

Validation of Distributed Periodic Real-Time Systems Using CAN Protocol with Finite Automata*

Gaëlle Largeteau

LISI, Université de Poitiers & E.N.S.M.A

Téléport 2, 1 av. C. Ader, BP 40109, F-86961 Futuroscope Chasseneuil Cédex
and

Dominique Geniet

LISI, Université de Poitiers & E.N.S.M.A

Téléport 2, 1 av. C. Ader, BP 40109, F-86961 Futuroscope Chasseneuil Cédex
and

Jean-Philippe Dubernard

LIFAR, Université de Rouen

Place Émile Blondel, F-76821 Mont-Saint-Aignan Cédex

Abstract

In a previous work, we define a temporal model based on regular languages to validate periodic real-time systems: the feasibility decisional process is expressed by means of algebraic operations on languages, such as intersection, Hadamard product, and language center computing. Here, we describe how this model can be used to validate periodic distributed real-time systems. We base this description on the example of the CAN network protocol.

1. INTRODUCTION

Real-time systems are reactive: they react continuously to the stimuli coming from their environment. In addition, hard real-time systems must react in a given delay, whatever the processor load and the synchronization (resource sharing, communication). To validate a hard real-time system consists in guaranteeing firstly its algorithmic correction (functional validation), and secondly that, whatever the context, each task is always able to react before its deadline (temporal validation).

Two different approaches usually address this problem: on-line or off-line policies. Using an

off-line policy consists in searching (by means of simulations) a scheduling sequence that satisfies the given temporal constraints. Once coded in a sequencer, this sequence will drive the application. Reversely, an on-line scheduler drives the application by choosing, at each swap, the task to schedule. This choice is based on heuristics whose role is on the one hand to provide to the scheduler a decisional capacity (the problem is NP-hard), and on the other hand to make it the faster possible (the processor cannot be used for a long time for scheduling management). The use of heuristics makes the temporal validity of such a scheduler unprovable. Thus, its validation is obtained by the way of an off-line analysis (by simulation).

These two approaches have generated a great number of works, in the real-time community. Most of them are based on the task temporal model proposed by Liu and Leyland in [1]. First of all, the expressing power of the different proposed methods has been studied. This leads to the notion of **optimality** of an on-line scheduler: it gives a correct sequence, if one exists. When the tasks are interdependent (i.e. resource sharing or communication are concerned), it is proved that an optimal on-line algorithm cannot be found [2][3][4]: simulation cannot be avoided. Since the system must be validated for any processing duration, one must give an upper bound

*À paraître dans les actes de **Systemics, Cybernetics and Informatics**'2001, 22-25 Juillet 2001, Orlando, Floride (USA)

for the duration of simulation (because simulation must obviously operate on a finite time interval). [3] gets a value for independent task systems, which is improved and generalized for distributed systems of interdependent tasks in [5].

Most of the proposed approaches solve a specific problem: task or message scheduling, load balancing, etc. In [6], we define a new methodology, based on regular languages, to validate periodic real-time systems: each periodic task is associated with a language which collects its *correct* behaviours, and we use compositional techniques based on Arnold and Nivat synchronized products [7] to decide the feasibility of the task system. In [8], we extend this technique to real-time systems composed of both periodic and sporadic tasks.

In our technique, time is implicit. This property leads the model to be easier than others¹ to analyse. Indeed on the one hand, the decisional process is directly based on the finite automata analysis toolbox, which algorithms are known to be very efficient. For instance, the central theorem of [5]² is generalized to multi-CPU target architectures as a corollary of the existence of the automaton associated with a real-time system. On the other hand, our approach allows to express fine properties (like preemption variations during the life of a task, for instance). In timed approaches, these kinds of properties may be more complex to model, because they impose both complex equational techniques or additional structural definitions.

Here, we describe how our approach can be used to validate distributed real-time systems. Our description is based on an industrial network protocol, CAN, that is currently used by Peugeot (for instance), in new cars.

In part , we present the class of real-time applications that we study. In part , we describe the temporal model used to validate real-time systems. In section , we show how CAN protocol can be modeled, and we give some synchronization techniques to validate a distributed real-time system where tasks communicate through a CAN network.

2. DISTRIBUTED REAL-TIME SYSTEMS

A control-command system is a **real-time** system when the physical driven process has to interact strongly with its environment. To avoid crash situations, the computer must run with a speed compatible with the one of the physical process. Then, such a system is **reactive**. It is composed of elementary processes, named **tasks**. These tasks may have different functions (see Figure 1.Left):

- Scanning tasks have to get the physical datas, coming from the captors of the physical process. These captors usually send their datas regularly, with a physical defined frequency. Of course, scanning tasks have to get the datas with the same frequency. They are called **periodic** tasks. The periodic task τ_i is characterized with four timing attributes: its **period** T_i , its **deadline** D_i , its **first activation date** r_i , and its **CPU time** C_i .
- Decision tasks receive datas from all the scanning tasks (and, sometimes, from other decisional tasks). They have to produce the flow of driving decisions to transmit to the process motor commands. Of course, their speed must be in accordance with the one of scanning tasks.
- Command tasks have to transmit command messages to the motor driving devices.

These periodic tasks are completed with sporadic tasks, which are only activated when alarm signals are received from the captors. These tasks are characterized by two time attributes: C_i and D_i . In the following, a real-time system is supposed to be composed of two task systems: $(\tau_i)_{i \in [1, n]}$ is the periodic component of the system, and $(\alpha_i)_{i \in [1, p]}$ its non-periodic component. When the physical process is complex (see, for instance, on Figure 1.Right), the real-time command system is often distributed: many computers are used, which are connected to a network bus. For real-time systems, specific network protocols are used, to guarantee the respect of deadlines for delivrance of critical messages. So, a distributed real-time system can be viewed as presented on Figure 2.Right.

¹timed models [9], or Petri net based models [5]

²It proves the cyclicity of each infinite scheduler sequence for a real-time system, on a 1-CPU system.

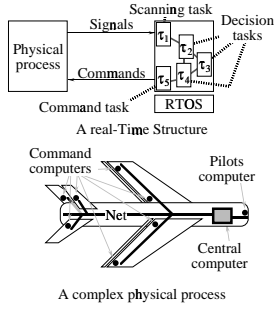


Figure 1: An example of real-time system for a complex physical process

3. MODELING REAL-TIME TASKS WITH FINITE AUTOMATA

Validate programs consists in proving that all their possible behaviours are consistent with the semantics associated with the program. For real-time systems, this validation process must be completed with a **temporal validation**, which consists in proving that whatever the evolution of the physical system, no task may miss its deadline. For instance, in a plane (see Figure 1.Right), a fire alarm signal cannot be missed by its scanning task because of the computer system scheduling policy.

In this paper, we are interested in the temporal validation of distributed real-time systems. Our technique [6][8] is based on a real-time task temporal model: a task is associated with a regular language that collects all its **valid temporal behaviours**: for $t \in \mathbb{N}$, a t -valid temporal behaviour of a task is an history of its processor allocation time units on $[0, t[$ such that we can guarantee that in the future (i.e. $[t, +\infty[$) of the task, there is at least one possible processor allocation sequence compatible with the temporal constraints of the task. On an algebraic plan, these valid temporal behaviours are words $\omega \in \{a, \bullet\}^*$, where a symbolizes that τ_i owns a processor for one time unit, and \bullet that τ_i is suspended for one time unit. Hence, the word $aaa \bullet \bullet aa$ models a processor allocation sequence where τ_i runs for 3 time units, and next is suspended for 2 time units, and then runs for 2 time units.

In [6], we show that the set $L(\tau_i)$ of the valid

temporal behaviours of a given periodic task³ is $Center(\bullet^{r_i}((a^{C_i} \text{III} \bullet^{D_i - C_i}) \bullet^{T_i - D_i})^*)$. When there is resource sharing or communication between tasks, we use the Hadamard product of languages for model concurrency: let ω and ζ be two temporal valid behaviours of tasks τ_1 and τ_2 , the behaviour (ω, ζ) of the system (τ_1, τ_2) is the word

$$\begin{pmatrix} \omega_1 \\ \zeta_1 \end{pmatrix} \begin{pmatrix} \omega_2 \\ \zeta_2 \end{pmatrix} \dots \begin{pmatrix} \omega_{|\omega|} \\ \zeta_{|\zeta|} \end{pmatrix}$$

The vector $\begin{pmatrix} \omega_i \\ \zeta_i \end{pmatrix}$ models that τ_1 is in state ω_i while τ_2 is in state ζ_i . We denote this operation $\omega \Omega \zeta$. This product is naturally extended to languages in the way $L \Omega M = \{\omega \Omega \zeta, (\omega, \zeta) \in L \times M\}$.

To synchronize tasks, we use Arnold-Nivat's techniques [7]: a is refined in $\{a, \text{critical statements}\}$, and the synchronization of tasks is implemented through algebraic operations on the sets of valid temporal behaviours. The temporal behaviours [6] of a task system is the center of the synchronized product of the $L(\tau_i)$'s. This language is $L((\tau_i)_{i \in [1, n]}) = Center\left(S^* \cap \bigcap_{i=1}^{i=n} L(\tau_i)\right)$,

where S collects all the $\begin{pmatrix} \omega_1 \\ \vdots \\ \omega_n \end{pmatrix}$ corresponding to valid instantaneous configurations.

4. THE CAN NETWORKING PROTOCOL

The CAN⁴ protocol was developed from 1983. It is a MAC layer based on CSMA/AMP technique. When a node transmits datas, it follows the protocol designed on Figure 2.Left:

1. It waits for medium freeness.
2. It starts an arbitration game. Every node concerned in a transmission will play.
3. The arbitration game winner transmits its message, all other nodes read it (transmission follows a broadcasting protocol). If the

³here, we deal with tasks with fixed execution times.

⁴Controller Area Network

arbitration initiator node is not the winner, it will try again to transmit in the future, as soon as it can.

- Transmission is terminated with a *EOT* signal.

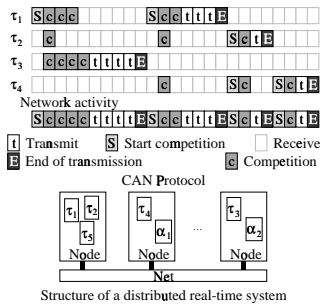


Figure 2: CAN protocol

Used to transmit a set of periodic messages whose transmission dates can be statically computed, this protocol is deterministic. In periodic real-time systems, the competition identifiers of the nodes are messages identifiers. As far as we know statically the relation between senders and messages, the transmission dates can statically be computed. Thus, we can use our technique for this protocol.

5. MODELING COMMUNICATION WITH FINITE AUTOMATA

Viewing communication as tasks

Managing interdependence, like communication, leads to guarantee the corresponding properties for the tasks: mutual exclusion, precedence etc.

In Arnold-Nivat's model [7], this control is implemented through virtual tasks, whose role is to model physical resources or messages. Arnold and Nivat consider the communication as a slave task: it is associated with a regular language of behaviors, L_r , i.e. to an automaton A_r . The language $L = \left(\bigcap_{i=1}^{i=n} (L(\tau_i)) \right) \Omega \left(\bigcap_{k=1}^{k=m} (L_{r_k}) \right)$ is a set of words built on an alphabet which letters pattern is $(a_1, \dots, a_n, s_1, \dots, s_m)$, where the a_i 's are the τ_i 's states, and the s_i 's the slave tasks statements. A word $\omega = (\omega_1)(\omega_2) \dots (\omega_{|\omega|}) \in L$ is a valid behaviour of the task system (in the

sense where it respects the correct use of communication protocols) if, foreach ω_i , the execution of a R_k critical statement (P or V) by one of the τ_j 's is always simultaneously performed by the virtual task associated with R_k . This constraint leads ω to follow the predicate: $\forall i \in [1, |\omega|], \forall k \in [1, m], \forall x$ critical statement on $R_k, \omega_{i_k} = x \iff \exists ! j \in [1, n]$ such that $\omega_{i_j} = \omega_{i_k}$. Let us call S the set of valid vectors (i.e. that follows the previous predicate). Each ω in $L \cap S^*$ is a valid behaviour: it respects both the temporal constraints and the protocol used for sharing critical resources. Moreover, it can be infinitely extended by respect to these constraints.

In the net communication context, we extend this approach, according the net to a slave task. This task must have some particular characteristics :

- It is not preemptive : once a message transmission starts on the medium, no other transmission can interrupt it.
- The medium work is independent from the sites. In our model, we consider it as a processor.

Thus, net communication is a non preemptive slave task. Then, it is modeled through a finite automaton which structure directly depends on the followed protocol. For Can, the Net communication automaton is described in section .

Integrating CAN protocol into finite automata model

In the CAN protocol, a task can be in four states : waiting, prepare the competition, competition and transmission. So, it is naturally associated with a four states automaton (see Figure 3). The First state is the waiting state, waiting for the medium freeness. An incoming message is represented by the SOF transition, the second state prepares the arbitration game. Then the competition between all the messages begins: the system in the state 3. Once the arbitration game finishes, the automaton goes into the state 4 and the winner starts transmitting its message. At the end of the transmission, the automaton goes back to the first state by the EOT transition.

A Rendez-Vous communication between real-time tasks

The transmitter performs : Send, waiting instructions, transmission , EOT. The receiver performs:

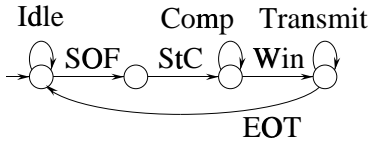


Figure 3: CAN protocol model automaton

Receive, receiving the message, EOR. The synchronization mechanism consists in building the product of the three automata (transmitter, receiver, protocol) and see whether transitions are valid or not. A transition of the product automaton is labeled with (emitter action, receiver action, protocol action). The only valid transitions

are the 3-tuples $\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix}$ that satisfy:

$$\begin{aligned} \omega_1 = Send &\Rightarrow \omega_3 = SOF \vee \\ \omega_2 = Receive &\Rightarrow \omega_3 = Win \vee \\ \omega_2 = EOR &\Rightarrow \omega_3 = EOT \end{aligned}$$

Let S be the set of such configurations. Computing the set of valid⁵ behaviours is implemented by applying the technique of section for the pair $(\tau_i, Protocol)$.

Principles for integrating asynchronous communication in the model

When the communication is asynchronous, sending a message consists in saving it on a spool. The network board driver starts transmissions until this spool is empty: each transmission deletes a message. To model such a communication, we must express each network board driver (it is a node of the network) as a task. A message is transmitted from a task to an other, via a buffer, a driver, the network, an other driver, an other buffer, and finally the destination task. The dynamics is:

1. The transmitter puts its message in the buffer, and then the driver reads it.
2. The driver starts competition with other drivers and when it wins, it transmits the message and deletes it from its buffer.
3. When the reception ends, the receiver driver stores the message in its buffer.

⁵in the sense of correct net use

4. Finally, the receiver task reads the message from the buffer.

To express this process in the model, we need first to model the driver as an automaton (a driver is a task). This automaton (see Figure 4.Left) models the details of the communication between the drivers (competition, transmission, reception). There are two different synchroniza-

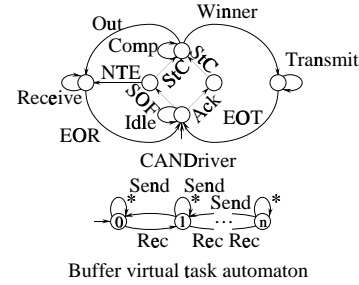


Figure 4: CAN drivers model automaton

tions:

- First, we synchronize tasks and their driver by the use of a classical resource sharing automaton (see Figure 4.Right): the states of this automaton store the number of pending messages.
- Second, the drivers have to be synchronized by following the net protocol : the automaton presented in Figure 3 is now viewed as a *synchronizer* task.

To model the synchronization between a task, its target buffer, and the network driver, we collect the set S of valid instantaneous configurations $\omega = (\omega_1, \omega_2, \omega_3)$, where ω_1 models the activity state of the task, ω_2 the state of the buffer, and ω_3 the activity state of the driver. The configuration ω belongs to the set of valid configurations if

$$\begin{aligned} \omega_1 = Send &\Rightarrow \omega_2 = Smsg \vee \\ \omega_3 = SOF &\Rightarrow \omega_2 = Msg \vee \\ \omega_3 = StC &\Rightarrow \omega_2 = Msg \vee \\ \omega_3 = NTE &\Rightarrow \omega_2 = NoMsg \vee \\ \omega_3 = EOR &\Rightarrow \omega_2 = Rmsg \vee \\ \omega_3 = EOR &\Rightarrow \omega_2 = Smsg \vee \\ \omega_1 = Receive &\Rightarrow \omega_2 = Rmsg \end{aligned}$$

On the same way, the drivers have to be synchronized for the access of network medium. The virtual task used for this synchronization is the CAN

protocol. So, an instantaneous configuration of the system (*Drivers, Protocol*) is a $(n + 1)$ -tuple ω : for $i \in [1, n]$, ω_i models the i^{th} driver activity state, and ω_{n+1} models the protocol state. the configuration ω is a valid instantaneous one if

$$\begin{aligned} \omega_{n+1} = Idle &\Rightarrow (\forall i \in [1, n], \omega_i = Idle) \vee \\ \omega_{n+1} = SOF &\Rightarrow (\exists i \in [1, n], (\omega_i = SOF) \\ &\quad \wedge (\forall j \in [1, n] \setminus \{i\}, \omega_j = Ack)) \vee \\ \omega_{n+1} = StC &\Rightarrow (\forall i \in [1, n], \omega_i \in \{NTE, StC\}) \vee \\ \omega_{n+1} = Comp &\Rightarrow \\ &\quad \left\{ \begin{array}{l} (\exists (i, j) \in [1, n] \times ([1, n] \setminus i), \omega_i = \omega_j = Comp) \\ \neg(\exists i \in [1, n], \\ \omega_i \in \{Idle, StC, EOR, Emit, Win, Ack, SOF\}) \end{array} \right\} \\ \vee \omega_{n+1} = Win &\Rightarrow (\exists i \in [1, n], ((\omega_i = Winner) \\ &\quad \wedge (\forall j \in [1, n] \setminus i, \omega_j \in \{Out, Receive\}))) \vee \\ \omega_{n+1} = Transmit &\Rightarrow (\exists i \in [1, n], (\omega_i = Transmit) \\ &\quad \wedge (\forall j \in [1, n] \setminus i, \omega_j = Receive)) \vee \\ \omega_{n+1} = EOT &\Rightarrow (\exists i \in [1, n], (\omega_i = EOT) \wedge \\ &\quad (\forall j \in [1, n] \setminus i, \omega_j = EOR)) \end{aligned}$$

6. CONCLUSION

So, the CAN protocol can be integrated in our temporal model as a specific virtual task: the model is a finite automaton. We have shown that both synchronous and asynchronous communication can be implemented. In synchronous mode, tasks are directly synchronized by the use of the protocol automaton; in asynchronous mode, buffer states and network drivers have to be expressed, and the synchronization of tasks needs three atomic synchronizations: transmitter and its driver, the drivers, and the target driver and the receiver task. Here, CAN is an example, this methodology can be used with any network protocol that can be described with a finite automaton.

Then, we plan to extend this work in two directions. First, we will show that all the classical industrial network protocols can be integrated in this approach. Second, we will extend our methodology to aperiodic network traffic (alarm messages). Future works will explore them.

References

[1] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard

real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

- [2] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, pages 301–324, 1990.
- [3] J.Y.T. Leung and M.L. Merill. A note on preemptive scheduling of periodic real-time tasks. *Information Processing Letters*, 11(3):115–118, 1980.
- [4] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, pages 237–250, 1982.
- [5] E. Grolleau. *Ordonnancement Temps-Réel Hors-Ligne Optimal à l'Aide de Réseaux de Petri en Environnement Monoprocasseur et Multiprocasseur*. PhD thesis, Univ. Poitiers, 1999.
- [6] D. Geniet. Validation d'applications temps-réel à contraintes strictes à l'aide de langages rationnels. In *RTS'2000*, pages 91–106, 2000.
- [7] A. Arnold and M. Nivat. Comportements de processus. Technical Report 82-12, Univ. Paris 7, 1982.
- [8] D. Geniet and J.P. Dubernard. Ordonnancement de tâches sporadiques à contraintes strictes à l'aide de séries génératrices. In *Proc. of RTS'01*, 2001.
- [9] L. Aceto, P. Bouyer, A. Burgue no, and K. G. Larsen. The power of reachability testing for timed automata. In *Proc. of 18th Conf. Found. of Software Technology and Theor. Comp. Sci.*, LNCS 1530, pages 245–256. Springer-Verlag, December 1998.