Object-Tokens in High-Level Grafchart

Johnsson, Charlotta; Årzén, Karl-Erik

1996

OBJECT TOKENS IN HIGH-LEVEL GRAFCHART

Charlotta Johnsson and Karl-Erik Årzén

Department of Automatic Control,
Lund Institute of Technology,
Box 118, S-221 00 Lund, Sweden,
lotta@control.lth.se, karlerik@control.lth.se

Abstract: The paper presents an high-level extension to Grafchart, a Grafcet-based toolbox for supervisory control applications. The extension allows tokens to be objects with attributes. Contrary to most high-level extensions to Petri Nets and Grafcet it is not based on arc inscriptions.

Keywords: Grafcet, Petri nets, High-Level Petri Nets, Supervisory Control

1. Introduction

Grafchart is the name of a toolbox for sequential supervisory control applications that has been developed at the Department of Automatic Control since 1991 [Årzén, 1991], [Årzén, 1994b]. Grafchart is implemented in G2 [Moore *et al.*, 1990], an object-oriented graphical programming environment developed for supervisory control applications. Grafchart is based on the graphical syntax of Grafcet [David, 1995]. Grafcet originally developed in France during the 1970s as a formal specification method for logical controllers. It has since become the basis for Sequential Function Chart (SFC) language through the international standards (IEC 848) and (IEC 1131-3). In SFC the emphasis has gradually shifted towards providing a graphical programming language for sequential control problems. Grafchart continues this development. It is aimed at supervisory control applications of a sequential nature. This can either mean that the process is of a sequential nature, e.g., a process with different operating modes, or that the application itself can be decomposed into sequential steps.

Grafchart is used industrially in an oil refinery application [Årzén, 1994a]. The system uses expert system techniques coupled with numerical optimization in a decision-support system that gives on-line advice regarding the distribution of hydrogen resources in the refinery. In [López González *et al.*, 1994], Grafchart is used to implement a flexible manufacturing cell. Here, Grafchart is used in a four-layered hierarchical structure to represent the plant-wide operating phases of the control system, to describe the sequences of tasks to be executed to manufacture the parts, to describe the tasks at the workstation level, and, finally, to describe the different services offered by the device drivers in the cell. Grafchart has also been used to implement a prototype of a training simulator for a sugar crystallization process [Nilsson, 1991].

High-Level (H-L) Grafchart is an extension to Grafchart that is currently under implementation. H-L Grafchart combines the graphical language of Grafcet/SFC with object-oriented programming language constructs and ideas from High-Level Petri Nets. The features that are new in H-L Grafchart, compared with Grafchart, are parameterization, methods and message passing, object tokens, and multi-dimensional charts. The first two of these additions are presented in [Årzén, 1996b]. The topic of this paper is the third addition, object tokens. Here the tokens of the function chart are objects that carry information similar to tokens in High-Level Petri Nets. An application of H-L Grafchart for recipe-based batch control is presented in [Johnsson and Årzén, 1994] and in [Johnsson and Årzén, 1996]. Alarm filtering based on H-L Grafchart is presented in [Årzén, 1996a].

Grafchart and the parameterization and message passing features of H-L Grafchart are presented in Section 2. The object token extension of H-L Grafchart is presented in Section 3. A difference from High-Level Petri Nets [Jensen and Rozenberg, 1991] and previous work on Coloured Grafcet [Agaoua, 1987], [Suau, 1989] is that arc inscriptions are not used.

2. High-Level Grafchart

Grafchart is based on the graphical syntax of Grafcet summarized in Fig. 1. A step represent a state, phase or mode that can be active or inactive. Associated with the step are actions that are performed when the step is active. Steps are connected by transitions. Each transition has an associated receptivity that could be a logical condition, an event or an combination of an event and a logical condition. When the receptivity of a transition becomes true the transition fires i.e. deactivates the steps preceding it and activates the steps succeeding it. Grafchart supports alternative

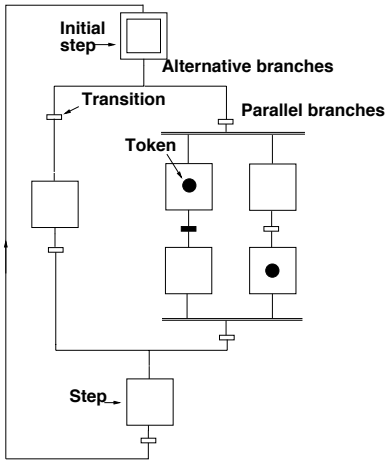

Figure 1. Grafcet example

branches (distribution OR) and parallel branches (distribution AND) in the same way as Grafcet.

Function charts

All the basic building blocks in Grafchart are defined as G2 objects and the arcs are implemented as G2 connections. A Grafchart function chart can either be closed or terminated by a sink transition. Optionally, a function chart can be encapsulated by a *Grafchart process* object. The user can define subclasses to all Grafchart elements, e.g., Grafchart processes or steps, and in these subclasses add attributes. In this way the elements can be specialized. The attributes of an element act as parameters whose values can be referenced and changed from step actions and referenced from transitions of the function chart.

Steps

The main difference between Grafcet and Grafchart concerns the way step actions are represented. Since Grafcet has been developed as a logical controller the actions that can be done in a step are of a boolean nature. Different types of actions are allowed, e.g. normal (level) actions, stored (impulse) actions, time-delayed actions and time-limited actions.

The actions that can be associated with a step in Grafchart are more general; they can be compared with the statements of a conventional programming language. A Grafchart action can be of four basic types: *always, initially, finally* and *abortive*. All actions can be conditional or unconditional.Always actions are executed periodically while the step is active. Initially actions are executed once when the step becomes active. Similarly, finally actions are executed once immediately before the step becomes deactivated and abortive actions are executed once immediately before the step is aborted. The actions are represented by *action templates*. These are text strings that during compilation are translated into the corresponding G2 rules. The actions that may be performed in a step action are the action types provided by G2,

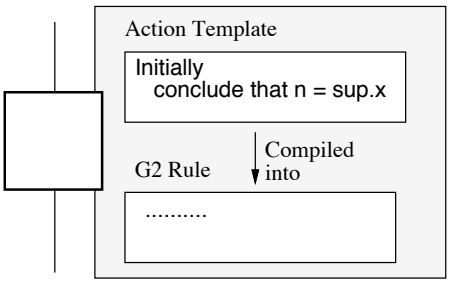

Figure 2. Grafcet example

e.g., assignments, procedure invocations, object creation and deletion, animation actions, etc. The action templates can reference parameters with the dot notation `sup.attribute`. During compilation this is translated into the appropriate G2 expression, i.e. a reference to an attribute. Lexical scoping is used. The situation is shown in Fig. 2

Transitions

Each transition contains two attributes: `event` and `condition`. Here, the user enters the event expression and/or the logical condition telling when the transition should fire. The `sup.attribute` notation may be used. During compilation this is translated into the appropriate G2 rule (compare step actions). A transition that contains an event expression will give rise to a so called *whenever rule* that is fired asynchronously when the event occurs. A transition with only a logical condition give rise to a scanned rule with the shortest scan interval possible.

Execution Model

The formally correct way of executing Grafcet is defined by an interpretation algorithm [David and Alla, 1992]. The algorithm is based on the hypotheses that external events never occur simultaneously and that the logical controller is always faster than the process it controls. When Grafcet is implemented in a Programmable Logic Controller (PLC) the step-transition structure and the step actions are translated into PLC language, e.g. ladder logic or instruction lists. In this case the previous hypotheses may no longer hold.

The execution model of Grafchart is tightly coupled to G2's execution model. Steps and transitions are defined as G2 objects that have *activatable subworkspaces*. A workspace is visualized as a virtual, rectangular window upon which various G2 items such as rules, procedures, objects, displays, and interaction buttons can be placed. A workspace can also be attached to an object. In this case the workspace is called a subworkspace of that object. When a subworkspace is deactivated, all the items on the workspace are inactive and "invisible" to the G2 execution system. This means, e.g., that rules placed on a deactivated subworkspace cannot be invoked. The step actions of a step are internally represented as rules that are placed

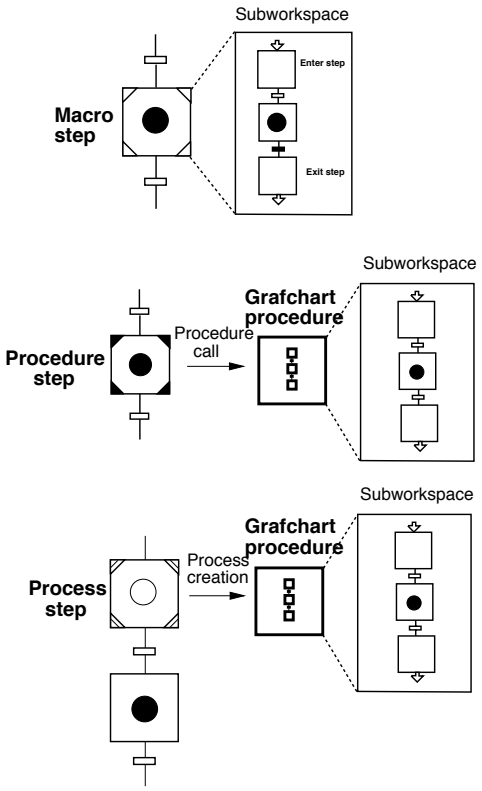

Figure 3. Macro step, procedure step and process step.

on the subworkspace of the step. These rules may only be invoked when the associated step is active and hence the subworkspace is activated. The receptivity of a transition is also internally represented as a rule that is placed on the subworkspace of the transition. This subworkspace in only active when the transition is enabled. When the transition fires, i.e., the rule condition becomes true, the rule starts a procedure that takes care of the activation and deactivation of steps and transitions.

Hierarchical abstractions

Macro steps are used to represent steps that have an internal structure of (sub)steps, transitions and macro steps. The internal structure is placed on the subworkspace of the macro step, see Fig. 3 (top). Special enter-step and exit-step objects are used to indicate the first and the last substep of a macro step. When the transition preceding a macro step becomes true, the enter-step upon the subworkspace of the macro step and all the transitions following that enter-step are activated. The transitions following a macro step will not become active until the execution of the macro step has reached an exit-step. A macro step may also contain actions. The initial action of a macro step are executed before the initial steps of the enter step of the macro step. The finally actions of a macro step are executed after the finally actions of an exit step of the macro step. Always actions are executed all the time the macro step is active independently of which substep that is active. Abortive actions on the macro level are executed when the macro step is aborted.

Sequences that are executed in more than one place in a function chart can be represented as *Grafchart procedures*, see Fig. 3 (mid). The procedure body is stored on the subworkspace of the procedure. The Grafchart procedure may have parameters that are visible in the procedure body. The call to a procedure is represented by a *procedure step*. The procedure step contains a `procedure` attribute that contains the name of the procedure that should be called and a `parameters` attribute which is used to set the values of the parameters in the procedure. It is possible both to pass in parameter values to the procedure and to pass out parameter values from the procedure.

A Grafchart procedure can also be a method of a general G2 object. For example, a G2 object representing a chemical batch reactor can have Grafchart methods for charging, discharging, agitating, heating, etc. Inside the method body, it is possible to reference the object itself and the attributes of this object using the `self` and `self.attribute` notation. A method is called from a procedure step by changing the value of the `procedure` attribute of the procedure step to the form `<object> <method>` where `<object>` is a reference to the object and `<method>` is a reference to the method name.

The Grafchart procedures are reentrant. Each procedure invocation executes in its own local copy of the procedure body. This makes recursive procedure calls possible.

A procedure step is the equivalent of a procedure call in an ordinary programming language. Sometimes it is useful to start a procedure as a separate execution thread, i.e., to start the procedure as a separate process. This is possible with the *process step*, see Fig. 3 (bottom). The transitions after a process step become firable as soon as the execution has started in the Grafchart procedure. An outlined circle token is shown in the process step as long as the process is executing.

Exception Transitions

Grafchart supports *exception transitions*, a special type of transition that may only be connected to macro steps and procedure steps. This transition is enabled all the time while the macro step is active. If the exception transition becomes firable while the corresponding macro step is executing the execution will be aborted, abortive actions, if any, are executed, and the step following the exception transition will become active. Macro steps and procedure steps "remember" their execution state from the time they were aborted and it is possible to restart them from that state. Grafchart also supports various types of macro actions.

Firing Rules

Grafcet has three firing rules:

1. All firable transitions are immediately fired.
2. Several simultaneously firable transitions are simultaneously fired.
3. When a step must be simultaneously activated and deactivated, it remains active.

The second one is the one that distinguishes Grafcet from Petri Nets the most. In an distribution OR situation, in Grafcet, the transitions in the both branches can simultaneously be firable. In this case both the transitions will fire and both the "alternative" branches will be executed. This is often not the intention of the designer and it is recommended that these transitions are made mutually exclusive (in SFC it is required). Grafchart treats this situation by non-deterministically chosing one of the transitions. Concerning Rule 3, Grafchart executes the initially and finally actions for all steps also if they are part of an unstable situation.

3. Object Tokens

In Grafcet a token is simply an boolean indicator that tells if the step is active or not. In the object token extension to H-L Grafchart that is currently implemented, a token is an object that carries information, i.e. has attributes. A step may contain multiple tokens that are of the same or of different class. It is possible to reference and change the attributes of a token object from step actions and and reference attributes of a token object that enables a transition from the receptivity of the transition.

Object tokens are inspired by Coloured or High-Level Petri Nets [Jensen and Rozenberg, 1991] in which the tokens are abstract data types or objects. The main advantage of High-Level Petri Nets is that they allow a compact modeling of large systems that consist of several similar substructures.

Arc inscriptions

Most of the existing work on High-Level Petri Nets and Coloured Grafcet, [Agaoua, 1987], [Suau, 1989], is based on arc inscriptions. Arc inscriptions can be of two main types. In the *function representation*, [David and Alla, 1992] and [Jensen, 1981], the arc inscriptions are *functions* that are associated with the input arcs and output arcs of a transition. Associated with each transition is a set of firing colours. Each of these indicates a firing possibility of the transition. The functions on the input arcs determine the numbers and the colours of the tokens that will be removed from the input places when the transition fires with respect to a certain firing colour. Similarly, the functions on the output arcs determine how many and which colour types that will be added to the output places of the transition. Hence, firing a transition conceptually corresponds to deleting tokens in the input places and creating tokens in the output places. The *expression representation* instead uses arc expressions in combination with transition guards [Jensen, 1990]. It has been shown that the two approaches are equivalent and can be transformed into each other.

A major reason why arc inscriptions are necessary in H-L Petri Nets is that each transition may have multiple input places and multiple output places. The arc inscriptions are needed to specify how the different input places should contribute to the enabling of the transition and how the different output places should be affected when the transition is fired. If each transition only had one input place and one output place arc inscriptions would be unnecessary. Then the same information could be encoded in the transitions. If we consider Grafcet without parallelism (Distribution AND and Junction AND), each transition has only one input step and one output step. The firing of a transition can now be seen as moving a token from the input step to the output step. This is the approach taken in H-L Grafchart. Each transition is viewed as if it only has one input step and one output step. If the transition is followed by a distribution AND then all the output steps of the transition are treated as a single step by the transition. Similarly if the transition is preceded by a junction AND then all the input steps of the transition are treated as a single step by the transition. Due to this it is not necessary to use arc inscriptions in High-Level Grafchart.

Even though arc inscriptions are not allowed in High-Level Grafchart this does not restrict the different ways of firing a transition. By instead allowing the receptivities of the transition to be written in several different ways the same performance, as the one achieved using arc inscriptions in H-L Petri Nets, can be achieved. Colortransformations, attribute changes and deletion and creation of tokens are all possible to do. Using arc inscriptions one easily looses the clarity of the net. Since one of the main advantages of Grafcet is its clear and intuitively understandable way of representing sequences it is undesirable to use arc inscriptions in H-L Grafchart.

Transitions and Receptivities

Each transition has one receptivity for each token class that it can be enabled by, see Fig. 4. The receptivity of token class `P` of transition `t` becomes enabled as soon as one instance of class `P` arrives in the input place of `t`. The condition of the receptivity may refer to the attributes of the token class instance that enables the receptivity. It may also refer to the presence of other tokens in the input step. When the receptivity becomes firable an operation is performed on the tokens that are referenced by the receptivity. In the standard case the operation would be that a token is moved from the input step to the output step. In sink and source transitions the operation will be to

delete and create object tokens and to initialize their attributes. The operations can also be more complex e.g. an attribute of the token can be changed at the same time as the token is move from the input step to the output step, an other token placed in the input step can be deleted or a new token can be created and placed in the output step. It is also possible to move, not only the token that enables the receptivity but also one or several of the tokens referred to in the condition part of the receptivity.

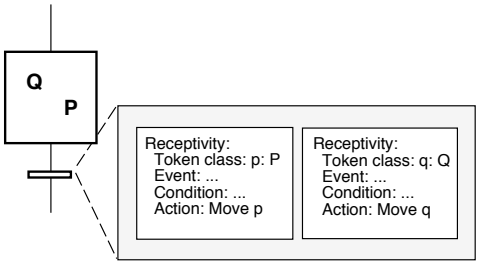

Figure 4. Transition with multiple receptivities

Parallelism

Parallelism, using object tokens, is handled in a similar way as parallelism using ordinary tokens. When the transition preceding an Distribution AND is fired the moved token is "duplicated" and added to the first step in **all** the parallel branches. It is, however, still the same token object in all the parallel branches. The reason for this is that the actual token is only a pointer to the token object that it represents. After the firing the tokens in all the parallel branches will point to the same object token according to Figure 5. In this way, if a token object attribute is changed in one of the branches this will directly be visible for all tokens in all branches.

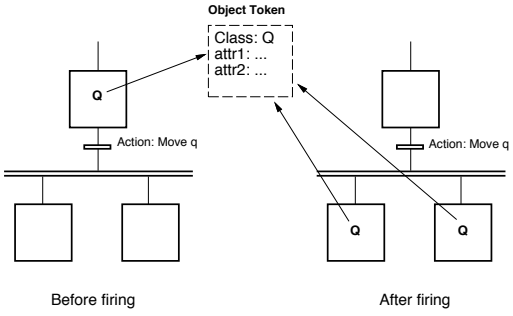

Figure 5. Distribution AND

The transition after a synchronization (junction AND) can only become enabled with respect to a token class `C` if all the input steps of the transition contains tokens that points at **the same** token object of class `C`.

A problem with the described approach is that it does not allow a distribution AND case that involves different token objects and where the different tokens follow different branches. The corresponding Petri Net situation is shown in Fig. 6 (left).

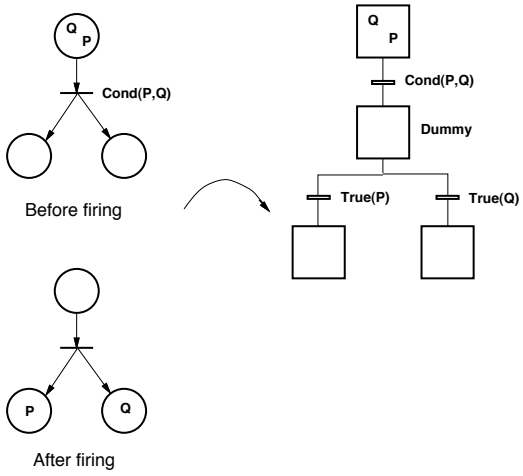

Figure 6. Distribution AND as Distribution OR

After firing, the `P` token should follow the left path and the `Q` token should follow the right path.

In H-L Grafchart this is solved by translating the distribution AND case into an distribution OR case and introducing an empty dummy step according to Figure 6 (right). The first transition contains a receptivity with a condition denoted `Cond(P,Q)`, that involves both object tokens of class `P` and object tokens of class `Q`. The receptivity could, e.g., be

```
Receptivity:
 Token class: p: P
 Event:
 Condition: ∃ q: Q | p.x = q.y
 Action: move p , move q
```

Alternatively the token class could be `Q` with a condition testing if a token of class `P` is present and if the `y` attribute of `Q` equals the attribute `x` of `P`. The transitions after the dummy step are immediately firable with respect to `P` (the left transition) and with respect to `Q` (the right transition). In the same way a junction AND involving multiple token object classes is translated into a junction OR plus a dummy step.

Step actions

To each step actions can be associated. The action types are the same as in Grafchart. The difference is that in High-Level Grafchart each action is associated with a token class, see Fig. 7. An initially (finally) action is executed when an instance of its token class enters (leaves) the step. An always action is executed when an instance of its token class is present in the step. The action may contain conditions that depend on the presence of tokens of other classes and on the values of their attributes.

Initial Marking

With more than one token class it is necessary to be able to have multiple initial steps and to be able to specify which object tokens that they should initially

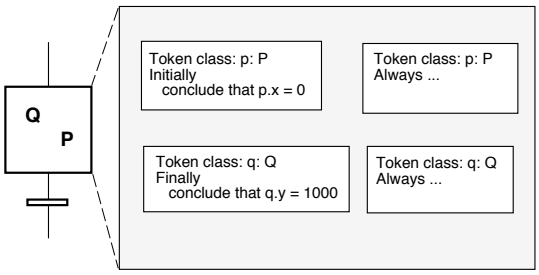

Figure 7. Step actions

contain and what the initial values of their attributes should be.

Procedure steps

A procedure step is a step from which a procedure or a method is called. In Grafchart a procedure step has associated with it, a certain procedure or method. In High-Level Grafchart the procedure step can be used in two different ways. The first is to let the token contain information about the procedure that should be called. Tokens of different classes may then cause calls to different procedures. The second way is to restrict the token to only contain the attributes sent to the procedure and not the procedure itself. Which procedure that should be called is in the latter case determined by the procedure step alone. Which way that will be implemented is still an open question.

4. Conclusions

In this article we have shown how object tokens, i.e tokens containing information in terms of attributes, can be used in High-Level Grafchart. An approach without arc inscriptions has been described. Instead the transition expressions (the receptivities) may be more complicated. The object token extension to H-L Grafchart increases the reusability and can be of great use in many applications.

This work was supported by the TFR project "Integrated Control and Diagnosis", TFR-92-956 and by the NUTEK REGINA project "High-Level Grafcet for supervisory sequential control".

5. References

Agaoua, S. (1987): *Spécification et commande des systèmes à événements discrets, le grafcet coloré.* PhD thesis, Grenoble University (INPG).

Årzén, K.-E. (1991): "Sequential function charts for knowledge-based, real-time applications." In *Proc. Third IFAC Workshop on AI in Real-Time Control*, Rohnert Park, California.

Årzén, K.-E. (1994a): "Grafcet for intelligent supervisory control applications." *Automatica*, **30:10**.

Årzén, K.-E. (1994b): "Parameterized high-level Grafcet for structuring real-time KBS applications." In *Preprints of the 2nd IFAC Workshop on Computer Software Structures Integrating AI/KBS in Process Control Systems, Lund, Sweden.*

Årzén, K.-E. (1996a): "A Grafcet based approach to alarm filtering." In *Proc. of the IFAC World Congress 1996.* Submitted to.

Årzén, K.-E. (1996b): "Grafchart: A graphical language for sequential supervisory control." In *Proc. of the IFAC World Congress 1996.* Submitted to.

David, R. (1995): "Grafcet: A powerful tool for specification of logic controllers." *IEEE Transactions on Control Systems Technology*, **3:3**, pp. 253–268.

David, R. and H. Alla (1992): *Petri Nets and Grafcet: Tools for modelling discrete events systems.* Prentice-Hall International (UK) Ltd.

Jensen, K. (1981): "Coloured Petri Nets and the invariant method." *Theoretical Computer Science, North-Holland*, **14**, pp. 317–336.

Jensen, K. (1990): "Coloured Petri Nets: A high level language for systems design and analysis." In Rozenberg, Ed., *Advances in Petri Nets 1990*, volume 483 of *Lecture Notes in Computer Science*, pp. 342–416. Springer, Berlin Heidelberg New York.

Jensen, K. and G. Rozenberg (1991): *High-level Petri Nets.* Springer Verlag.

Johnsson, C. and K.-E. Årzén (1994): "High-level Grafcet and batch control." In *Symposium ADPM'94—Automation of Mixed Processes: Dynamical Hybrid Systems*, Brussels, Belgium.

Johnsson, C. and K.-E. Årzén (1996): "Batch recipe structures using High-level Grafchart." In *Proc. of the IFAC World Congress 1996.* Submitted to.

López González, J. M., J. I. Llorente González, J. M. Santamaría Yugueros, O. Pereda Martínez, and E. Alvarez de los Mozos (1994): "Graphical methods for flexible machining cell control using G2." In *Proc. of the Gensym European User Society Meeting, Edinburgh, October.*

Moore, R., H. Rosenof, and G. Stanley (1990): "Process control using a real time expert system." In *Preprints 11th IFAC World Congress*, Tallinn, Estonia.

Nilsson, B. (1991): "En on-linesimulator för operatörsstöd," (An on-line simulator for operator support). Report TFRT-3209, Department of Automatic Control, Lund Institute of Technology.

Suau, D. (1989): *Grafcet colore: Conception et realisation d'un outil de generation de simulation et de commande temps reel.* PhD thesis, Universite de Montpellier (Montpellier II) Sciences et Techniques du Languedoc.