24

Adaptive Implementation of Discrete Event Control Systems based on Sequential Function Charts

Ramón Piedrafita and José Luis Villarroel
Department of Computer Science and Systems Engineering,
University of Zaragoza
Spain

1. Introduction

The discrete-event system (DES) is a class of dynamic systems whose behaviour is governed by *discrete events* and they state occupy a discrete symbolic-valued state at each time instant. These discrete events occur asynchronously and instantaneously at discrete instants of time and lead to a change of the state. Between event occurrences, the state of DES is unaffected. The DES behaviour is described by the sequence of events that occur and the sequence of states. Examples of DES abound in the industrial world as automated manufacturing systems, monitoring and control systems, supervisory systems; in building automation; in control of aircraft systems, railway systems…(Cassandras 1993).
An example of a discrete event system is the classic programmable logic controller (PLC) controlling a sequential machine. The PLC acts as a discrete event control system (DECS). The DECS acts through the outputs over the actuators of the machines, and receives information of the state of the machines or events that happen in them through sensors. In the design of a DECS is neccesary to specify its dynamic behaviour, that is, the form of generating its outputs in response to the inputs. This specification can be carried out in different forms and will be a model of the desired behaviour of the system. There may be various desired behaviours for the same machine if the actions to be performed are different. The specification for the desired behaviour can be performed using the formalism of Petri nets. The technology translation can be done in a PLC in Sequential Function Chart language (SFC).
Programmable Logic Controllers are extensively used in the control of production systems and their use is, at the present, widespread in most industrial sectors. The combination of the PLCs intelligence with the development of sensors and actuators, ever more specialized, allows a greater number of processes to be automated. These devices offer a series of advantages that meet some of the most important manufacturing industry requirements in recent years, such as low cost, capacity to control complex systems, flexibility (they can be quickly and easily re-programmed), reduced downtime and easier programming, and reliable and robust components ensuring their operation for a long time.

The reaction time of a PLC is a fundamental matter in discrete event control systems. The PLC reads the inputs, executes the SFC and writes the output in a cyclic or periodic manner. In this chapter, we are interested in the execution time of algorithms that make the SFC of a control application evolve. We will show that the reaction time of a PLC depends greatly on the SFC structure, on the events sequence and also on the algorithm that executes the SFC. With the objective of minimizing the reaction time, we decided to design a Supervisor controller, which we have called the Execution Time Controller (ETC). The aim of the ETC is to determine in real time which algorithm executes the SFC the fastest and to change the execution algorithm when necessary.
We propose to adapt the classical implementation techniques of Petri nets to execute SFCs. Thus, we have developed execution algorithms derived, on the one hand, from the Deferred Transit and the Immediate Transit SFC evolution models and, on the other hand, from Petri net implementation techniques (Brute Force, Enabled Transitions and Representing Places).
The organization of this chapter is as follows. Section 2 is devoted to Discrete Event Systems, and Section 3 to Sequential Function Charts. Section 4 shows several implementation techniques of the SFC whose execution time is analyzed in Section 5. In Section 6 we present the Execution Time Controller. In Section 7 the technique is evaluated. The section describes the tests run to evaluate the estimation techniques and the working of the ETC in real time. Finally, in Section 8, we present the main conclusions.

2. Discrete Event Control Systems

An example of a discrete events system is the classic PLC controlling a sequential machine. The PLC acts as a discrete event control system (DECS) (see Fig. 1). The DECS acts on the machines by sending output signals to the actuators and receives information about the state of the machines or events occurring in them through sensors. The DECS receives input signals not only from the machine sensors, but also from the commands of the control panel, from supervision systems and even from other DECS. An output signal can be a signal sent to an actuator to act on a physical process, to increase a variable or to send a message.
The main function of discrete event control systems is to govern the workings of a machine in such a way that the desired behaviour is achieved. This is based on the coordination between the information received and the actions ordered to be carried out. A machine carries out the action ordered by the control system until the system decides that the action has been completed at which point it orders the machine to cease the action. In order that the control system can decide to end the action, it needs to obtain information indicating that the action should finish. This information can come from the sensors placed in the machine. With this information, the control system knows that it must execute an evolution. It has to pass from the state in which it performs the action to the subsequent state which could be one of many (perform another action, await material, etc.).
An approach to the design of a DECS involves specifying its dynamic behaviour, in other words the way it generates its outputs in response to the inputs. This specification can be carried out in various ways and will be a model of the desired functioning of the system. The same machine may have different ways of functioning if the actions to be performed are different. The specification of the desired behaviour can be carried out using formalisms such as Petri nets. The technology translation can be done in a PLC using the Sequential Function Chart language (see Fig. 2).

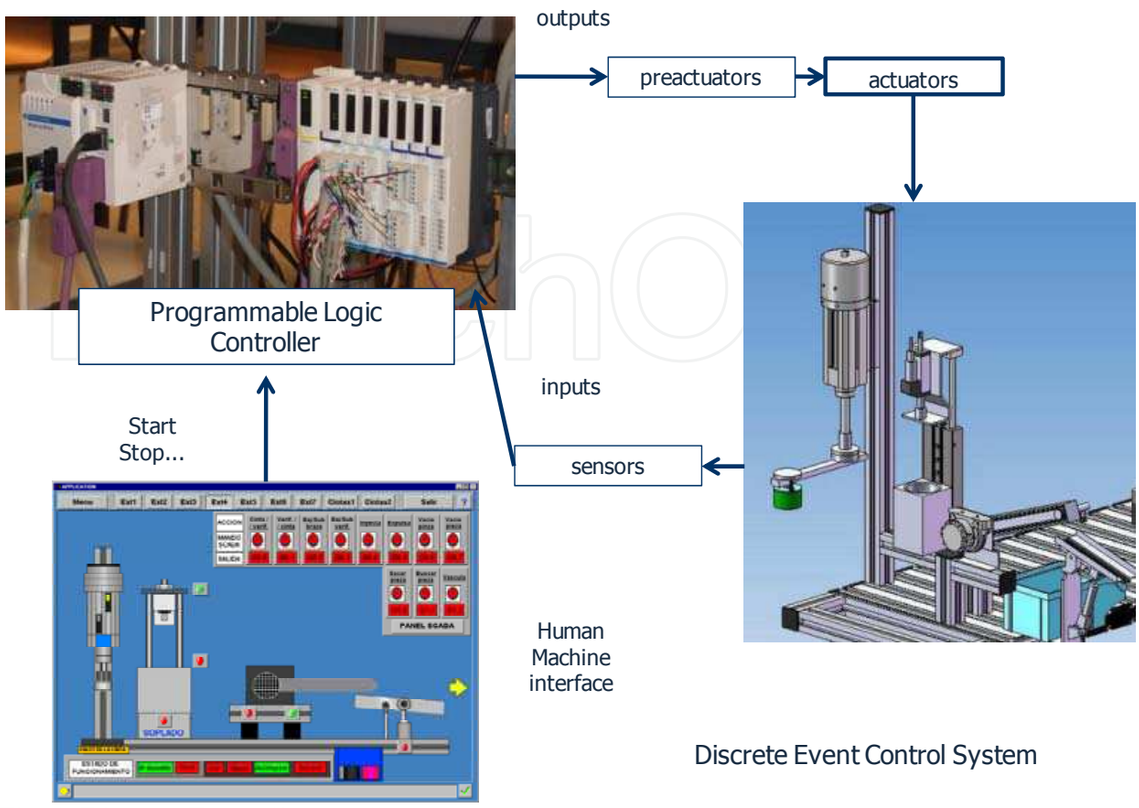

Fig. 1. Discrete Event Control System

3. Sequential Function Charts

In 1975, one of the working groups of the now defunct AFCET (Association Francaise pour la Cibernétique Economique et Technique), the Logic Systems group, decided to establish a commission for the standardization of the representation of logic controller specifications. In August 1977 a commission comprising 12 academics and researchers and 12 representatives of companies such as EDF, CEA, Merlin-Gerín, and Telemecanique signed the final report.
In brief, the group was looking for a model for the representation and specification of the functioning of systems controlled by logic controllers, through automatisms. The specification model only describes the desired behaviour, without detailing the technology with which the real implementation is effected. The model was named Grafcet (David 1995) and is recognised by standard IEC-848 (IEC 1988).
Similar to Grafcet, the Sequential Function Chart (SFC) are standardized in IEC 61131 (ISO/IEC 2001) where is defined as one of the main PLC programming languages. A SFC program is organized into a set of steps and transitions connected by direct links. Associated with each step is a set of actions, and with each transition a transition predicate.

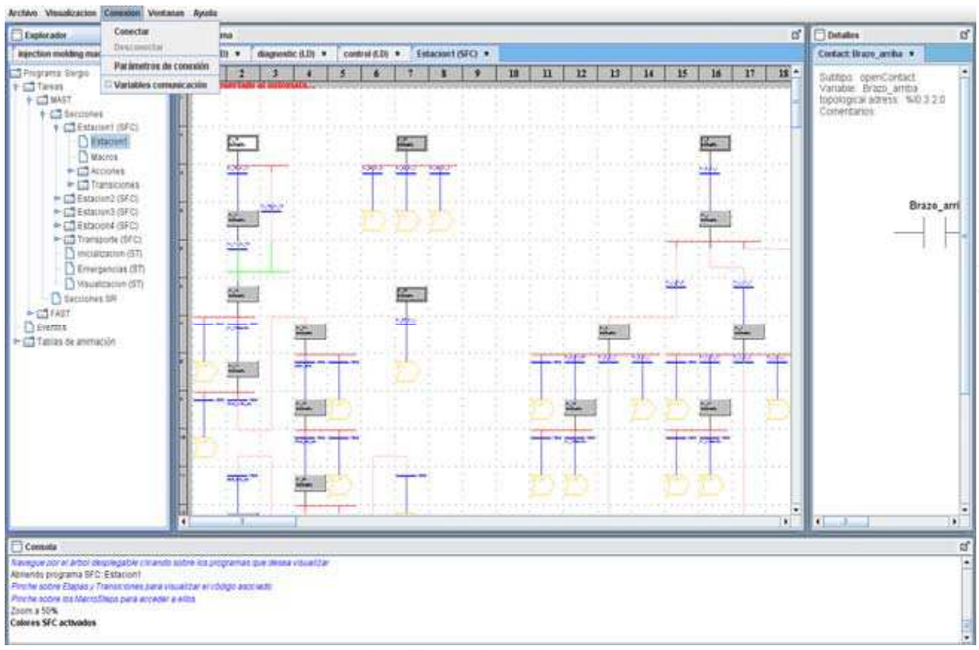

Fig. 2. PLC programming in Sequential Function Chart

The SFCs are binary Petri nets with an interpretation for the control of industrial systems (Silva 1985)

- Immediate actions are associated with the deactivation and activation of the steps (e.g., control signal changes, code execution).
- Level control signals are associated with active steps.
- Predicates are associated with transitions, as are additional preconditions for the firing of enabled transitions. Predicates are functions of system inputs or internal variables.

We take as an example the SFC shown in Fig. 3. The initial step (*Automatic_star*) is drawn with a double rectangle. The two output transitions of the initial step (*move_piece* and *NOT move_piece*) are in conflict. The default priority rule for solving a conflict is a left to right precedence. The standard does not require a priority relation between transitions or that the transitions predicates are in mutual exclusion.

When all the input steps of a transition are active and the transition predicate or condition is true, the transition is fired, the input steps are deactivated and the output steps are activated. In the example of the Fig. 3: if the step named *handgotoup* is active and the transition *hand_up* is true, the step *handgotoup* is deactivated and the step named *handgotopiece* is activated.

Actions can be programmed in a step. The type of programmed action is defined by the action qualifier. For example, a type N action is executed in all the cycles in which the step is active. The S, SD, SL, and SD actions are activated when the step in which they are programmed is activated, stored in an action buffer and from this point on are independent of the state of the step. They can only be deactivated by a type R action. Time limited actions can be programmed with type L or D qualifiers. There are also impulse type actions such as type P that are executed only when the step is activated.

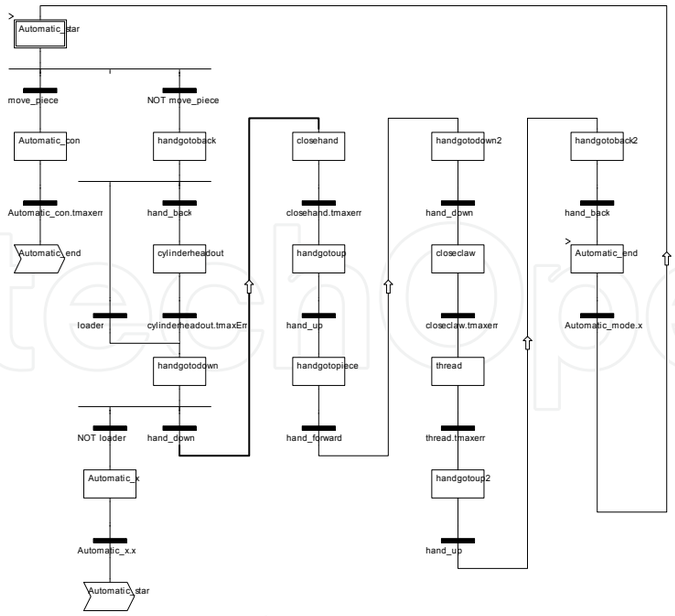

Fig. 3. Sequential Function Chart example

Table 1 shows the actions that can be programmed in a SCF. In a PLC cycle, the following must be executed:

- Actions which depend on the state of a step: action qualifiers N, L, D, P, P0, P1.
- The step in which is programmed the storage of the stored actions (S, SL, SD, DS) and their cancellation (R).
- the stored actions (S, SL, SD, DS)

The action types and qualifiers are the standard ones of the IEC 61131 (ISO/IEC 2001).

Qualifier	Description
N	Non-stored, executes while step is active.
L	Limited, executes only a limited time while step is active.
D	Delayed, starts executing after the step has been active.
S	Stored, starts executing when the step is activated until reset.
R	Reset stored action.
SL	Stored and limited
SD	Stored and delayed
DS	Delayed and stored
P	Pulse, executes when the step is activated.
P1	Pulse, positive flank, executes once when the step is activated.
P0	Pulse, negative flank, executes once when the step is deactivated.

Table 1. SFC actions.

4. Implementation of Sequential Function Charts

In the last 25 years, researchers have devoted considerable attention to the software implementation of Petri Nets (PN); see for example (Colom, Silva et al. 1986) (Briz and Colom 1994) (Taubner 1988) (Bruno & Marchetto 1986) (Garcia & Villarroel 1999) (Piedrafita & Villarroel 2006a). A PN implementation can be hardware or software. However, we are interested in the second approach, the software implementation. A software implementation is a program which fires the PN transitions, observing marking evolution rules, i.e., it plays the "token game". An implementation is composed of a control part and an operational part. The control part corresponds to the structure, marking and evolution rules of the PN. On the other hand, the operational part is the set of actions and/or codes of the application, associated with the PN elements.

According to different criteria, a PN implementation can be mainly classified as compiled or interpreted, as sequential or concurrent and as centralized or decentralized.

An implementation is interpreted if the SFC PN and the marking are codified as data structures. These data structures are used by one or more tasks called interpreters to make the PN evolve. The interpreters do not depend on the implemented PN. A compiled implementation is based on the generation of one or more tasks whose control flow corresponds to PN evolutions.

A sequential implementation is composed of only one task, even in PN with concurrency. This kind of implementation is common in applications whose operational part is composed by impulse actions without significant execution time. A concurrent implementation is composed of a set of tasks whose number is equal to or greater than the actual concurrency of the PN. Examples of concurrent implementations can be seen in (Colom, Silva et al. 1986) or in (Taubner 1988).

In a centralized implementation the full control part is executed by just one task, commonly called the token player or coordinator. The operational part of the implementation can be distributed in a set of tasks to guarantee the concurrence expressed by the PN (see for example (Colom, Silva et al. 1986)).

The problem of implementing a SFC is very similar to implementing a PN. Currently most industrial PLCs run their programs in an interpreted and centralized manner. The PLC reads the inputs, runs the SFC interpreter (also called coordinator in this paper) and writes the outputs. In the execution of the SFC it is necessary to determine which transitions can fire, and fire them making the state of the SFC evolve. It will also make the actions programmed in the steps.

The algorithm to determine which transitions are enabled and can fire is important because it introduces some overhead in the controller execution and the reaction time is affected. In the present work we have implemented and study several algorithms in which different enabled transition search techniques are developed:

- Brute Force (BF). PN implementation technique.
- Deferred transit evolution model (DTEVM). SFC implementation technique.
- Immediate transit evolution model (ITEVM). SFC implementation technique.
- Static Representing Places (SRP). PN implementation technique.
- Enabled Transitions (ET). PN implementation technique.

With the objective of carrying out a comparative study, all of these techniques have been uniformly implemented.

In the Brute force algorithm all the transitions are tested for firing. Brute Force algorithms do not try to improve the search of enabled transitions. Works such as (Peng & Zhou 2004) (Uzam & Jones 1996) (Klein, Frey et al. 2003) belong to this implementation class.
The IEC-61131 standard is not very precise in the definition of the SFC execution rules. Different execution models have been proposed to interpret the standard. As with BF, in the Immediate Transit Evolution Model (ITEVM) algorithm all the SFC transitions are tested for firing (Hellgren, Fabian et al. 2005). However, the Deferred Transit Evolution Model (DTEVM) (Hellgren, Fabian et al. 2005) only performs the testing of the transitions descending from the active steps, improving in this way the Brute Force operation.
In (Lewis 1998) the IEC-61131 standard is interpreted and the following tasks are proposed to run an SFC:

1. Determine the set of active steps
2. Evaluate all transitions associated with the active steps
3. Execute actions with falling edge action flag one last time
4. Execute active actions
5. Deactivate active steps that precede transition conditions that are true and activate the corresponding succeeding steps
6. Update the activity conditions of the actions
7. Return to step 1

These tasks are implemented in the DTEVM algorithm. In DTEVM, the transition conditions of all transitions leading from active steps (marked places in Petri net terminology) are evaluated first. Then, the transitions that were found to be fireable are executed one by one. In ITEVM, the transition conditions of all transitions of SFC are evaluated one by one. In the case of a transition condition being true, i.e., the corresponding transition is fireable, this transition is fired immediately.
In the Static Representing Places (SRP) algorithm, only the output transitions of some representative marked steps are tested (Colom, Silva et al. 1986). Each transition is represented by one of its input steps, the Representing Place. The remaining input steps are called synchronization steps. Only transitions whose Representing step is marked are considered as candidates for firing.
In the Enabled Transitions algorithm, only totally enabled transitions are tested. A characterization of the enabling of transitions, other than marking, is supplied, and only fully enabled transitions are considered. This kind of technique is studied in works such as (Silva & Velilla 1982) and (Briz. 1995).

4.1 Algorithm execution cycle

All implementation techniques are based on a treatment cycle which processes steps or transitions commonly stored in lists. The implementation of treatment the cycle is based on two kinds of lists that make an SFC evolve: treatment lists to be processed in the present treatment cycle and formation lists to be processed in the next cycle. The fundamental difference between each of the implementation techniques lies in the way in which the formation lists are built, and hence in the transitions which are considered in each treatment cycle.
One of the most expensive operations in execution time is the search and insertion in lists. The time cost of such operations depends directly on the size of the lists. Therefore, it is stated in the algorithms where it carries out such operations.

The basic treatment cycle of a SFC interpreter consists of three phases: (1) Enabling Test, (2) Transition firings (with two sub-phases: start and end), and (3) Lists update.
The Enabling Test phase verifies the enabling of the transitions belonging to the treatment list. A transition is enabled if all of the input steps are active. An enabled transition will be fired in the next phase if the associated condition is true.
All algorithms present two separate phases in the firing of transitions:

1- Start of transitions firing: deactivation of input steps of each fired transition.
2- End of transitions firing: activation of output steps of fired transitions.

The TransitionsFired list links both phases. In this way, the SFCs are executed step by step and avalanche effects are avoided. At the end of firing, the formation list is built with places or transitions being candidates for treatment in the next cycle.
Finally, at the end of the cycle, the elements of the formation list are analyzed and can become part of the treatment list for the next cycle.
In the following paragraphs we show the ET (Silva & Velilla 1982) (Briz. 1995), SRP (Colom, Silva et al. 1986) and the DTEVM (Hellgren, Fabian et al. 2005) algorithms in more detail to illustrate how all the techniques have been coded. The ITEVM algorithm can be consulted in (Hellgren, Fabian et al. 2005). The procedures for the execution of the actions programmed in the SFC have been included, with the update of the activity conditions of the actions (ISO/IEC 2001).

4.2 Enabled Transitions Technique

Program 1 presents the basic treatment cycle of the coordinator for the ET technique. This treatment cycle is also illustrated in Fig. 4. The following data structures will be available (see Fig. 4):

- Enabled Transitions List (ETL). Treatment list made up of the transitions with all active input steps.
- Almost Enabled Transitions List (AETL). Formation list which is built with the output transitions of the steps activated in the firing of the transitions, i.e., the transitions that can become enabled in the next cycle.

```
loop forever
  Executeactionswithfallingedge();
  Executeactiveactions();
// enabling test
  while elements in ETL do
    T = next_element (ETL);
    if enabled (T) and predicate(T) then
// start of transitions firing
      Demark_input_steps(T, ETL);
             // ETL updating
      Add(Transitionsfired, T);
    end if ;
  end while ;
// end of transitions firing
  while elements in Transitionsfired do
    T = next_element (Transitionsfired);
```

```
    Mark_output_steps(T, AETL);
      // update AETL
  end while ;
  Clear(Transitionsfired);
//list update
  Update(ETL, AETL); //operations of search and insertions in list
  Clear(AETL);
  Updateactivityconditions();
end loop ;
```

Program 1. ET Coordinator Treatment Loop

For each transition of the SFC a data structure is necessary that stores:

- List of input steps
- List of output steps

At the start of transitions firing Demark_input_steps (T, ETL) encapsulates the deactivation of the input steps of the transition fired, and the update of the ETL list. In this technique, the ETL (the treatment list) contains all transitions enabled at the beginning of the cycle. From this list each fired transition must be extracted and also the disabled transitions belonging to effective conflicts.

In the function Update(ETL, AETL) the treatment list is prepared for the next cycle. The transitions in AETL are verified for enabling and, if positively verified, are added to the ETL (if they do not already belong). At this point, the algorithm performs search and insertion in list operations.

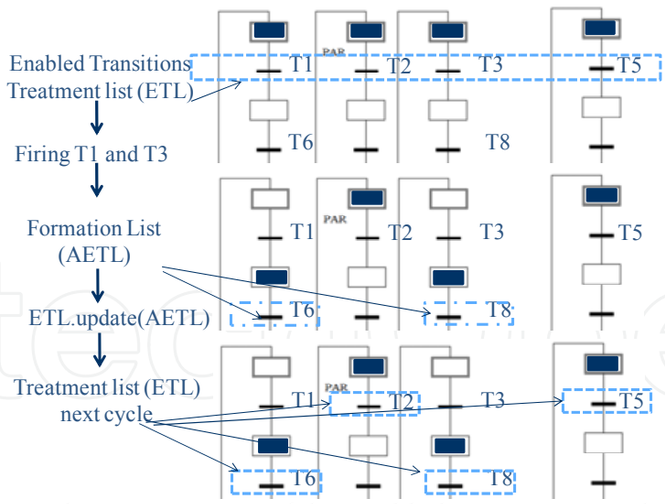

Fig. 4. Treatment List and Formation List of the Enabled Transitions Technique

4.3 Static Representing Places Technique

Program 2 presents the basic treatment cycle of the coordinator for the SRP technique. This treatment cycle is also illustrated in Fig. 5.

```
loop forever
  Executeactionswithfallingedge();
```

```
  Executeactiveactions();
  while elements in ARSL do
    Rstep = next_element (ARSL);
    Transitionsrepr = Rstep.transitionsrep;
// enabling test
    while T in Transitionsrepr do
      if enabled (T) and predicate(T) then
// start of transitions firing
       Demark_input_steps (T, ARSL, ASSL); // ARSL and ASSL updating
        Add(Transitionsfired , T);
         Break ();
      end if;
    end while ;
  end while ;
// end transitions firing
  while T in Transitionsfired do
    Mark_output_steps(T, ARSLnext, ASSLnext);
              // ARSLnext and ASSLnext updating
              // involves search and insertion in list operations
  end while ;
  Clear(Transitionsfired);
//list update
  Update(ARSL, ARSLnext);
       // involves search and insertion in list operations
  Update(ASSL, ASSLnext);
       // involves search and insertion in list operations
  Clear(ARSLnext); Clear(ASSLnext);
  Updateactivityconditions();
end loop ;
```

Program 2. SRP Treatment Loop

The following data structures will be available (see Fig. 5):

- Active Representing Steps list (ARSL) and Active Synchronization Steps list (ASSL). Treatment lists containing the active Representing and Synchronization Steps.
- Active Representing Steps list (ARSLnext) and Active Synchronization Steps list (ASSLnext). Formation lists with the Steps that will be active in the next cycle by the firing of the transitions.

For each representing step a data structure is necessary that contains:

- List of transitions represented by the Step

In all the transitions of the SFC a data structure will be necessary that stores:

- Representing step
- List of synchronization steps
- List of transitions in conflict

- List of active representing steps after firing
- List of active synchronization steps after firing

In each cycle only the output transitions of an active representing step are verified for enabling. If a represented transition fires, the verification process for the representing step ends because the rest of the represented transitions become disabled (they are in effective conflict).

At the start of the transitions firing phase the function `Demark_input_steps (T, ARSL, ASSL)` encapsulates the deactivation of the input steps of the transition fired, and the updating of the ARSL and ASSL lists. The deactivated steps should be removed from the ARSL (if it is the representing step of the transition) or from ASSL (if it is a synchronization step of the transition). These fired transitions are added to the list `Transitionsfired`.

At the end of the transitions firing phase the function `Mark_output_steps (T, ARSLnext, ASSLnext)` encapsulates the activation of the output steps of the transition fired and the building of the lists ARSLnext and ASSLnext. The output steps of the transitions in the `Transitionsfired` list are activated. The activated steps should be added to the list ARSLnext (if it is the representing step) or to ASSLnext (if it is a synchronization step). At this point, the algorithm performs search and insertion in list operations.

At the end of the cycle, the ARSL list is updated in `Update (ARSL, ARSLnext)`. The ARSLnext elements are added to the ARSL (if they do not already belong). The ASSL list is also updated in `Update(ASSL, ASSLnext)`. The ASSLnext elements are added to the ASSL (if they do not already belong). At this point, the algorithm also performs search and insertion in list operations.

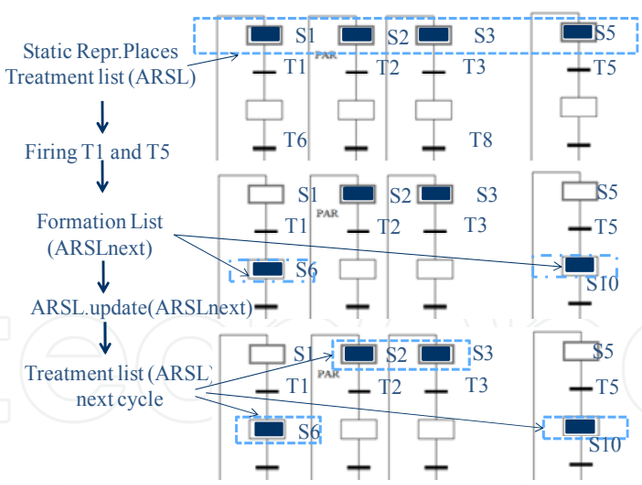

Fig. 5. Treatment List and Formation List of the Representing Places Technique

4.4 Deferred Transit Evolution Model Technique

Program 3 presents the basic treatment cycle of the coordinator for the DTEVM technique. The following data structures will be available (see Fig. 6):

- Active Steps list (ASL). Treatment lists containing all the active steps.
- Enabled Transitions List (ETL). Treatment lists containing the transitions with their input steps active and with their predicate condition true.

This treatment cycle is also illustrated in Fig. 6. The search of the Active Steps is carried out in DTEVM at the start of each cycle, in the function computeactivesteps. The execution time of this search function is proportional to the number of steps of the SFC.

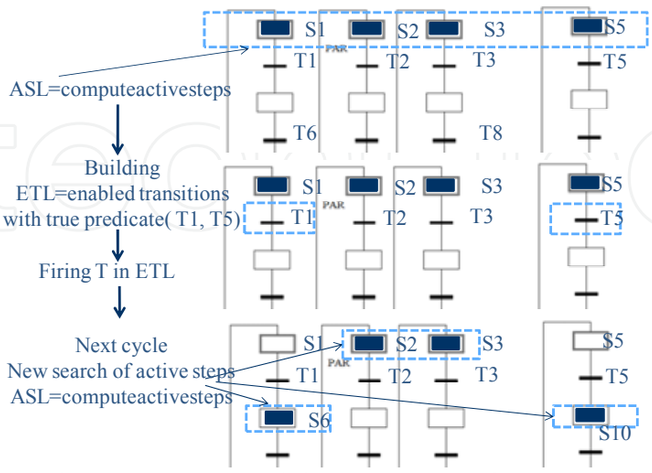

Fig. 6. Treatment List and Formation List of the DTEVM Technique

The enabling test of the transitions is carried out in two phases. First, it finds the enabled transitions with true predicates that are output of the steps in the ASL list, drawing up the ETL list. It then goes through this list and fires the transitions. The enabling must be re-evaluated to prevent the firing of transitions in conflict. This algorithm does not perform any search and insertion in list operations.

```
loop forever
  ASL=computeactivesteps();
// enabling test
  while elements in ASL do
    Activestep = next_element (ASL);
    Transoutput= Activestep.Transoutput;
    while T in Transoutput do
      if enabled (T) and predicate(T)then Add(ETL, T); end if;
    end while ;
  end while ;
  Executeactionswithfallingedge();Executeactiveactions();
// start transitions firing
  while T in ETL do
    if enabled(T) then
      Demark_input_steps(T);Add(Transitionsfired, T);
    end if;
  end while ;
// end transitions firing
  while T in Transitionsfired do
    Mark_output_steps(T);
```

```
  end while ;
  Clear(Transitionsfired);Updateactivityconditions();
end loop;
```

Program 3. DTEVM Treatment Loop

5. Estimation of the execution time of the algorithms.

An analysis of SFC implementation algorithms was carried out in (Piedrafita & Villarroel 2008 a). Brute Force (BF), Enabled Transitions (ET), Static Representing Places (SRP) Inmediate Transit Evolution Model (ITEVM) and Deferred Transit Evolution Model (DTEVM) were analyzed. The main ideas obtained in (Piedrafita & Villarroel 2008 a) are:

- The implementation of the Enabled Transitions and Static Representing Places algorithms can lead to enormous savings in execution time compared to the Brute Force algorithm.
- The choice of the most suitable type of algorithm to execute a SFC depends on the SFC behavior (effective concurrency vs. effective conflicts).

The presented tests show that the relative performance of implementation algorithms depends on the model concurrency structure but also on the dynamics imposed by the controlled system. In most of the cases, the SRP and the ET algorithms coming from PN field have good behaviors. The PN implementation techniques provide an improvement in the development of industrial controllers based on SFC language.

The execution of SFCs without a suitable algorithm can suppose an increasing of the computing time, and a worse and slower answer in control applications. It is very difficult to estimate what algorithm will run faster an SFC. In real-time control only one algorithm can run the SFC, thus it must be possible to estimate what would be the execution time of the other alternative non executed algorithms

The execution time, given its ease of measuring, is the physical parameter that most easily allows the performance of an algorithm to be evaluated. However, the execution time must be considered as an explicit measure of the performance of an algorithm, where it directly reflects the influence of the other parameters.

The execution time of the algorithms described in the previous section will depend on the number of transitions tested for enabling in each cycle, and on the number of search and insertion in list operations. The computation time of the test for enabling operations does not depend on the size of the SFC. However, the computation time of the search and insertion in list operations does depend directly on the size of the algorithm lists.

The number of transitions tested for enabling in the ET technique is the sum of the sizes of ETL and AETL. For the SRP technique, the number of transitions tested for enabling start from a minimum, being the number of Active Representing Steps (if firing even the first transition represented) to a maximum, being the total of the transitions represented by the Active Representing Steps (if firing even the last transition represented or if there is no firing transition). For the DTEVM technique, the number of transitions tested start from a minimum, the total of the output transitions of the Active Steps, to a maximum, twice the total of the output transitions of the Active Steps (if all predicates are true).

One of the most expensive operations in execution time is the *search and insertion in lists*. The presented techniques frequently use this type of operation, especially in the real time building of formation lists and in the final phase of updating lists. The execution time of

such operations depends directly on the size of the lists. There are techniques that abound in the use of *search and insertion list* operations, such as Representing Places. In other techniques such as Brute Force this type of operation is not performed since the lists are not updated.

The search and insertion in list operations are performed in techniques such as ET or SRP because of the managing in real time of the treatment and formation lists. In the algorithms such operations are performed at the end of the firing of transitions and in the final update of the lists. Hence, if no transitions fire, the number of such operations is null. In the ET technique, the number of this kind of operation is the number of transitions of AETL that are enabled and become part of ETL. In SRP, it is twice the number of Steps that become active in the transitions firing, because SRP manages four lists. The computation time of the search and insertion in list operations depends directly on the size of the lists.

The SFC implementation techniques are based on a cyclic treatment (see Program 1 to 3). The main loop goes through the treatment and formation lists using an algorithm that depends on the executed technique. The algorithm cycle execution time depends on the size of the treatment and formation lists. The size of the treatment lists in the case of ET and SRP depends on the current SFC state. This determines the number of enabled transitions and the number of active representing steps. The size of the formation lists depends on the number of transitions that fire in the cycle. Thus, the execution time depends on the evolution of the SFC state, the SFC structure and the sequence of events.

As algorithms use different lists, their execution times will be different. The estimation of the algorithm execution time is based on the measurement of the mean time taken by these loops and on the estimation in real time of the size of the treatment and formation lists.

First, we study the SRP algorithm. The cycle execution time (CET) can be estimated by the following expression:

$$\begin{aligned} CET(SRP) = {} & Tenabl * SIZE(ARSL) * TR_{TESTED} + Tfiring * FT_{NUMBER} + \\ & TinsertStep * (SIZE(ARSL_{NEXT})/2) * (SIZE(ARSL_{NEXT})) + TinsertStep \\ & * (SIZE(ASSL_{NEXT})/2) * SIZE(ASSL_{NEXT})) + TinsertStep * \\ & (SIZE(ARSL_{NEXT}) * (SIZE(ARSL)) + TinsertStep * (SIZE(ASSL_{NEXT})) * SIZE(ASSL)) \end{aligned} \quad (1)$$

Where FTnumber is the number of fired transitions; Trtested is the mean represented transitions tested of an active representing step; Tenabl is the time for the enabling test operation of one transition; Tfiring is the mean time for firing one transition; TinsertStep is the mean time necessary for the search and insertion in list operation of one Step in a List of size one (performed in the final phase of updating list).

The ET algorithm is also analyzed. The cycle execution time can be estimated by the following expression:

$$\begin{aligned} CET(ET) = {} & Tenabl * (SIZE(ETL) + SIZE(AETL)) + Tfiring * FT_{NUMBER} + \\ & Tinserttran * SIZE(AETL) * SIZE(ETL) \end{aligned} \quad (2)$$

Where TinsertStep is the mean time necessary for the search and insertion in list operation of one transition in a List of size one (performed in the final phase of updating list).

Establishing expressions for other implementation techniques is not complicated. Let us consider, for example, the brute force technique. The cycle execution time expression of the BF algorithm is:

$$CET(BF) = Tenabl * size(TL) + Tfiring * FTnumber \quad (3)$$

TL is the list with all transitions of the SFC.

6. The SFC Execution Time Controller

With the objective of minimizing SFC execution time, we decided to design a Supervisor controller which we have called the Execution Time Controller (ETC). The first version of the ETC is presented in (Piedrafita & Villarroel 2008 b).
The main function of the ETC is to determine in real time which algorithm executes a SFC fastest. The ETC executes the algorithm chosen and estimates the execution time of the other non-executed algorithms, choosing the best algorithm in line with the controlled system. If necessary, the ETC changes the algorithm. In the next section we present in detail how the execution time (ExT) of the running and the alternative algorithms are estimated. To avoid the overload of continuous algorithm changes, an integral cost function is used:

$$\varepsilon = ExT_{calculated}(running\_alg) - ExT_{estimated}(alternative\_alg) \quad (4)$$

$$I(k) = \begin{cases} I(k-1) + \varepsilon(k) & if \quad I(k-1) + \varepsilon(k) > 0 \\ 0 & if \quad I(k-1) + \varepsilon(k) \le 0 \end{cases}$$

The change is made when I(k) is greater than half of the execution time of the executed algorithm. When a change happens, I (k-1) = 0.

```
// Offline Control
Load SFC
Measuring Times
First Choice of the best algorithm
Return to initial steps
// Online Control
loop forever
  Read Inputs
  SFC execution with the best algorithm
  Write Outputs
  Compute execution time of running_alg
  Estimate execution time of alternate_alg
  Compute I(k)
  If I(k)>(ExT_calculated(running_alg)/2) then
    Change algorithm
    Initialize data structures
    I(k-1)=0
  End if
  Wait for next period();
end loop
```

Program 4. Execution Time Controller

6.1 Times measuring.

The Tenabl, Tfiring, TinsertStep and Tinserttran times are measured in an offline execution test. For this purpose, the required measurement instrumentation is incorporated into the program. This instrumentation comprises the instructions required for reading the real time system clock and the necessary time difference calculations.

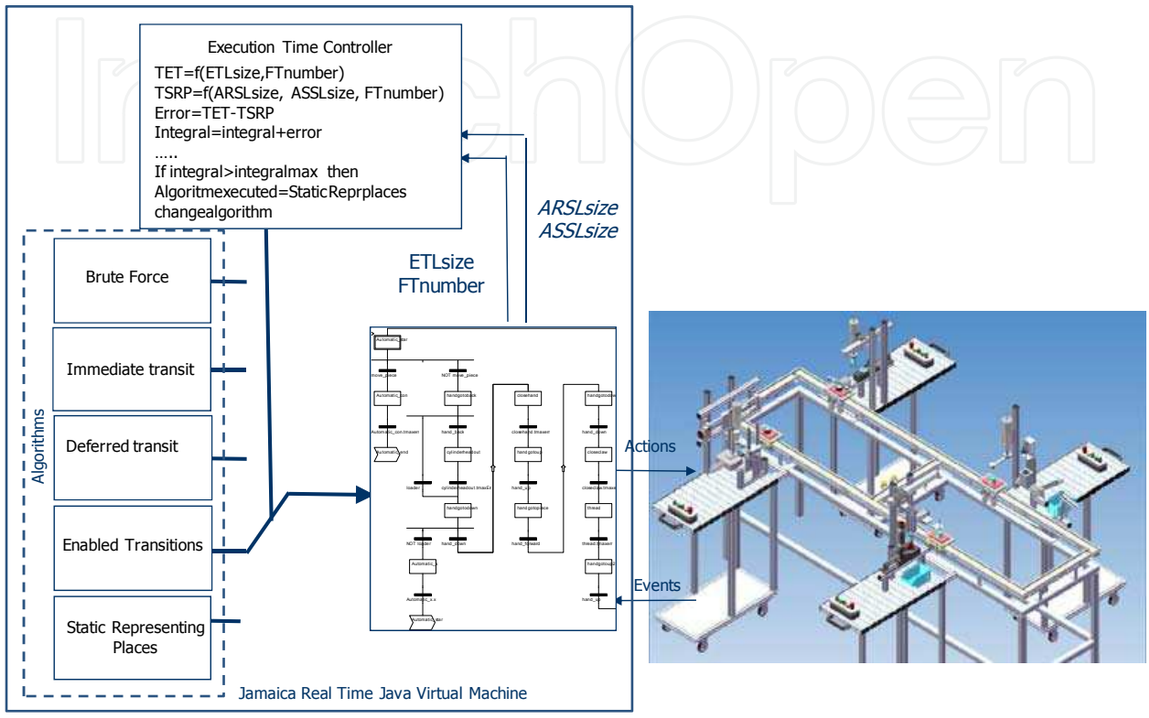

Fig. 7. Execution Time Controller

The Time measuring test consists of the execution of the SFC with one of the algorithms carrying out the firing of 2000 transitions without executing the programmed actions. The condition associated with the transitions is considered true so that the firing is immediate. If there are conflicts, the transition that fires is chosen at random.

For example, if the execution is done with the SRP algorithm during the test, the measurement of the Tenabl, Tfiring and TinsertStep times will proceed. The test is then repeated with the ET algorithm, and the Tinserttran time measurement is carried out.

The cycle times of each algorithm are also measured in these tests and the algorithm with the shortest cycle time is chosen for the first execution of the SFC with control actions.

6.2 Estimation of Data structure size.

In the real time execution of the ETC (see Program 4), the execution time of the executed algorithm can be measured by reading the system clock. To avoid an overload of the control actions, the execution time of the executed algorithm (runnig_alg) is then calculated with equations (1) to (3) (this depends on the algorithm being executed). In this case FTNUMBER and the sizes of the lists (ASRL, ASRLNEXT, …) are known by the ETC.

The ETC should obtain enough information to determine what would be the computation time of the other algorithms not executed at that moment. There is no problem with the algorithm being executed. For the other algorithms, should be obtained the size in real time of the treatment and formation lists if they were being executed. From this data it can be estimated what would be their computation time. This estimation should be carried out in real time and with an small overload in the execution time of the algorithm.
The execution times of the alternative algorithms should also be estimated with equations (1) to (3). The number of transitions fired is known given that it is the same as for the algorithm that is being executed. The value of the times measured in the test is also known. However, the size of the other lists must be estimated.
For example, in the execution of the ET algorithm, the size of the lists of the SRP algorithm must be estimated. The mean number of active representing steps and active synchronization steps is more or less constant in most SFCs; therefore, the size (ARSL) and the size (ASSL) will be the mean value estimated in the offline time measurement test.
Consequently, it can be stated that, on average, the firing of a transition involves the unmarking of its representing Step and the marking of a new one. The size (ARSL$_{NEXT}$) can be approximated by the number of transitions fired.

$$\text{size (ARSL}_{NEXT}) \approx \text{FT}_{NUMBER} \tag{5}$$

The size(ASSL$_{NEXT}$) can be approximated by the expression:

$$\text{size(ASSL}_{NEXT}) \approx \text{FT}_{NUMBER} * (f_{p\_}1) \tag{6}$$

fp is the average parallelism factor (number of output places of a transition) of the fired transitions.

$$\begin{aligned}\text{CET(SRP)= Tenabl *size(ARSL)*TRtested+ Tfiring * FT}_{NUMBER} + \text{TinsertStep *}\\ \text{(FTnumber /2)* FT}_{NUMBER} + \text{TinsertStep * FT}_{NUMBER} * (\text{fp}-\\ 1)/2)*\text{size(ASSLnext))+ TinsertStep * FT}_{NUMBER} *(\text{size(ARSL)) + TinsertStep *}\\ \text{FT}_{NUMBER} * (\text{fp}-1)*\text{size(ASSL))=}\\ \text{F1 (size(ARSL),size(ASSL), FT}_{NUMBER})\end{aligned} \tag{7}$$

To estimate the size of the lists of the SRP algorithm when the algorithm executed is FB, the same technique is used given that in real time it is only necessary to know the number of transitions fired and the mean parallelism factor of these transitions.
If the SFC is executed with the SRP algorithm, it should be estimated what would be the computation time of the execution of the SFC with the ET algorithm. Therefore the sizes of the ETL and AETL lists should be estimated. The FT$_{NUMBER}$ is known because the two algorithms make the SFC evolve in the same way and therefore the number of fired transitions will be the same for both.
The size of the ETL list is estimated in the SRP execution when the sensibilization of the transitions represented by the active representing steps is tested. When the SRP algorithm finds an enabled transition it fires it and continues with the next active representing step. If, therefore, it is necessary to know how many transitions there are enabled among those represented by the representing place, two possible solutions are adopted:

- A first option is that the algorithm runs over all the transitions represented by a marked representing place, estimating their enabling.

- When the SRP algorithm finds an enabled transition it fires it, and the rest of represented transitions are not verified for enabling. An approximation is carried out considering enabled half of the rest of transitions.

The second solution was chosen for the tests given that the computation time is shorter. The size of the AETL list is estimated at the firing of the transitions, when the output steps are activated, as is the size of the set of output transitions of the output steps of the fired transitions.

$$\text{SIZE(AETL)} = \text{FTNUMBER} * f_p * f_d \tag{8}$$

fp is the parallelism factor (number of output steps of a transition) of the transitions fired and fd is the descendants factor (average number of output transitions of a step) of the steps activated in the transitions firing.

$$\begin{aligned} \text{CET(ET)} &= \text{Tenabl} * (\text{size(ETL)} + \text{size (AETL)}) + \text{Tfiring} * \text{FTNUMBER} + \\ \text{Tinserttran} * \text{size(AETL)} &* \text{size(ETL)} = \text{Tenabl} * (\text{size(ETL)} + \text{FTNUMBER} * f_p * f_d + \\ \text{Tfiring} * \text{FTNUMBER} &+ \text{Tinserttran} * \text{FTNUMBER} * f_p * f_d * \text{size(ETL)} = \\ & F_2(\text{size(ETL)}, \text{FTNUMBER}) \end{aligned} \tag{9}$$

A different technique is used to estimate the size of the ET algorithm lists when the FB algorithm is executed. Because with FB all the transitions in the SFC are covered in the enabling test, the size of the ETL list can be accurately known. In the first version of the ETC (Piedrafita & Villarroel 2008 b) , it was necessary to measure the size of the treatment and formation lists. In this second version of the ETC is only necessary to measure the size of the treatment lists, since the size of the formation lists is calculated from the number of transitions fired FTNUMBER.

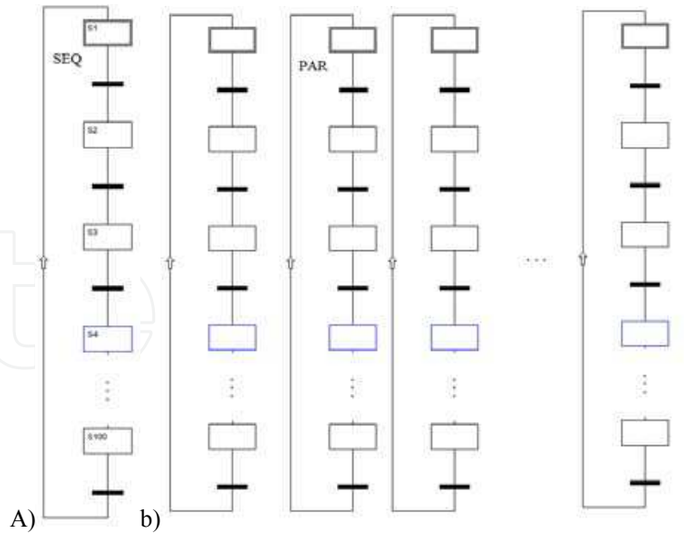

Fig. 8. SFCs Library

7. Technique Evaluation

7.1 Execution Platform

We have implemented the techniques in the Java language using the Java Real-time Specification (Bollella & Gosling 2000.) and following some ideas presented in (Piedrafita & Villarroel 2006a), (Piedrafita & Villarroel 2006b) and (Piedrafita & Villarroel 2007). In our implementations, we used the Real Time Java Virtual Machine JamaicaVM v2.7 (Aicas 2007). The target hardware was a personal computer with a Pentium IV processor at 1.7GHz, running Red Hat Linux 2.4. The Coordinator is implemented as a Periodic Real Time Thread of high Priority. The execution is made in a single processor and threads are scheduled following a static priorities policy without round-robin.

In the implementations developed here, the program loads the SFC structure from an XML file generated by an SFC editor. The implementation is independent of the SFC, and is therefore an interpreted implementation.

A library of Sequential Function Charts has been developed for carrying out the tests. The library is based on four base models which can be scaled using a parameter. These models represent most of the cases developed in industrial control: sequential systems and concurrent systems. The library comprises the following SFCs:

- SEQ. SFC with one sequential process composed of 1 to 100 steps (Fig. 8.a).
- PAR. SFC with p (1..100) sequential processes with 20 steps (Fig. 8.b).

7.2 Real Time Execution of ETC

The ETC controller has been tested with all the SFCs in the library and also with a real control application. This is a Flexible Manufacturing Cell in the Computer Science Department of the University of Zaragoza. The ETC obtains a high degree of success in all of the performed experiments, but here for the sake of brevity we present the results of the three most representative experiments. However, an exhaustive report of the experiments can be consulted in (Piedrafita 2008), accessible via the Internet.

The execution of the ETC takes place in the Real Time Java Virtual Machine Jamaica. This is implemented as Periodic Real Time Thread of high Priority with 20 ms of period. The execution is made in a single processor and threads are scheduled following a static priorities policy without round-robin.

The first experiment that we present is over a sequential SFC of 35 steps (see Fig. 9 left) and illustrates that the SRP algorithm is always the best in this experiment (Piedrafita & Villarroel 2007).

Fig. 9.a, shows the Real Time execution of The Execution Time Controller (ETC), the Real Time estimation of the same algorithm (SRP), and the Real Time estimation of one alternative algorithm(ET in this case). Fig. 9.b, shows also the Real Time execution of the algorithm SRP, and the Real Time execution of the algorithm ET.

The ETC chooses the SRP from the start since the estimation of the execution time of this algorithm is smaller than that of the ET algorithm. Because SRP is always better than ET, the integral cost function I(k) remains permanently null and therefore no algorithm change is performed.

The second experiment that we present is over a concurrent SFC compound of 10 sequential SFCs (see Fig. 9 c and d) and illustrates that the SRP algorithm is the best in this experiment (Piedrafita & Villarroel 2007) .

Fig. 9.c, shows the Real Time execution of The Execution Time Controller (ETC), the Real Time estimation of the same algorithm (SRP), and the Real Time estimation of one alternative algorithm(ET in this case) and the integral cost function I(k). Fig. 9.e, shows also the Real Time execution of the algorithm SRP and the Real Time execution of the algorithm ET.

As in the first experiment, the ETC chooses the SRP from the start since the estimation of the execution time of this algorithm is smaller than that of the ET algorithm. Because SRP is always better than ET, the integral cost function I(k) remains permanently null and no algorithm change is performed.

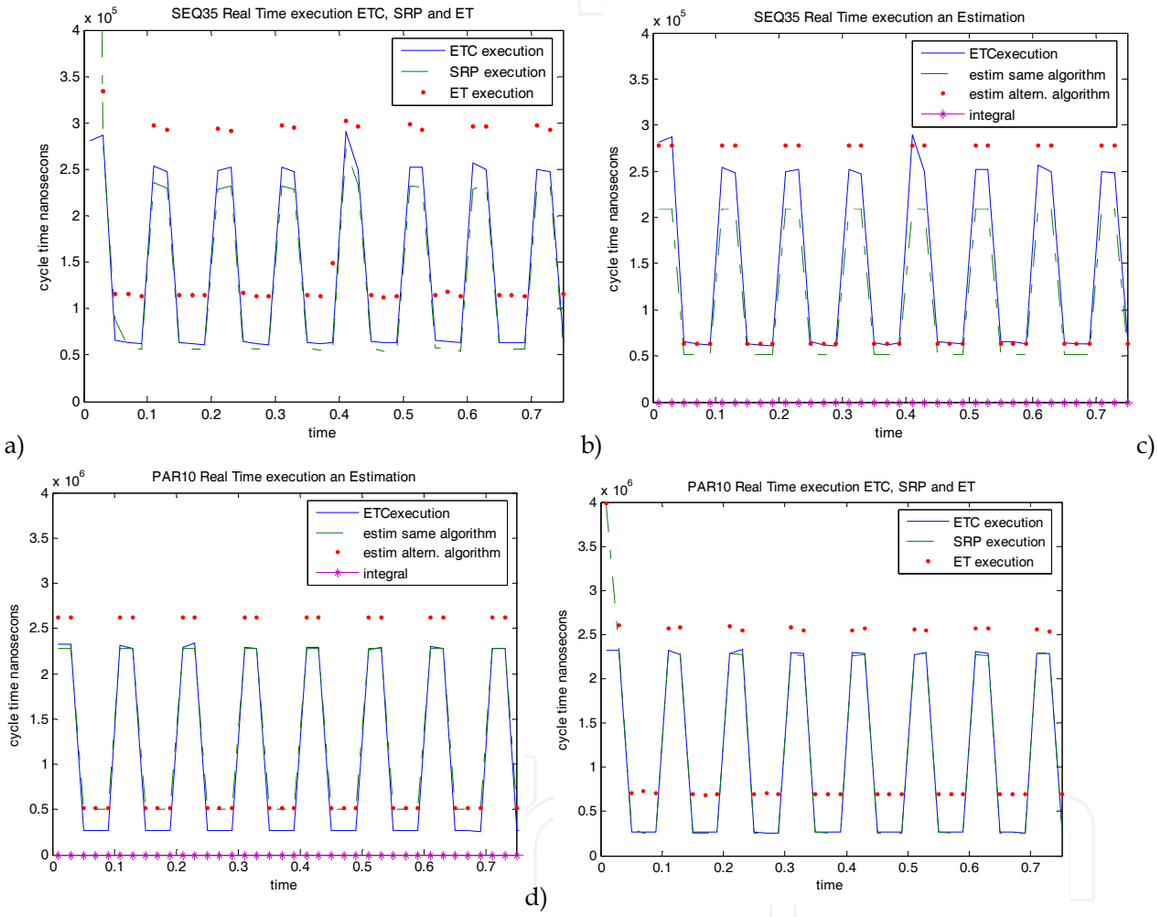

Fig. 9. Real Time Execution of the ETC with a sequential SFC of 35 states and a concurrent SFC compound of 10 sequential SFCs.

The third experiment that we present is over a concurrent SFC compound of 40 sequential SFCs (see Fig. 10 a and b) and illustrates that the SRP algorithm is the best algorithm in this experiment when not firing transitions, and ET is the best algorithm when all possible transitions are firing.

In the first 0.4 seconds all possible transitions fire and the best algorithm is ET, while in the next 0.4 seconds no transitions fire and the best algorithm is SRP. The event sequence is cyclically repeated.

At the beginning, the ETC executes the ET algorithm which, as we have seen, is the better algorithm in the first 0.4 seconds. However, at the instant 0.4 no events reach the SFC and SRP becomes better, therefore I(k) increases and the ETC changes to SRP at instant 0.48. At instant 0.8, the events reach the SFC and ET becomes better, therefore I(k) increases again and the ETC changes to ET at instant 0.9. For the whole evolution, ETC changes to SRP at instants 0.48, 1.28 and 2.08 and to ET at instants 0.9, 1.7 and 2.5. With the observed behaviour, the ETC achieves the minimum possible execution time of the SFC.
In industrial control it is very common that, in many cycles, events do not reach the SFC, and so no transition is fired, and when they are fired their quantity is variable. We can therefore differentiate between two operation regimes:

- Without events regime (static). No transitions are fired and the algorithm only runs the enabling test.
- With events regime (dynamic). Transitions are fired and the algorithm must run all the phases: enabling test, firing and updating of lists.

If desired, the ETC can choose the algorithm that has the shortest computation time in the enabling test (without events regime). The integral cost function is only calculated when no transitions are fired. The integral cost function is:

$$\varepsilon = ExT_{calculated}(running\_a\lg) - ExT_{estimated}(alternative\_a\lg)$$

If (FTnumber==0) {

$$\varepsilon = ExT_{calculated}(running\_a\lg) - ExT_{estimated}(alternative\_a\lg) \tag{10}$$

$$I(k) = \begin{cases} I(k-1) + \varepsilon(k) & if \quad I(k-1) + \varepsilon(k) > 0 \\ 0 & if \quad I(k-1) + \varepsilon(k) \le 0 \end{cases}$$

}

The execution of the ETC with this cost function can be seen in Fig. 10 c and d. It can be observed that the integral is only calculated when no transitions fire. At the instant 0.48 the ETC changes to the SRP algorithm, which has the shortest computation time in the without events regime.
If it is required that the ETC achieve the shortest reaction time for events, the integral cost function is only calculated when transitions fire. In this way the ETC chooses as the best algorithm that which has the shortest computation time in the firing of transitions. The integral cost function is:

$$\varepsilon = ExT_{calculated}(running\_a\lg) - ExT_{estimated}(alternative\_a\lg)$$

If (FTnumber >0) {

$$\varepsilon = ExT_{calculated}(running\_a\lg) - ExT_{estimated}(alternative\_a\lg) \tag{11}$$

$$I(k) = \begin{cases} I(k-1) + \varepsilon(k) & if \quad I(k-1) + \varepsilon(k) > 0 \\ 0 & if \quad I(k-1) + \varepsilon(k) \le 0 \end{cases}$$

}

The execution of the ETC with this function is shown in Fig. 10 e and f. It can be seen that the integral is only calculated when transitions are fired. At the instant 0.48 el ETC chooses the ET algorithm, which has the shortest computation time in the with events regime.

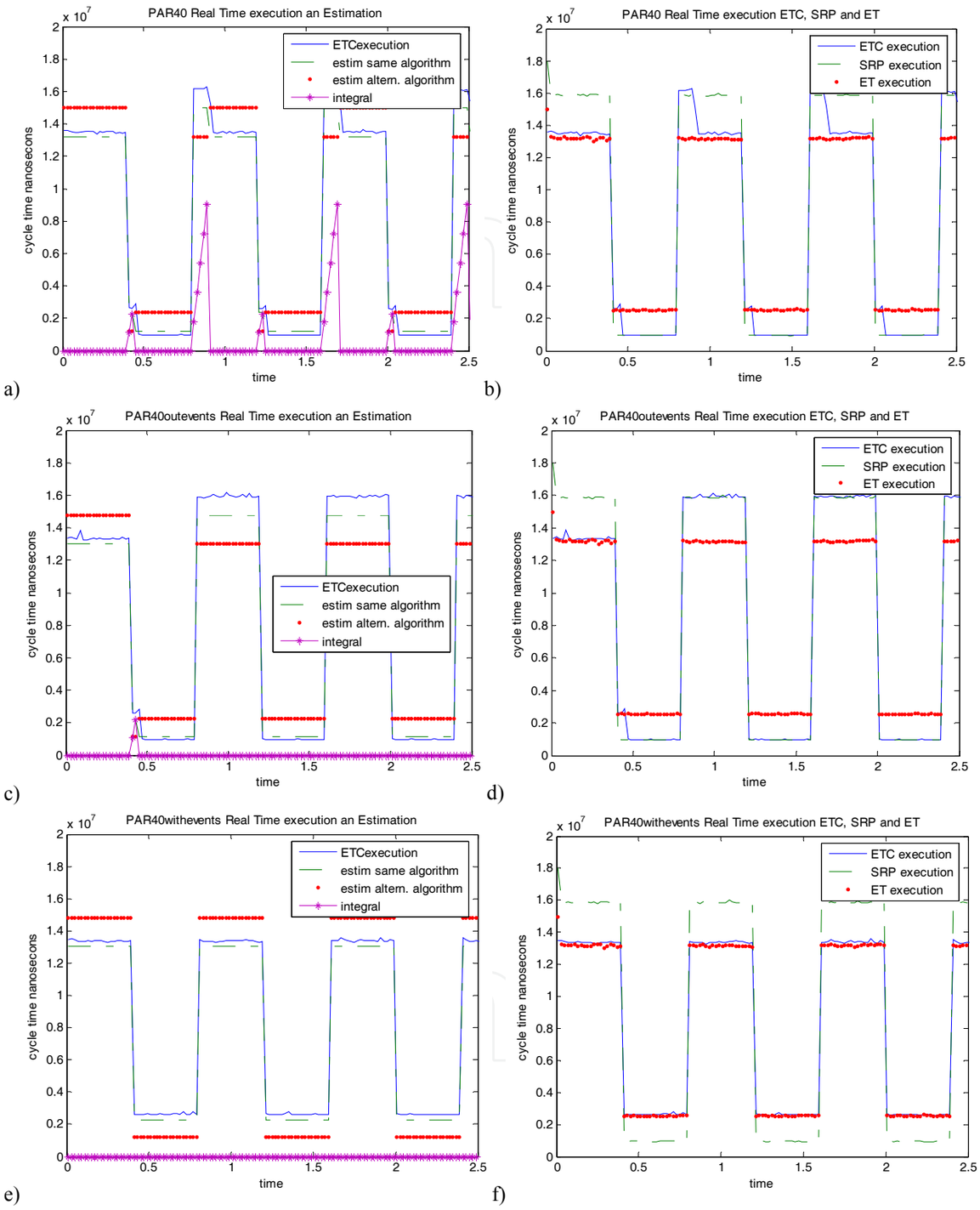

Fig. 10. Real Time Execution of the ETC with a concurrent SFC compound of 40 sequential SFCs.

8. Conclusions

In this work we have developed an adaptive implementation of Discrete Event Control Systems, the Execution Time Controller, which allows choosing in real time the most suitable algorithm to execute a Sequential Function Chart. The main function of the ETC will be to determine which algorithm executes a SFC the fastest. The proposed technique is analyzed with the two most important algorithms (from the point of view of performance): the enabled transitions and the static representing places. However, the ETC can work with any SFC implementation algorithm.

The ETC executes the chosen algorithm and estimates the execution time of other non-executed algorithms, deciding the best one in line. The execution of a SFC without a suitable algorithm can lead to a significant increase in the execution time, together with a less satisfactory and slower answer in control applications. The technique has been tested on a wide SFC library. Moreover, the ETC has also been tested in a real control application. The technique has a high success rate in the choice of the best implementation algorithm.

In the first version of the ETC, it was necessary to measure the size of the treatment and formation lists. In this second version of the ETC is only necessary to measure the size of the treatment lists, since the size of the formation lists is calculated from the number of transitions fired FTNUMBER

The ETC ensures that the control system to react in the shortest time possible, increasing quality control. In many cases this can also reduce the period of the task which implements the SFC interpreter. This implies an increase in quality control. This reduction in execution time may also be allocated to the execution of other tasks with heavy run-time such as the graphical interface or communications.

The ETC allows faster reaction times in SFC based control systems and also minimizes the power consumed by the controller.

9. References

Aicas, (2007). JamaicaVM Realtime Java Technology. http://www.aicas.com/jamaica.html.

Bollella, G. & J. Gosling (2000.). "The Real-time Specification for Java." Computer 33(6): 47-54.

Briz, J. L. & J. M. Colom (1994). "Implementation of Weighted Place/Transition Nets based on Linear Enabling Functions." Application and Theory of Petri Nets 815: 99–118.

Briz., J. L. (1995). "Técnicas de implementación de redes de Petri. ." PhD thesis, Univ. Zaragoza.

Bruno, G. & G. Marchetto (1986). "Process-translatable Petri nets for the rapid Prototyping of Process-Control Systems." Ieee Transactions on Software Engineering 12(2): 346-357.

Cassandras, C. G. (1993). Discrete event systems, Springer.

Colom, J. M., M. Silva, et al. (1986). "On software implementation of Petri nets and colored Petri nets using high-level concurrent languages." Seventh European Workshop on applications and theory of Petri nets, Oxford, July 86: 207-241.

David, R. (1995). "GRAFCET - A powerfull Tool for specification of logic controllers." Ieee Transactions on Control Systems Technology 3(3): 253-268.

Garcia, F. J. & J. L. Villarroel (1999). Translating time Petri net structures into Ada 95 statements. Reliable Software Technologies - Ada-Europe' 99. Berlin, Springer-Verlag Berlin. 1622: 158-169.

Hellgren, A., M. Fabian, et al. (2005). "On the execution of sequential function charts." Control Engineering Practice 13(10): 1283-1293.

IEC (1988). Preparation of Function Charts for Control Systems publication 848.

ISO/IEC (2001). "International standard IEC 61131-3 (2nd ed.). Programmable logic controllers—Part 3. ISO/IEC (final draft). ."

Klein, S., G. Frey, et al. (2003). PLC Programming with Signal Interpreted Petri Nets. ICATPN 2003, Eindhoven, Srpinger Verlag.

Lewis, R. W. (1998). "Programming industrial control systems using IEC 1131-3." IEEE control engineering series 50.

Peng, S. S. & M. C. Zhou (2004). "Ladder diagram and Petri-net-based discrete-event control design methods." Systems, Man and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on 34(4): 523 – 531.

Piedrafita, R. (2008). Experimentation with the Execution Time Controller (Research Report). http://automata.cps.unizar.es/realtime. Zaragoza, Departamento de Informática e Ingeniería de Sistemas.: 176.

Piedrafita, R. & J. L. Villarroel (2006 a). "Implementation of time petri nets in real-time Java." Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems: 178-187.

Piedrafita, R. & J. L. Villarroel (2006 b). Petri Nets and Java. Real-Time Control of a flexible manufacturing cell. Emerging Technologies and Factory Automation, 2006. ETFA'06. IEEE Conference on: 1246-1253.

Piedrafita, R. & J. L. Villarroel (2007). "Performance evaluation of petri nets execution algorithms." Systems, Man and Cybernetics, 2007. ISIC. IEEE International Conference on: 1400-1407.

Piedrafita, R. & J. L. Villarroel (2008 a). Evaluation of Sequential Function Charts Execution Techniques. The Active Steps Algorithm. Emerging Technologies and Factory Automation. IEEE Conference on. Hamburg, Germany.

Piedrafita, R. & J. L. Villarroel (2008 b). Adaptive Petri Nets Implementation. The Execution Time Controller. 9th International Workshop on Discrete Event Systems 2008, Gothenburg, Sweden.Silva, M. (1985). Las Redes de Petri en la Automatica y la Informatica. Editorial AC, Madrid, 1985, Spanish.

Silva, M. and S. Velilla (1982). "Programmable logic controllers and Petri nets: A comparative study." IFAC/IFIP Symposium on Software for Computer Control, Madrid, Spain, October 1982: 83–88.

Taubner, D. (1988). "On the implementation of Petri nets." Lecture Notes in Computer Science 340: 418-439.

Uzam, M. and A. H. Jones (1996). Towards a Unified Methodology for Converting Coloured Petri Net Controllers into Ladder Logic Using TPLL: Part I - Methodology. International Workshop on Discrete Event Systems - WODES'96. Edinburgh, UK: 178 - 183.

Factory Automation

Edited by Javier Silvestre-Blanes

ISBN 978-953-307-024-7
Hard cover, 602 pages
Publisher InTech
Published online 01, March, 2010
Published in print edition March, 2010

Factory automation has evolved significantly in the last few decades, and is today a complex, interdisciplinary, scientific area. In this book a selection of papers on topics related to factory automation is presented, covering a broad spectrum, so that the reader may become familiar with the various fields, and also study them in more depth where required. Within various chapters in this book, special attention is given to distributed applications and their use of networks, since it is one of the most relevant subjects in the evolution of factory automation. Different Medium Access Control and networks are analyzed, while Ethernet and Wireless networks are looked at in more detail, since they are among the hottest topics in recent research. Another important subject is everything concerning the increase in the complexity of factory automation, and the need for flexibility and interoperability. Finally the use of multi-agent systems, advanced control, formal methods, or the application in this field of RFID, are additional examples of the ideas and disciplines that experts around the world have analyzed in their work.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Ramon Piedrafita and Jose Luis Villarroel (2010). Adaptive Implementation of Discrete Event Control Systems based on Sequential Function Charts, Factory Automation, Javier Silvestre-Blanes (Ed.), ISBN: 978-953-307-024-7, InTech, Available from: http://www.intechopen.com/books/factory-automation/adaptive-implementation-of-discrete-event-control-systems-based-on-sequential-function-charts

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen