56TH INTERNATIONAL SCIENTIFIC COLLOQUIUM
Ilmenau University of Technology, 12 – 16 September 2011
URN: urn:nbn:gbv:ilm1-2011iwk:5

A FORMAL APPROACH FOR AEROSPACE SYSTEMS CONTROL CONSIDERING SFC SPECIFICATION AND C PROGRAMMING LANGUAGE

P. Borges / J. Machado / E. Seabra / L. F. Silva

Mechanical Engineering Department, CT2M Research Centre,
School of Engineering, University of Minho
Campus de Azurém, 4800-058 Guimarães, Portugal
Phone: +351 253 510220
Fax: +351 253 516007
Email: pborgesmail@gmail.com; {jmachado, eseabra, lffsilva}@dem.uminho.pt

ABSTRACT

The C programming language is one of the most used in critical embedded real-time controllers applied at aerospace systems. Despite its potential, it is a very general language, with many maintenance problems and with a little or without graphical structure. The absence of formal verification techniques - even if it is possible to find some works associated to C programming language formal verification - is a fact.
In this paper, it is proposed a methodology, that is divided in two main steps, and has, as main goal, to obtain safe C program code from a SFC specification: in first step some tools and techniques are used in order to assure the quality of the SFC specification and, on the second step, the goal is to translate (in a systematic way) the safe SFC specification to C code considering crucial aspects like taking into account aspects related with time specification.

Index Terms - Dependable Systems; C code, SFC; Safe Controllers; Real Time Embedded Systems

1. INTRODUCTION

Aerospace systems software is developed taking into account some precautions to avoid dangerous situations. Usually the controllers of these systems are critical embedded real-time controllers and the respective software programs are developed in C programming language [1].
The work presented herein is developed in the context of obtaining safe controllers for aerospace systems, in collaboration between the Technological Institute of Aeronautics (Brazil) and University of Minho (Portugal).

The absence of specification formalisms, associated to this language, is also a negative point that increases the occurrence of some problems when it is developed the code, namely related with code reutilization or code interpretation.

In the context of this lack some techniques can be used for improving quality of developed software, like, for instance, test and formal verification [2] among others.

Some authors [3] tried, before, to use formalisms from the industrial automation field in order to develop some techniques of translation of these formalisms to C programming language. The main lacks of the mentioned work are that the behaviour of the controller was not considered - and from our point of view, it is not, only, necessary to translate the formalism, but to consider, too, the behaviour of the controller where the code will be implemented - and also extremely important, aspects relied with time specification were not considered too. This last aspect is very important because, on the specification of behaviour of mechatronic systems, the specification of time is always a very serious and important subject.

Although presenting a global approach for formal verification of aerospace systems programmed using C language, this paper addresses special attention at time aspects, starting with time specification, following with time's formal verification, till time programming using C programming language.

In order to achieve the main proposed goal, of this paper, this section was devoted to exposing the actual context of the work; section 2 presents the global approach proposed for formal verification of SFC specification [4] and respective direct translation and formal verification, using formal verification techniques, and sequent implementation using C programming language; section 3 presents a case study, with a SFC specification - which includes specification of time and respective translation to algebraic equations - that will be the basis for formal verification tasks and for programming tasks with C programming language; section 4 presents the model that is formally verified and explains how this model

has been obtained; further, section 5 presents the C code obtained from SFC specification; and, finally, section 6 presents some conclusions about the current work and possible future directions.

2. A GLOBAL APPROACH FOR OBTAINING DEPENDABLE C CODE FROM SFC (IEC 60848) SPECIFICATION

Dependability [5] [6] is the concept that better describes aerospace systems' controllers. More and more these controllers are becoming increasingly complex. For this reason, to assure all the behaviour requirements for these systems is a very hard and complex task. The non-accomplishment of those requirements can lead to catastrophic situations with undesired accidents [7].

One of the most interesting analysis techniques, among others, that lead to very promising results – when developing software for those systems – is Formal Verification [8]. The ideal approach, in this context, would be to be able to apply formal verification techniques, directly, using C programming language code. This approach is, nowadays, possible using the model-checkers BLAST [9] and CMBC [10]. However, using this approaches, there are still many limitations, namely the inability to deal with some functions of C code. Even if it is possible to verify some functions of the C code, there exist other real limitations, like for instance, in reusing parts of C code for similar applications.

Considering the limitations mentioned above, if a system behaviour is modelled by a formalism and then there exists the possibility of translating this formalism to a model-checker, where it can be proved a set of system's behaviour properties, it is possible to be sure that the created specification is correct. After this, if the same formalism – translated to C code with systematic rules – is the base of the C program code, the designer will be assured of the quality of the C program code.

Considering this reasoning, [11] proposed an approach for formal verification of real-time systems, with systematic steps since the specification till the obtaining of the C program code (figure 1).

As specification formalism it was chosen the SFC [4] and as model-checker it was chosen the model-checker UPPAAL [12] since it is a model-checker that allows the designer to deal with time and to verify real-time systems. The choice of the SFC formalism is justified by the fact that there are some works that explain how to convert a SFC specification to Timed-Automata, that are the input formalism of UPPAAL software. [13] [14] [15].

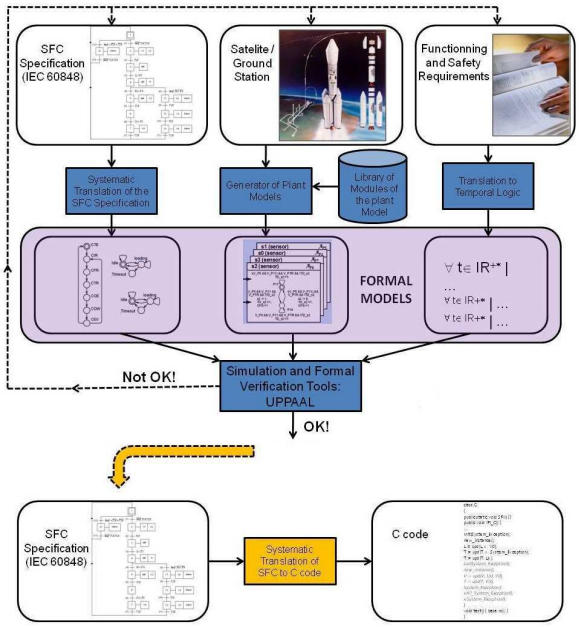

Fig. 1 Proposed approach for formal verification of a SFC specification.

The formal verification of the specification can take into account, or not, plant models, depending of the type of properties to prove [13]. So considering, or not, plant models is also mentioned on the proposed global approach presented in figure 1.

The hard task related with writing behaviour properties of the system is also addressed [16] and the definition of property patterns is an important aspect that were considered on the proposed approach.

The proposed methodology is divided in two main steps and has, as main goal, to obtain safe C program code from a SFC specification.

A very important aspect, when considering this approach is to deal with time specificities, when time is specified on the SFC specification of the controller behaviour.

3. ILLUSTRATION USING A CASE STUDY

Based on the algebraic equations and on the execution algorithm for the execution of the SFC [17], the main idea – as basis of this approach - is to translate the SFC specification to algebraic equations and to use them as basis for formal verification tasks and also as basis for developing the C program code, in a systematic way. If the specification (based on those equations) is verified with desired results, during formal verification tasks, it can be assumed that the program, in C programming language, is correct because the basis for formal verification tasks and programming tasks were exactly the same: the algebraic equations obtained, in a systematic way, from SFC specification.

Let's consider the SFC specification presented in figure 2.

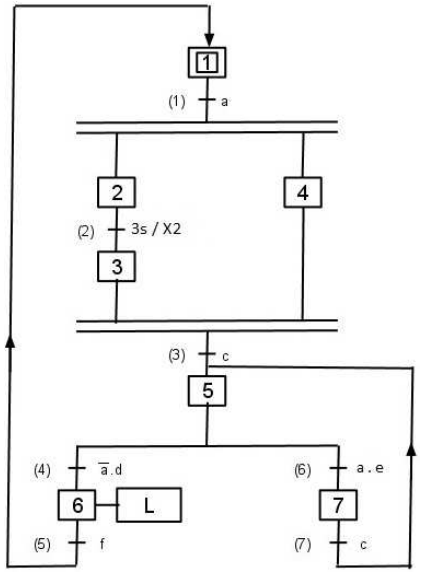

Fig. 2 SFC specification, corresponding to an illustrative case study.

This SFC specification, although being a simple specification has some particularities that deserve to be studied, such as: “and”, “or”, “sequence selection” and “specification of time”, that can be stated in the figure.

Considering some rules for obtaining algebraic equations from an SFC specification [17] the corresponding algebraic equations of the SFC specification presented, in figure 2, are:

Clearing conditions:
CC(1) := X1 . a
CC(2) := X2 . 3sX12
...
CC(7) := X7 . c

Step variables:
$X1_{(t+1)}$:= CC5 + $X1_{(t)}$. /CC(1)
...
$X5_{(t+1)}$:= CC3 + CC7 +$X5_{(t)}$. /(CC(4) + CC(6))
...
$X7_{(t+1)}$:= CC6 + $X7_{(t)}$. /CC(7)

Outputs:
L := X6

Concerning algebraic equations mentioned above, all variables are Boolean variables and, in the notations: “:=” means “takes the logical value of”; “.” is the logical *and*; “+” is the logical *or*; “/” is the logical *not*; and “3sX12” is a logical variable that will take the logical value “1” three (3) seconds after activating the step 2 of the specification.

4. CREATION OF THE MODEL FOR FORMAL VERIFICTION TASKS

As indicated on the approach presented on the figure 1, the formal verification tasks can be performed considering, or not, plant models of the analyzed system. As, in this paper, the main focus is related with dealing of time specification, no physical system was associated at the SFC specification of figure 2.

However, concerning formal verification non-model-based [18] (without considering plant models) we have considered three main models: a model for the controller behaviour (named as *CONTROLLER_BEHAVIOUR*), a model concerning the controller program (*CONTROLLER_PROGRAM*), and a model for modelling the time (*TIMER*). Also, all the input variables of the SFC specification were modelled as random variables using, for each one, a model composed by a location and two transitions: on one of those transitions it is assigned the logical value “1” to the variable and on the other transition it is assigned the logical value “0” to the variable. This model was instantiated for all the input variables of the SFC specification.

In this section the model of the program and the model of the time are presented, discussed and, also, the synchronization between them is illustrated.

The model of the timer can be instantiated as many times it is needed.

The model of the controller program is illustrated in figure 3.

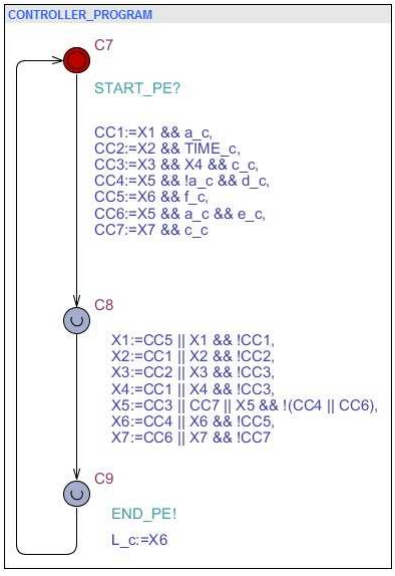

Fig. 3 Model of the program, verified with UPPAAL, corresponding to SFC specification of the case study.

This model is composed by three locations (or states) and during transitions between states, there is assigned the logical value to each Boolean variable

associated to the transition of the model. On the transition from location C7 to C8 it is also previewed a synchronization message that is responsible for connecting this model with the model of the controller behaviour. So, when the model of the controller behaviour sends the message "START_PE" (meaning "*start program evolution*"), the evolution of this model starts immediately. In the first transition, all the variables concerning clear conditions are actualized and the evolution of the model drives to the location C8. From location C8, the model evolves immediately – because location C8 is an *Urgent* location - to C9 location, actualizing the values of all the step variables associated to this transition. From this location (C9) the model evolves immediately to C7 location, sending a message - to the model of the controller behaviour - that the evolution of the model of the program has been finished ("END_PE"; "*end of program evolution*"). This last evolution actualizes the values of the outputs corresponding to the SFC specification. The model stands in this location (C7) till the moment that it will receive again the message "START_PE".

In order to obtain more interesting results, on the formal verification tasks, the variables used on the calculation of the clearing conditions (see transition from C7 to C8, in figure 3) are copies from the real inputs of the model. As we can understand, when the controller is running, it is not sensible to the changing of the inputs. Taking this very important fact into account, on the model of the controller behaviour, there are made – during inputs reading step - the copy of all inputs with the following nomenclature:

a_c is the copy of variable *a; b_c* is the copy of variable *b; c_c* is the copy of variable *c; TIME_c* is the copy of variable *TIME;* and so on...

Concerning the formal verification of time specification, it was defined a model to deal with this specific aspect, presented in figure 4.

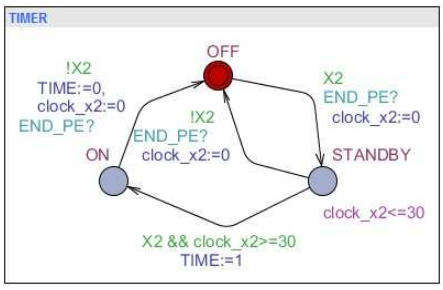

Fig. 4 Model of the time specification, verified with UPPAAL, corresponding to time of SFC specification of the case study.

In this model three locations are considered. The location *OFF* (initial location) means that the timer is *off* and that the variable that starts the timing process is *off* too (in this case, the step variable X2).

If the step variable changes its logical state, from *off* to *on*, then the model evolves from location *OFF* to location *STANDBY* and, in this location, starts the process of timing, associated to the clock of the model, named as *clock_x2*.

The time unit considered was 0.1 seconds, so the clock must finish time on 30 * 0.1 seconds = 3 seconds (time specified on the SFC specification). When this time is elapsed, the transition from location *STANDBY* to location *ON* is fired and logical value "1" is assigned to variable *TIME*. At any moment of this process, if the step variable, that starts the process (in this case X2), changes from logical value "1" to logical value "0", the model evolves immediately to location *OFF* and the model remains on this location till next changing of logical value, of the step variable, from "0" to "1". In this context, logical value "0" is always assigned to variable *TIME*.

Each time that model of the program finishes its evolution, the model of the timer is actualized by the synchronization message "END_PE". This message forces the evolution of this model, in order to be assumed, during all *scan* cycles of the controller, that the model is actualized and takes always the more recent value of the step variable *X2*.

When *TIME* variable changes from logical value "0" to "1" – on the transition from location *STANDBY* to location *ON*, on the model of the timer (figure 4) – its value will be taken into account on the next evolution of the model of the program, during transition from location C7 to location C8 (figure 3).

With this configuration, behaviour properties (of any system) dealing with time can be verified. The time associated at the *scan* cycle of the controller is modelled on the model of the controller. Also, some aspects of the controller, like being a monotask or multitask controller, among others, can be considered in this model: *CONTROLLER_BAHAVIOUR*.

5. C CODE PROGRAM OBTAINED FROM ALGEBRAIC EQUATIONS OF THE SPECIFCATION

It is important to highlight that the same algebraic equations obtained from SFC specification and verified with UPPAAL model-checker, are the same that will be used as basis for obtaining the respective C program code. As illustrated in figure 1, the SFC specification can be re-designed if the obtaining results are not satisfying, during the formal verification tasks.

Based on the algebraic equations and on the execution algorithm for the execution of the SFC presented above (figure 2), the C program code can be developed in systematic way.

The task of writing C code can, now, be done in a simple, rigorous and systematic way. It is possible, now, to give a *top-down* structure for C code and it is

possible, too, to introduce the concept *task*, from a hierarchical point of view, on the C code.

The C programming language is a very complete language and allows, through the use of simple and few functions (*for*, *while*, *if*, *vectors*, *matrices*, *pointers* and a few more), to create large and complex programs.

The language is based on the concept of creating functions that execute some specific tasks that can be reused on several programs.

When developing programs, the user needs to create some functions that can be interesting and useful for him. There exist also, some libraries that can be inserted on a program and they carry out hundreds of (previously defined) functions that can be reused any time as necessary, on the current program, or others.

Concerning the proposed approach and considering, also, the execution algorithm of a SFC, the program is developed taking into account that the code will be implemented on a monocyclic and mono-task real-time controller.

The execution algorithm of a SFC is comprised of several steps: *inputs reading* (reads the input variables that model sensor signals), *program execution* (calculates and actualize the values of the internal controller variables), *outputs updating* (actualize the values that model the orders that the controller sends to the plant) and *timers* (dealing with time aspects). The elaboration of the C program has, as structure, precisely the structure mentioned above.

One important characteristic of C programming language is that this language allows reusing some libraries where are allocated several functions. This fact allows us to use those libraries. The libraries that are used, on our approach are: *stdlib.h*, *conio.h e time.h*.

On the library *stdlib.h* there are found some standard functions of C programming language, almost always needed when developing a C program. The library *time.h* is useful, too, because it deals with aspects related with time, being the time measured in milliseconds.

Our program declares, first, the mentioned libraries (figure 5) and then there are declared the variables int x[N_STATES], xold[N_STATES], ct[N_TRANS] and in[N_INPUTS].

In order to be possible to use the same approach, in a systematic way, with other SFC specifications the number of inputs is an integer number that can be easily changed concerning other application of the same kind. For this, it is only necessary to change the size of the arrays.

```
#include <stdlib.h>
#include <conio.h>
#include <time.h>

#define N_INPUTS 50
#define N_STATES 20
#define N_TRANS 20
#define N_TIMERS 20

int x[N_STATES],xold[N_STATES],ct[N_TRANS], in[N_INPUTS];
```

Fig. 5 Initialization

As mentioned before, the code is divided into four main parts:

- Input (reading) of data (variables that model the sensors behaviour);
- Calculation of the Clearing Conditions and the calculation of the Step variables of the SFC (that model the internal variables of the controller);
- Updating of outputs, based on changing of the step variables of the SFC (variables that model the orders sent from the controller to the plant).
- Dealing with time aspects

The inputs reading task is represented by the following *for* cycle:

```
void readinputs() // INPUTS -------------------
{int i;
  for (i=0;i<N_INPUTS;i++) in[i]=0;
 /* to define according the used controller*/}
```

Fig. 6 Reading Inputs

This cycle is incremented and all the inputs are actualized. It is important to focus that, in figure 8, appears a comment "to define according the controller" that means that this part of the code is specific from each controller device. The physical inputs address must be indicated in order to allow the reading of inputs. For instance, *in[1]*, *in[2]*, *in[3]*, *in[4]*, *in[5]* and *in[6]* would represent, respectively the variables *a*, *b*, *c*, *d*, *e* and *f*.

Figure 7 presents the most important part of the code. In our methodology, all the other blocks can de reused, and this one too. This one is the only one that demands the new elaboration of the equations (corresponding to clearing conditions and step variables of the SFC) because these equations depend directly of each specific SFC specification. The function *readinputs* is presented in figure 6 and the function *updateoutputs* is presented in figure 8.

```
int main(){
 int i;
  x[1]=1;//activate initial state
  while (1){
    readinputs(); //read inputs
    ct[1]=x[1] && in[1];//compute cts
    //...
    ct[7]=x[7] && in[3];

    for(i=0;i<N_STATES;i++) xold[i]=x[i];//compute xt+1
    x[1]=ct[5] || x[1] && !ct[1];
    //...
    x[7]=ct[6] || x[7] && !ct[7];
    updateoutputs();//update outputs
    }
}
```

Fig. 7 *Main* function. Translation of SFC clearing conditions and SFC step variables

In C the *main()* function is used as the starting point of the program. Inside the *main* function the order in which the equations are written is followed. According to figure 9, the initialization is done with *int* declarations. The variables are declared and the first step variable of the SFC (x[1]) is activated (logical value 1). Further, a *while* cycle is created, and the program is always running according the execution cycle of the controller. The functions are executed by the following order: *readinputs*, program execution (calculation of the *clearing conditions* and then the *controller state variables*) and, finally, the *updateoutputs* function (figure 9).

The *xold[i]* variable means the logical value of the variable in the previous cycle and the *x[i]* variable means the current value of the state variable.

```
void updateoutputs() //OUTPUTS-----------------
{int i;
  for(i=0;i<N_STATES;i++){
  if (xold[i]!=x[i]) x[i]=i;
/* to define according the used controller*/ }
}
```

Fig. 8 Update outputs

Finally, the program updates the values of the output variables (figure 8) with *for* and *if* cycles. The value of the output is changed if *x[i]* is not equal to *xold[i]*: x → xold[i]!=x[i]. As with the reasoning presented for the input variables, the indexation of the outputs to physical addresses of the controller devices depends directly of the used controller device, so the comment: "to define according the used controller", in figure 8.

Till now, the programming of timers was not yet detailed and illustrated. The implementation of timers will require the definition of a structure for each timer, in order to systematize this programming task. As detailed before, the variable that will start time "counting" is the step variable X2.

In order to facilitate the task of timers programming, it will be created an array of timer, when we can easily define the size of the array from one application to another of the same kind. We must remember that our idea is to make this approach as an systematic approach in order to be easily applied in another applications of the same kind. By definition, for each SFC, is defined an array for respective timers with size equal to the number of steps of this SFC specification. Of course that, neither all the steps (with their respective step variables) will be used as basis for the respective timer, but this fact makes our approach more systematic and easy to use.

For instance, concerning a SFC of 25 steps, it would be created, automatically, an array with 25 timers: one for each step variable of the SFC.

```
// TIMERS ---------------------------------------------------------------

typedef struct timer{
int interval;
clock_t start;
} ttimer;

ttimer timers[N_STATES];

void newtimer(int x, int interval) {
timers[x].interval=interval;
}

int timeron(int x) {
return timers[x].start!=0 && (curtime-timers[x].start)>=timers[x].interval;
}

void updatetimers() {
clock_t t;int i;

t=clock();
for (i=0;i<N_STATES;i++){
if (timers[i].interval!=0){
    if (x[i]==0) timers[i].start=0;
        if (xold[i]==0 && x[i]==1) timers[i].start=t;
    }
   }
  }
```

Fig. 9 C Programming of timers, associated to the SFC specification of the case study.

Figure 9 illustrates the C programming of timers, following our proposed approach. Initially, it is created a structure for each timer, it is realized the respective indexation and, finally, the respective actualization of state of the timers. The creation of the timer, indicating the step variable of the SFC at which it is associated, is done by the function *newtimer*.

The function *timeron* is executed in each scan cycle and will indicate if the timer is on state *on* or not. It initiates the "*time counting*" if it detected the state changing of the respective step variable and will deactivate the timer when the step variable, that originates the timer, returns to *off*. Finally, the timers are updated, in each scan cycle, by the function *updatetimers*.

Not only timers can be treated this way, following our approach, but also counters can be programmed following the same reasoning. The programming of counters is similar to programming of timers. It must be created a structure for each counter that would contain the value to count, the current value of the counter and the *reset* of the counter. Of course that this similarity is true if the variable, that will originate the counting function, is a step variable of the SFC specification.

6. CONCLUSIONS AND FUTURE WORK

This paper has shown how to obtain a simple and systematic translation of a SFC specification to a C program code. The main focus of this paper was to present a detailed discussion about some aspects of time specification like modelling, formal verification and obtaining the correspondent C program from the initial SFC specification of time.

Also, some very important aspects like taking into account the cyclic behaviour of the controller device were considered. The goals of the paper were totally accomplished.

With the proposed approach, the reusing of small parts of the code – for similar applications - is simple because the graphical use of the specification formalism allows changes and different organizations in a simple and commode way.

The work presented in this paper is inserted in a complex project development and, in parallel with the systematic obtaining of C programs code - in a near future - the authors intended to define some rules in order to facilitate the elaboration of the C program code facilitating the tasks of its formal verification (using model-checkers for direct model-checking of C code) trying to eliminate some gaps existing, nowadays, in this field.

7. REFERENCES

[1] Ritchie, D. (1993). The Development of the C Language. Second History of Programming Languages conference, Cambridge.

[2] A Clarke, E., Kroening, D., and Lerda, F., (2004). A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, TACAS 2004, Vol. 2988 of Lecture Notes in Computer Science, pp 168–176, Springer.

[3] Bayó-Puxan, O., Rafecas-Sabaté, J., Gomis-Bellmunt, O., and Bergas-Jané, J. (2008). A GRAFCET- compiler methodology for C-programmed microcontrollers, Assembly Automation Emerald Group Publishing Limited, pp. 55–60.

[4] EN 2002 (2002) - European Standard 60848: GRAFCET specification language for sequential function charts.

[5] Roussel, J.M., and Giua, A. (2005). Designing dependable logic controllers using the supervisory control theory. in CDROM Preprints 16th IFAC World Congress, Praha, Czech Republic, paper n° 04427, 6 pages.

[6] Johnson, T.L. (2004). Improving automation software dependability: A role for formal methods? *Special Issue on Manufacturing Plant Control: Challenges and Issues. 11th IFAC INCOM'04*. Symposium on Information Control Problems in Manufacturing, Vol. 15, Issue 11, pp. 1403-1415.

[7] Leveson, N. (2005). Role of Software in Spacecraft Accidents. *Journal of Spacecrafts and Rockets*, vol. 41, no. 4, pp. 564-575.

[8] Nadjm-Tehrani, S., and Strömberg, J. (1999) Formal Verification of Dynamic Properties in an Aerospace Application. *Formal Methods in System Design archive*, Vol. 14, Issue 2, ISSN:0925-9856, pp. 135 – 169.

[9] Hezinger, R., Jhala, R., Majumbar, R. and Sutre, G. (2003) Software verification with Blast.

[10] Clarke, E., Kroening, D., and Lerda, F., (2004). A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, TACAS 2004, Vol. 2988 of Lecture Notes in Computer Science, pp 168–176, Springer.

[11] Borges, P., Villani, E., Machado, J., Ferreira, J., and Campos, J. (2010). Abordagem Sistemática para o Controlo Seguro de Sistemas aeroespaciais. XIV International Congress on Project Engineering, Spain.

[12] Bengtsson, J., and Larsson, F. (1996). Uppaal a Tool for Automatic Verification of Real-Time Systems. Docs Technical Report Nr 96/97, Uppsala University, ISSN 0283-0574.

[13] Machado J. (2006). Influence de la prise en compte d'un modèle de processus en vérification formelle des Systèmes à Evénements Discrets. PhD Thesis, École Normale Supérieure de Cachan, France.

[14] Remelhe, M.P., Lohmann, S., Stursberg, O., and Engell, S. (2004). Algorithmic Verification of Logic Controllers given as Sequential Function Charts. IEEE International Symposium on Computer Aided Control Systems Design Taipei, Taiwan

[15] Stursberg, O., Lohmann, S., and Engell, S. (2005). Improving dependability of logic controllers by algorithmic verification World Congress IFAC, Vol. 16, Part 1, Czech Republic.

[16] Campos, J., and Machado, J. (2009). Pattern-based Analysis of Automated Production Systems. 13 th IFAC Symposium on Information Control Problems in Manufacturing, In Proceedings of the 13 th IFAC Symposium on Information Control Problems in Manufacturing, Moscow.

[17] Machado J., Seabra E., Campos J., Soares F., Leão C, (2011). Safe controllers design for industrial automation systems. Computers & Industrial Engineering 60 (2011) 635–653

[18] Frey, G. and L. Litz (2000). Formal Methods in PLC programming. In: 2000 IEEE International Conference on Systems, Man & Cybernetics. pp. 2431-2436, Nashville.