (q_1', q_2') \in \mathcal{R}. \end{cases}$$

Translating parallel composition

Let $\mathcal{S}'(P_i) = (Q_i, q_{0i}, T_i, \mathcal{F}_i, X_i)$ be the semantics of program P_i, for $i \in \{1, 2\}$. Then $\mathcal{S}'(P_1\|P_2) = (Q_1\|Q_2, q_{01}\|q_{02}, T', \lambda q_1\|q_2 \bullet \mathcal{F}_1(q_1) \wedge \mathcal{F}_2(q_2), X_1 \cup X_2)$ with $Q_1\|Q_2 = \{q_1\|q_2, q_1 \in Q_1, q_2 \in Q_2\}$ and T' is defined from T_1, T_2 by:

$$\frac{q_1 \xrightarrow{C_1.\ m_1/o_1\ .A_1} q_1', \quad q_2 \xrightarrow{C_2.\ m_2/o_2\ .A_2} q_2', \quad (m_1 = m_2 = \text{EMPTY}) \vee (m_1 \neq \text{EMPTY} \wedge m_2 \neq \text{EMPTY} \wedge (m_1 \wedge m_2 \neq \texttt{false}))}{q_1\|q_2 \xrightarrow{C_1 \wedge C_2.\ m_1 \wedge m_2/o_1 \cup o_2\ .A_1 \cup A_2} q_1'\|q_2'} \quad [\text{Pt}]$$

Note that X_1 and X_2 are disjoint. $C_1 \wedge C_2$ cannot be the `false` function, since the two functions have disjoint variable sets. Temporal (resp. nontemporal) transitions can be taken together. The two cannot be merged.

Translating encapsulation

Let $\mathcal{S}'(P) = (Q, q_0, T, \mathcal{F}, X)$ be the semantics of a program P. Then $\mathcal{S}'(\overline{P^Y}) = (\overline{Q^Y}, \overline{q_0}^Y, T', \lambda \overline{q}^Y \bullet \mathcal{F}(q), X)$ with $\overline{Q^Y} = \{\overline{q}^Y, q \in Q\}$ and T' is defined from T by:

$$\frac{q \xrightarrow{C.\ m/o\ .A} q', \quad m \neq \text{EMPTY}, \quad m^+ \cap Y \subseteq o \wedge m^- \cap Y \cap o = \emptyset}{\overline{q}^Y \xrightarrow{C.\ m[\alpha - Y]/o - Y\ .A} \overline{q'}^Y} \quad [\text{It1}]$$

$$\frac{q \xrightarrow{C.\ \text{EMPTY}/o\ .A} q', \quad Y \cap o = \emptyset}{\overline{q}^Y \xrightarrow{C.\ \text{EMPTY}/o\ .A} \overline{q'}^Y} \quad [\text{It2}]$$

Translating refinement

Let $\mathcal{S}'(R_i) = (Q_i, q_{0i}, T_i, \mathcal{F}_i, X_i)$ be the semantics of programs $R_1, ..., R_n$. Then $\mathcal{S}'(\mathbf{R}_{(Q,q_0,T,\mathcal{T})}(R_1, ..., R_2)) = (Q', q', T', \mathcal{F}', X')$ with:

$$\begin{aligned} Q' &= \{\mathbf{R}_{(Q,q_0,x,T,\mathcal{T})}(y_1, ..., y_n), x \in Q, y_i \in Q_i \text{ for } i \in [1, n]\} \\ q' &= \mathbf{R}_{(Q,q_0,q_0,T,\mathcal{T})}(q_{01}, ..., q_{0n}) \\ \mathcal{F}' &= \lambda \mathbf{R}_{(Q,q_0,x,T,\mathcal{T})}(y_1, ..., y_n) \bullet (var(x) < \mathcal{T}(x)) \wedge \mathcal{F}_x(y_x) \\ X' &= \bigcup_{i=1}^{n} X_i \cup \{var(x), x \in Q, \mathcal{T}(x) \neq +\infty\} \end{aligned}$$

and T' is defined from $T, T_1, ..., T_n$ by the following rules. [R1at], [R1bt] : both the automaton and the refining subprogram react. In [R1at] the transition of the automaton is not a temporal transition; in [R1bt] it is. As for the parallel composition, temporal and nontemporal transitions cannot be merged.

$$\frac{\begin{array}{c}(q_c, n_1, o_1, q_d) \in T, \quad r_c \xrightarrow{C_2.\ m_2/o_2\ .A_2} r'_c, \\ m_2 \neq \text{EMPTY} \wedge (n_1 \wedge m_2) \neq \texttt{false} \wedge n_1 \neq \texttt{to}\end{array}}{\begin{array}{l}\mathbf{R}_{(Q,q_0,q_c,T,\mathcal{T})}(r_1,...,r_n) \xrightarrow{C_2 \wedge (var(q_c)<\mathcal{T}(c)).\ n_1 \wedge m_2/o_1 \cup o_2\ .A_2 \cup X_d \cup \{var(q_d)\}} \\ \mathbf{R}_{(Q,q_0,q_d,T,\mathcal{T})}(q_{01},...,q_{0n})\end{array}} \text{[R1at]}$$

$$\frac{(q_c, \texttt{to}, o_1, q_d) \in T, \quad r_c \xrightarrow{C_2.\ \text{EMPTY}/o_2\ .A_2} r'_c}{\begin{array}{l}\mathbf{R}_{(Q,q_0,q_c,T,\mathcal{T})}(r_1,...,r_n) \xrightarrow{C_2 \wedge (var(q_c)=\mathcal{T}(c)).\ \text{EMPTY}/o_1 \cup o_2\ .A_2 \cup X_d \cup \{var(q_d)\}} \\ \mathbf{R}_{(Q,q_0,q_d,T,\mathcal{T})}(q_{01},...,q_{0n})\end{array}} \text{[R1bt]}$$

[R2t] : only the refining subprogram reacts. It can be either a temporal or a nontemporal transition.

$$\frac{(m_2 = \text{EMPTY}) \vee (m_2 \neq \text{EMPTY} \wedge \text{nonreact}(q_c, m_2, T)), r_c \xrightarrow{C_2.\ m_2/o_2\ .A_2} r'_c}{\begin{array}{l}\mathbf{R}_{(Q,q_0,q_c,T,\mathcal{T})}(r_1,...,r_c,...,r_n) \xrightarrow{C_2 \wedge (var(q_c)<\mathcal{T}(c)).\ m_2/o_2\ .A_2} \\ \mathbf{R}_{(Q,q_0,q_c,T,\mathcal{T})}(r_1,...,r'_c,...,r_n)\end{array}} \text{[R2t]}$$

where: $\text{nonreact}(q_c, m_2, T) \iff \nexists m_1$ s.t. $(\exists q_d, o_1, (q_c, m_1, o_1, q_d) \in T$ and $m_1 \wedge m_2 \neq \texttt{false})$

$\mathcal{S}'(\text{NIL}) = (\{\text{NIL}\}, \text{NIL}, T, \lambda x \bullet \text{NIL}, \emptyset)$, where T is defined by: NIL $\xrightarrow{\texttt{true.}\ m/\emptyset\ .\emptyset}$ NIL

4 Verification of real-time properties

We propose to verify the following *quantitative* real-time property on the reflex game program: a reflex measure never takes more than $L + L' + L''$ units of time. Expressed in terms of program boxes, the property is: *the system never spends more than $L + L' + L''$ units of time in box* GameOn.

inG and outG have been added to make the entering and leaving of the GameOn box *observable* in the global behaviour of the program. To obtain the labeled Timed Graph of Fig. 2, the temporized reflex game has been compiled into a Timed Graph, according to the $\mathcal{S}'$ semantic function, and then *abstracted*. All transition labels whose output sets contain inG (resp. outG) have been replaced by the atomic label inG (resp. outG). Note that no transition outputs both of them. The other transitions are labeled with an *invisible* action tau. In TCTL, the property may be expressed by: $\forall\Box(\texttt{after(inG)} \Longrightarrow \forall\Diamond_{\leq L+L'+L''}\texttt{after(outG)})$ where after(x) characterizes the set of states reached by executing a transition labeled by x.

The Argos compiler [MV92] has been extended to implement the $\mathcal{S}'$ semantics. A prototype without error detection may be used to connect the ARGONAUTE development environment to the KRONOS tool. To verify the above property, one has

to replace the constants L, L' and L'' by actual integer values, and to write the property in an appropriate syntax. KRONOS takes the timed graph and the property as its inputs, and produces a symbolic description of the set of states which verify the property. The main point is that the complexity of the computation does not depend on the actual values of the constants.

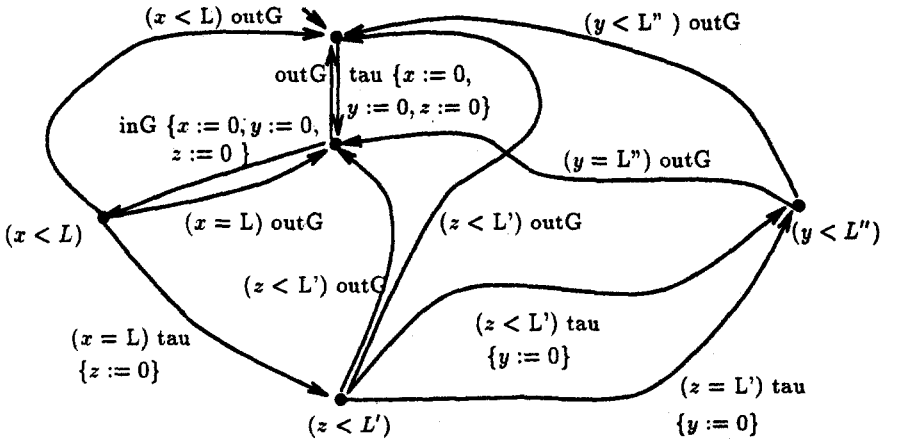


Fig. 2. The abstracted Timed Graph of the reflex game

Remark: KRONOS assumes that time is continuous. It is adequate for proving properties when the delay construct of Argos is considered to be a new essential feature: $\mathcal{S}'$ may be viewed as a *continuous time semantics* for Argos. However, if we consider the delay construct as a macro-notation, its interpretation is *discrete*. The properties proved for continuous time are not necessarily true for discrete time, and the user should be aware of this problem when thinking of the delay construct as a *discrete* time construct. However, for an interesting class of properties called *safety* properties, a property which is true for continuous time is also true for discrete time. The property we gave as an example is a safety property.

5 Conclusion

The Argos synchronous language for real-time systems may be viewed as a set of constructs that allows to describe a complex system as a composition of automata. A single construct is used to introduce hierarchy. It is used for describing interrupts, exceptions, suspensions... In this paper, we showed that a delay construct can be added to the language in a very simple way. The idea is to associate time bounds with the states of the basic automaton components. This simple extension allows the description of timeouts and watchdogs, merged with the other control structures of the language in a very clean way. The semantics of this extended Argos is a slightly modified version of the semantics presented in [Mar92]. The models are Timed Graphs, and the Argos constructs are extended to such objects. The Argos

compiler has been extended in order to produce Timed Graphs, and the KRONOS tool may be used to verify quantitative real-time properties on Argos programs.

Although Argos, being a synchronous language, naturally deals with multiform time, the approach presented here only deals with monoform time. One of the time scales used in a program is distinguished, and used to apply the delay construct. Even if the approach cannot be extended to multiform time in a simple way, due to the kind of algorithms applied to timed graphs, the benefit for a development environment like ARGONAUTE is unquestionable. For an approach which allows to deal with multiform time, see *"Delay analysis in synchronous programs"* by N. Halbwachs, same volume.

References

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the fifth annual IEEE symposium on Logics In Computer Science*, pages 414–425, Philadelphie, PA, USA, June 1990.

[BG88] G. Berry and G. Gonthier. *The ESTEREL Synchronous Programming Language: Design, Semantics, Implementation.* Technical Report, 842, INRIA, 1988.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: design, semantics, implementation. *Science Of Computer Programming*, 19(2):87–152, 1992.

[CHPP87] P. Caspi, N. Halbwachs, D. Pilaud, and J. Plaice. LUSTRE, a declarative language for programming synchronous systems. In *14th Symposium on Principles of Programming Languages*, january 1987.

[GBBG85] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gauthier. *Signal: A Data Flow Oriented Language for Signal Processing.* Technical Report, IRISA report 246, IRISA, Rennes, France, 1985.

[Har87] D. Harel. Statecharts : a visual approach to complex systems. *Science of Computer Programming*, 8:231–275, 1987.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *LICS'92*, June 1992.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*, LNCS 630, Springer Verlag, august 1992.

[MV92] F. Maraninchi and M. Vachon. An experience in compiling a mixed imperative/declarative language for reactive systems. In *International Workshop on Compiler Construction (poster session)*, Springer Verlag, LNCS 641, october 1992.

[NSY91] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to Timed Graphs and Hybrid Systems. In J.W. de Bakker, C. Huizing, W.P. de Roever, and G. Rozenberg, editors, *LNCS 600, proceedings of REX Workshop "Real-Time: Theory in Practice"*. Mook, The Nederlands, Springer Verlag, June 1991.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into timed automata. *IEEE Transactions on Software Engineering, special issue on Specification and Analysis of Real-Time Systems*, 1992.