Compiling Timed Algebras into Timed Automata

Sergio Yovine
Laboratoire de Génie Informatique, Institut IMAG
B.P. 53X, 38041 Grenoble cedex, France
e-mail: sergio@vercors.imag.fr

Abstract

Real-time systems are hard to model, specify and design. It is common practice today to pass from informal specifications to implementations and then test the behavior of the system on some cases, which is inadequate to develop bug-free real-time systems. To ensure correctness we need to employ automatic verification methods and tools. The following real-time software development process has been proposed in [7]: ATP is used to describe real-time systems and TCTL to specify the real-time requirements. The verification approach consists in translating ATP to timed automata and then performing model-checking. In this work we present a tool that compiles ATP terms into timed automata.

Keywords: real-time systems, process algebras, temporal logics, timed automata.

1 Introduction

Typical real-time applications are control systems (e.g., flight controllers, manufacturing systems), monitoring systems (e.g., air traffic, patient monitoring) and communication systems (e.g., communication protocols). The correctness of these systems is highly dependent on the timing delays of the components. Because of the intricacies of the timing relationships, real-time systems are quite hard to model, specify and design. Consequently, there is a great demand for formal methods applicable to real-time systems.

Developing formal methods for the design and analysis of real-time systems is a very active area of current research. Several different formalisms have been proposed. These include timed Petri Nets [4], timed process algebras [9, 10, 6], real-time temporal logics [2, 5] and timed automata [1]. Moreover, there is a pressing need to develop efficient tools to be embedded in a real-time software development process.

Timed algebras are process algebras extended with a set of constructs to express timing requirements. They have formal semantics, given in terms of transition systems or sets of traces, and are a quite adequate to describe the behavior of real-time systems. ATP [6] is a timed process algebra which introduces a mechanism to delay the execution of an action and provides two language constructs, namely timeouts and watchdogs, to express timing constraints.

Timed automata have been proposed in [1, 8] to model finite-state real-time systems. Essentially, timed automata are automata extended with a finite set of real-valued variables called timers. Timing constraints are expressed by associating predicates over timers to transitions. In [8] timed automata are extended with ATP operators and a semantics is provided accordingly. In fact, this motivates an abstract formal method for translating ATP terms into timed automata.

Temporal logic has been extensively used as a formalism for specifying the behavior of concurrent systems [3]. Various real-time extensions of CTL, all called TCTL, have been proposed [1, 5]. The main result in [1] is a model-checking algorithm to check whether a timed automaton satisfies a TCTL formula. In [5] a symbolic model-checking algorithm has been presented.

The following real-time software development process has been proposed in [7]: ATP is used to describe real-time systems and TCTL to specify the real-time requirements. The verification approach consists in translating ATP to timed automata and then performing model-checking. In this work we present a tool that compiles ATP terms into timed automata.

This paper is organized as follows. In sections 2 and 3 we give a brief summary of ATP and timed automata, respectively. In Section 4 and 5 we describe the essential features of the tool.

2 ATP

In this section we define the syntax and semantics of ATP [6, 8]. We assume that a real-time system is the composition of communicating processes. A system may evolve either by *performing an action*, that is a discrete state change, or by *letting time pass*. Let Act be a set of names for actions. The communication between actions is specified by a function $\cdot|\cdot : Act_\perp \times Act_\perp \to Act_\perp$, where $\perp \notin Act$ and $Act_\perp = Act \cup \{\perp\}$. For $a_1, a_2 \in Act$, $a_1|a_2 \in Act$ is the action resulting from the communication of a_1 and a_2. If $a_1|a_2 = \perp$, then the actions cannot interact.

Time is modeled as a global state variable ranging over a *time domain* D. Some examples of time domains are $\mathbb{N}$ (discrete-time), $\mathbb{Q}^+$ and $\mathbb{R}^+$ (dense-time). An important assumption guaranteeing consistency is that time progresses synchronously for all the processes.

Definition 2.1 The syntax of the algebra is the following:

$$P ::= \texttt{idle} \mid Y \mid aP \mid \texttt{delay}(a)P \mid P + P \mid P\ \texttt{timeout}(d)\ P \mid P\ \texttt{watchdog}(d)\ P \mid P||P \mid \texttt{restrict}\ H\ \texttt{in}\ P \mid \texttt{rec}Y.P$$

where $Y \in Var$, $a \in Act$, $d \in D - \{0\}$ and $H \subseteq Act$. We consider only closed terms. ■

We informally explain the meaning of the operators of the language. A formal operational semantics is given in [6, 8].

1. `idle` denotes the process that does nothing but letting time pass.

$$
\begin{array}{rcl}
System & = & \texttt{restrict}\ data_1!, data_1?, data_2!, data_2?\ \texttt{in}\ Sens_1 || Sens_2 || Mon \\
Sens_1 & = & sample_1 \\
 & & ((\texttt{delay}(data_1!)\ \texttt{idle})\ \texttt{timeout}(2)\ data_1!\ \texttt{idle})\ \texttt{watchdog}(10)\ Sens_1 \\
Sens_2 & = & sample_2 \\
 & & ((\texttt{delay}(data_2!)\ \texttt{idle})\ \texttt{timeout}(4)\ data_2!\ \texttt{idle})\ \texttt{watchdog}(10)\ Sens_2 \\
Mon & = & \texttt{delay}(data_1?)\ \texttt{delay}(data_2?)\ Comp \\
 & & + \\
 & & \texttt{delay}(data_1?)\ \texttt{delay}(data_2?)\ Comp \\
Comp & = & \texttt{idle}\ \texttt{timeout}(1)\ result\ Mon
\end{array}
$$

Figure 1: Simple monitoring system.

2. The *prefixing* construct has two forms. aP performs the action a *immediately* and then behaves as P. In $\texttt{delay}(a)P$ the execution of a can be arbitrarily delayed.

3. The *timeout* is a very common real-time construct. Timeouts are widely used in real-time systems to safeguard one part of the system against malfunctioning of another. A timeout is generated at the end of a period of time in which a certain event has not occurred. The process $P\ \texttt{timeout}(d)\ Q$ behaves as P if P performs an action before a time d. Otherwise, it behaves as Q.

4. Sometimes, the execution of a process needs to be monitored by a *watchdog* timer. The process sets the timer and it should reset it before the timer expires. When the process does not succeed in resetting the timer in time, the process is halted by a signal from the watchdog timer. The process $P\ \texttt{watchdog}(d)\ Q$ behaves as P during a time equal to d. At time d, P is aborted and Q is started.

5. The *non-deterministic choice* $P + Q$ behaves either like P or like Q, but only with respect to actions. In the case of time, we require that $P + Q$ can wait a time d if and only if both P and Q can do so.

6. The *parallel composition* $P||Q$ allows P and Q to proceed independently and, in addition, it allows communication between them. If P can perform $a \in Act$ and Q can perform $b \in Act$, then $P||Q$ can perform the action $a|b \in Act$. Notice that P and Q cannot synchronize when $a|b = \bot$. As above, we require that in the case of time, $P||Q$ can wait a time d if and only if both P and Q can.

7. The construct $\texttt{restrict}\ H\ \texttt{in}\ P$ disallows P to perform actions that belong to H.

Example 2.1 Consider a real-time system whose purpose is to collect data from two sensors, and to compute some function of the data and then to print the result.

Each sensor takes a sample reading every 10 milliseconds and attempts to communicate the data to the monitor. However, the communication must be done within a certain

interval of time, otherwise the data become worthless. This time depends on the sensor. When the communication is accomplished, the sensor sleeps until the next reading.

The monitor waits for both sensors to communicate the data. When the data is received, it takes 1 millisecond to compute the function and then prints the result.

In Figure 1 we show the ATP specification of the system, where the communication function is $data_1!|data_1? = data_1$ and $data_2!|data_2? = data_2$. ■

3 Timed automata

In this section we briefly describe timed automata as they are defined in [8].

We start out with a set X of timers ranging over a time domain D. V denotes the set of timer valuations, i.e., the set $[X \to D]$ of mappings from X to D. For $x \in X$, $x := 0$ is the assignment of the value 0 to x, called reset of x. We denote by $\mathcal{R}$ the set of lists of resets of this type. The set Φ of timing constraints is defined by the following grammar:

$$\phi ::= x \# c \mid x + c \# y + d \mid \neg\phi \mid \phi_1 \wedge \phi_2$$

where $x, y \in X$, $c, d \in \mathbb{N}$ and $\# \in \{<, \leq, >, \geq, =\}$.

An timed automaton is a tuple $G := \langle S, X, s_0, \mathtt{inv}, \to \rangle$where:

S	is a finite set of states,
X	is a set of timers,
s_0	is the initial state,
$\mathtt{inv} : S \to \Phi$	associates a timing constraint to each state,
$\to \subseteq S \times (\Phi \times Act \times \mathcal{R}) \times S$	is the set of transitions.

A timed automaton starts at state s_0 with all its timers initialized to zero. The states of the automaton represent the control states. Moving through an edge (s, ϕ, a, R, s') takes no time. A move sets to zero the values of the timers in R. The values of all the timers increase uniformly with time and, at any instant, the value of a timer is equal to the time elapsed since the last time it was reset. The automaton can perform a move only if the timing constraint ϕ associated to the edge is satisfied by the current values of the timers. The automaton may stay at a given state s but it cannot let time pass beyond the bound imposed by the associated safety timing constraint $\mathtt{inv}(s)$. At any instant, the state of the system can be fully described by specifying the current state of the automaton and the value of all its timers. The formal operational semantics of a timed automaton is given in [8, 7].

4 Graph of timers

A timer system T is defined to be the tuple $\langle X, \mathtt{lab}, \mathtt{type}, \mathtt{upbd}, \mathtt{comm} \rangle$, where X is a finite set of timers, $\mathtt{lab} : X \to Act$ is a partial function that maps each timer to an action name, $\mathtt{type} : X \to \{\mathtt{ac}, \mathtt{to}, \mathtt{wt}\}$ determines the type each timer, $\mathtt{upbd} : X \to \mathbb{N}_\omega$ represents constant upper bounds on timer values (when $\mathtt{lab}(t) = \omega$, there is no finite upper bound)

and $\mathbf{comm} : 2^X \to Act$ is a partial function mapping set of timers to action names (i.e., the communication function).

For instance, the timer system of the sensor process of Example 2.1 is shown in Figure 2: t_4 is associated to the watchdog construct which determines the periodicity of reading, t_2 represents the timeout on the delay of sending data to the monitor, which is modeled by the t_1.

timer	lab	type	upbd
t_0	*sample*	ac	0
t_1	*data_s*	ac	ω
t_2		to	2
t_3	*data_s*	ac	0
t_4		wt	10

Figure 2: Timer system of the sensor process.

A graph of timers G is a tuple $\langle T, X^0, \succ, \triangleright \rangle$ where T is a timer system, $X^0 \subseteq X$ is the set of initial timers and $\succ \subseteq X^2$, $\triangleright \subseteq X^2$.

The graph of timers of a term is obtained by a sintax-driven construction. Figure 3 shows the graph of timers of the sensor process. Notice that, if the data is sent before a time equal to $\mathbf{upbd}(t_2)$, denoted by the expiration of t_1, then t_2 should be stopped but not t_4. When the timeout expires, because a time equal to $\mathbf{upbd}(t_2)$ elapsed, the action *data_s* becomes enable and must be performed immediately. The enabling of the action corresponds to reset t_3. Since $\mathbf{upbd}(t_3)$ is equal to 0, then it expires immediately without letting time pass. This situation is captured by the relation $\triangleright$, $t \triangleright t'$ means that t' is reset when t expires. In the example, $t_2 \triangleright t_3$.

	t_0	t_1	t_2	t_3	t_4
$\succ$			$\{t_1\}$		$\{t_2\}$
$\triangleright$	$\{t_1\}$		$\{t_3\}$		$\{t_0\}$

Figure 3: Graph of timers of the sensor process.

Consider, for instance, the graph of timers of the sensor process (Figure 3). The timer t_0 is the initial timer. When it expires, t_4 is reset and also t_2 and t_1.

Now, consider the scenario where t_4, t_2 and t_1 have been reset simultaneously. When t_1 expires, because its associated action is performed, t_2 must be stopped, but not t_4 since it is a watchdog timer. Hence, in the "following" scenario only t_4 remains. We define the relation $\not\succ \subseteq X^2$:

$$\frac{x \succ x'}{x \not\succ x'} \qquad \frac{x \not\succ x' \wedge x' \succ x''}{x \not\succ x''} \qquad \frac{x \not\succ x' \wedge x' \triangleright x'' \wedge x \neq x'' \wedge \mathbf{type}(x') \neq \mathbf{ac}}{x \not\succ x''}.$$

The timed automaton of a term is $\langle S, X, s_0, \mathtt{inv}, \rightarrow \rangle$ where $S = 2^X$, $s_0 = X^0$ and $\rightarrow$ is constructed starting at the set of initial timers, applying the following rules for any timer t in the current state: (1) if t is an action timer, then all the timeout timers t' such that $t' \gg t$ are removed, (2) if t is a timeout or watchdog timer, then all the timers t' such that $t \gg t'$ are removed, (3) for all t' removed, we add the timers t'' such that $t' \triangleright^* t''$.

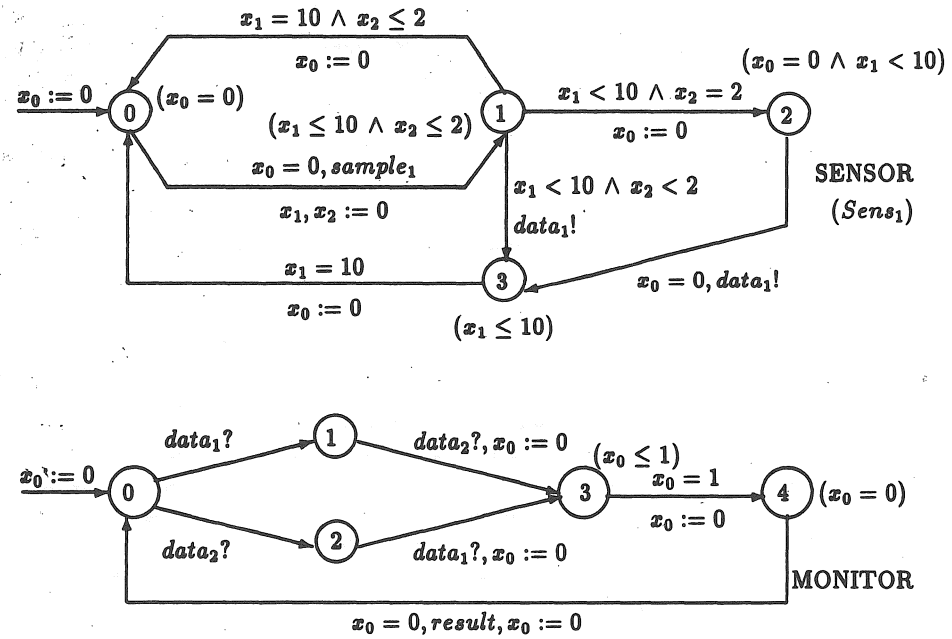

Figure 4: Timed automata for the monitoring system.

5 The tool

We discuss here some implementation details of the existing tool.

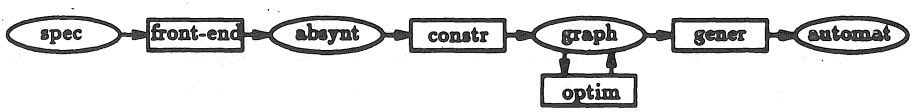

Figure 5: The functional architecture of the compiler.

The functional architecture of the compiler is shown in figure 5. The compilation is done in successive steps:

1. The front-end performs standard syntactic and semantic analysis, expands recursive definitions and constructs an attributed abstract syntax tree, where timers are associated to actions, timeouts and watchdogs.

2. The second phase constructs the *graph of timers*, following a syntax-driven definition.

3. The optimization phase reduces the number of timers by applying standard techniques for the allocation and assignment of registers during code generation (Fig 6).

4. The generation phase generates the corresponding timed automaton by performing some kind of reachability analysis. A state of the automaton is a set of timers. The generation phase mainly consists of three components: the first one computes for a given state of the automaton the outgoing transitions and successor states; the second component is the management of a fast-access extensible hash-table for storing and searching states; the third one consists in the breadth-first construction of the automaton, starting at the initial state.

class	timers
x_0	$\{t_0, t_1, t_3\}$
x_1	$\{t_4\}$
x_2	$\{t_2\}$

Figure 6: Optimization of timers.

Figure 4 shows the timed automata of the monitoring system processes generated by the tool. The compiler handles time parameters of the time constructs symbolically. That is, in the example, the timed automaton of sensor $Sens_2$ is the same as the one of $Sens_1$ shown in the figure, except that timer x_2 is compared to 4 instead of 2.

6 Conclusion

We have briefly presented ATP and timed automata, and illustrated their use with a very simple example. One interesting property of the compilation method is that the time parameters of the time constructs are handled symbolically. Consequently, the size of the timed automaton is completely independent of time constants. The current version of the tool is made up of approximately 11,000 lines of C code. It has not been yet used to compile large specifications. However, the performances achieved for the examples considered appear to be satisfactory.

References

[1] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real time systems. In *Proc. 5th Symp. on Logics in Computer Science*, pages 414–425, IEEE Computer Society Press, 1990.

[2] R. Alur and T. Henzinger. Logics and models of real-time: a survey. In *Proc. REX Workshop "Real-Time: Theory in Practice"*, Lecture Notes in Computer Science 600, Springer-Verlag, the Netherlands, June 1991.

[3] E.A. Emerson. Temporal and modal logic. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, pages 995–1072, Elsevier Science Publishers (North-Holland), 1990.

[4] C. Gehzzi, D. Mandrioli, S. Morasca, and M. Pezzè. A unified high-level Petri net formalism for time critical systems. *IEEE Transactions on Software Engineering*, March 1991.

[5] T.A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *Proc. 7th Symp. on Logics in Computer Science*, IEEE Computer Society Press, 1992.

[6] X. Nicollin and J. Sifakis. *The algebra of timed processes ATP: theory and application.* Technical Report RT-C26, LGI-IMAG, France, December 1990. To appear in *Information and Computation.*

[7] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, September 1992.

[8] X. Nicollin, J. Sifakis, and S. Yovine. From ATP to timed graphs and hybrid systems. In *Proc. REX Workshop "Real-Time: Theory in Practice"*, Lecture Notes in Computer Science 600, Springer-Verlag, the Netherlands, June 1991. To appear in *Acta Informatica.*

[9] G.M. Reed and A.W. Roscoe. A timed model for Communicating Sequential Processes. *Theoretical Computer Science*, 58:249–261, 1988.

[10] Wang Yi. Real-time behaviour of asynchronous agents. In J.C.M. Baeten and J.W. Klop, editors, *CONCUR 90: Theories of Concurrency*, Lecture Notes in Computer Science 458, Springer-Verlag, 1990.