# AN OBJECT-ORIENTED STATIC META-MODEL FOR SEQUENTIAL FUNCTION CHARTS

**Pardo Martínez, Xoán C.** **Ferreiro García, Ramón**

*Systems Engineering & Automatic Control Group. University of A Coruña*
*E.S. Marina Civil, Paseo de Ronda, 51. 15011, A Coruña*
*e-mail: {pardo, ferreiro}@des.fi.udc.es*

Abstract: This paper introduces an object oriented static meta-model for Sequential Function Charts (SFC). This meta-model is being implemented as part of a visual programming tool for the development of distributed control software. It could be used as well to allow object oriented SFC tools to interchange SFC models. *Copyright ©2000 IFAC*

Keywords: grafcet, object oriented meta-model, sequential function charts (SFC).

## 1. INTRODUCTION

Sequential Function Chart (SFC), also known as Grafcet –this will be the name used in the rest of the paper-, was originally defined as a graphical formalism to describe the behaviour of sequential control systems (IEC, 1988), but its inclusion as part of the standard IEC 1131-3 (IEC, 1993) for the programming of programmable controllers, as well as the advances made in the formal definition of its semantics (Lhoste, et al., 1997) and the development of validation and verification methods (Zaytoon, et al., 1997) have turned Grafcet into a powerful tool that can be used during most of the phases of a control system life-cycle: specification, validation, verification, programming, supervision, monitoring and diagnosis.

The work presented in this article is part of the SFC++ project (Pardo and Ferreiro, 1999) that aims to implement a visual programming tool for the development of distributed control software based in the integration of Grafcet with object-oriented manufacturing reference models. SFC++ architecture is composed of two subsystems : a development subsystem and a run-time subsystem. The development subsystem includes a Grafcet editor to describe the dynamic behaviour of system objects. At the core of this editor, an OO static meta-model is used to define the Grafcet syntax. This article describes this meta-model in detail.

## 2. GRAFCET SINTAX

A Grafcet model can be defined as a tuple <G, I> where:

G=<S, T, L, $S_0$> is the Grafcet structure, a directed graph where S is a finite, non empty, set of steps; T is a finite, non empty, set of transitions; L is a finite, non empty, set of directed links, linking either a step to a transition or a transition to a step; and $S_0 \subseteq S$ is a non empty set of initial steps.

I=<R, A> is the Grafcet interpretation, where R is the set of receptivities (boolean conditions) associated to transitions ; and A is the set of actions associated to steps.

Graphical representation of Grafcet elements is shown in Fig. 1.

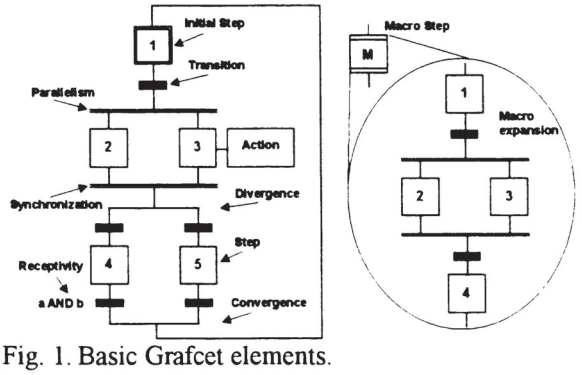


Fig. 1. Basic Grafcet elements.

The Grafcet structure defines the following elements: (a) *Steps*, that represent invariant states of a system. A step can be active or inactive. The set of active states at any given instant represents the *situation* of the system and, consequently, the set of initial steps $S_0$ defines its *initial situation*. (b) *Transitions*, that represent possible evolutions among system situations. A transition is said to be validated if every step having an incoming link with it is active. (c) Steps and transitions are linked together by means of *directed links*. A directed link always links a step to a transition or vice versa, but never two steps or transitions. It is possible to represent *divergence* and *parallelism* by linking one step to several transitions and one transition to several steps, respectively. In the same way *convergence* is described by linking several transitions to one step, and *synchronisation* by linking several steps to one transition. If a step (or transition) has no incoming links it is called *source step (transition)*, and if it has no outgoing links it is called *shaft step (transition)*.

With regard to the Grafcet interpretation, it defines the following elements: (a) *Receptivities*, that are boolean conditions associated to transitions. Whenever the receptivity of a validated transition is true the system evolves from one situation to another. (b) *Actions*, that are commands associated to steps. Actions are executed when the step becomes active. IEC (1993) defines several types of actions: pulse actions, conditional actions, limited actions, delayed actions, stored actions, etc.

But the description of complex control systems requires also support for hierarchical structures. Before introducing Grafcet hierarchical extensions the following partitions (Fig. 2) of a Grafcet must be defined: (a) A *connected Grafcet* is a Grafcet that for any pair of its components there is always a path between them. A path is a set of steps and transitions linked by directed links; (b) a *partial Grafcet* is a set of connected Grafcets; and (c) a *global Grafcet* is the set of partial Grafcets that describes a system. In a global Grafcet every connected Grafcet belongs to one and only one partial Grafcet.

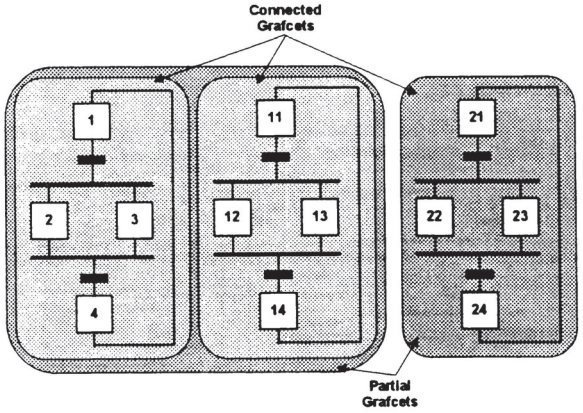


Fig. 2. Partitions of a Grafcet model.

Two hierarchy-related extensions to basic Grafcet definition have been proposed (UTE, 1992): *macro-steps* and *forcing orders*. A *macro-step* (Fig. 1) is an abbreviated representation of a connected Grafcet - called *macro-expansion*-. Macro-expansions must have only one input step and only one output step. There is a bijective relation between a macro-step and its macro-expansion. A *forcing order* allows forcing the situation of a partial Grafcet from another partial Grafcet. Forcing orders are specified as step actions and have the following syntax:

$$F/\ PG_n\ >\ \{Situation\}$$

where F/ indicates a forcing order, $PG_n$ is the "forced" partial Grafcet and {Situation} is the new situation of $PG_n$. Forcing orders define a total hierarchy among partial Grafcets.

## 3. GRAFCET OO STATIC META-MODEL

The proposed Grafcet static meta-model is based to a large extent on the one first proposed by Couffin, et al. (1997). The Unified Modeling Language (UML)[1] has been used to describe the meta-model and some well-formedness rules written in the Object Constraint Language (Warmer and Kleppe, 1999) have been added to clarify the meta-model semantics. Fig. 3 shows the meta-model package structure. Some packages (i.e. the ones labelled with "from common") do not belong really to the meta-model. They are used to integrate the meta-model into the SFC++ architecture.

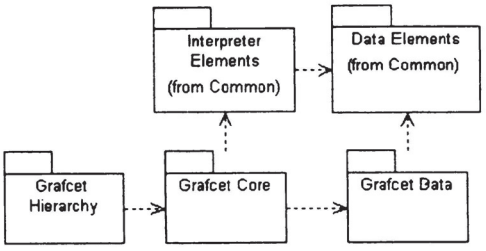


Fig. 3. Structure of the Grafcet static meta-model.

### *3.1. Common packages*

The Data Elements package (Fig. 4) defines those classes related to variables used in SFC++. *Variable* class is specified as a template class with three parameters: (1) *type*, the type of the value stored by the variable; (2) *access*, the kind of access allowed to the value of the variable: read access, write access and read/write access; and (3) *scope*, the localisation of the variable from the model point of view: external variables, internal variables visible from outside the model (shared variables) and internal variables not visible from outside the model (local variables). Both, access and scope types, have been defined as abstract classes.

[1] Sometimes the notation can be slightly different because the tool used does not support all UML features.

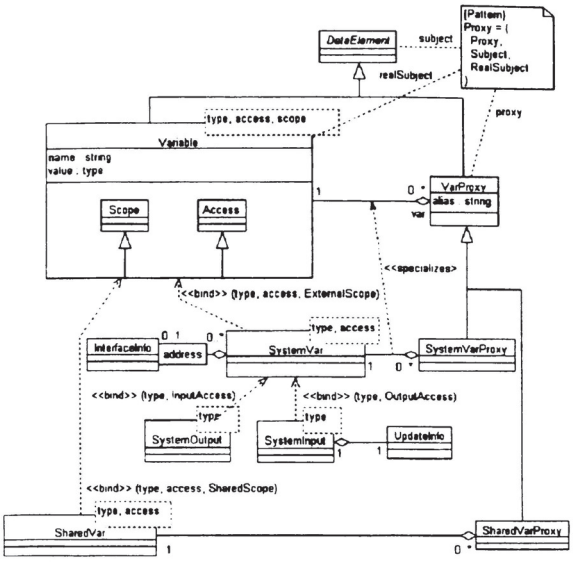


Fig. 4. Data elements package: data element classes.

Two specific kinds of variables are showed also in Fig. 4: (1) *SystemVar* class models physical variables. These variables have associated information about how to interface with the system to read or write them; and (2) *SharedVar* class models variables with shared scope and hence accessible from outside the model. In addition, the proxy pattern (Gamma et al, 1995) is implemented by means of the *VarProxy* class in order to allow a model to access variables defined outside it. This class stores a reference to a variable, an alias for its name and shares its common interface (defined in the *DataElement* class). Fig. 5 illustrates how *Var* and *Proxy* instances are used in models.

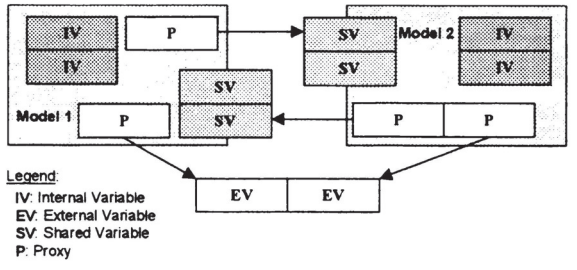


Fig. 5. Use of Var and Proxy classes.

The Interpreter Elements package (Fig. 6) contains a slight variation of the Interpreter pattern (Gamma et al., 1995). Classes needed to make the evaluation of expressions and the execution of commands implementation independent are defined in this package. For instance, the implementation of the evaluation of an expression like

```
(10 > 5) AND (true <> false)
```

can be done without any implementation concerns. By the contrary, the implementation of the evaluation of an expression like

```
↑var1 OR var2
```

requires at least two operations that depend on the implementation: to check up on var1 to detect a positive step and to read the value of var2.

The main classes defined in this package are: (a) *Enunciation,* that represents any piece of source code written in any high-level language; (b) *Expression,* an enunciation that can be evaluated to get a value. Expressions do not produce side effects and can contain *Terms*. Terms can be simple, in which case they are called *Factors* and they are used to read variables, or compound. Compound terms can contain expressions so they are used to model complex expressions; and (c) *Statement*, an enunciation that can produce side effects.

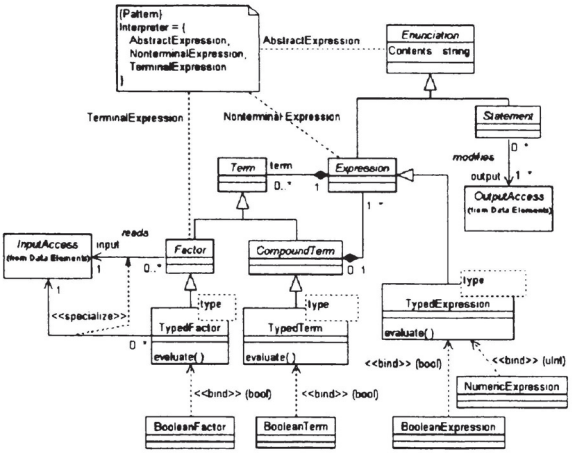


Fig. 6. Interpreter elements package.

### *3.2. Grafcet meta-model packages*

The Grafcet Data package defines enumerations and types of variables used in Grafcet models. Step bits, shared variables and local variables are modeled as bound classes produced from the Variable template class (Fig. 4).

The Grafcet Core package (Fig. 7) defines classes that model basic Grafcet elements: step, transition, receptivity, action, forcing order, macro step and connected Grafcet. For convenience an abstract class *GrafcetNode* has been defined as the base class of *Step* and *Transition*. Although there exist different types of steps they have been modeled using one only class *Step* with an attribute *type*. The reason is that differences between, for instance, an initial step and a 'normal' step are not structural : they have different graphical representations and their semantics are slightly different, but from the point of view of the Grafcet structure they are considered the same. With regard to source and shaft nodes they have been modeled by adding the *ports* attribute in the GrafcetNode class and modifying the multiplicity of the associations *goesBefore* and *goesAfter* between *Step* and *Transition*. 0..* multiplicity has been used instead of 1..* and the different cases have been modeled as constraints (see section 3.4). Connected Grafcet has been modeled with the *ConnectedGrafcet* class, that has a *groups* constrained association with the *GracfetNode* class (Fig. 11).

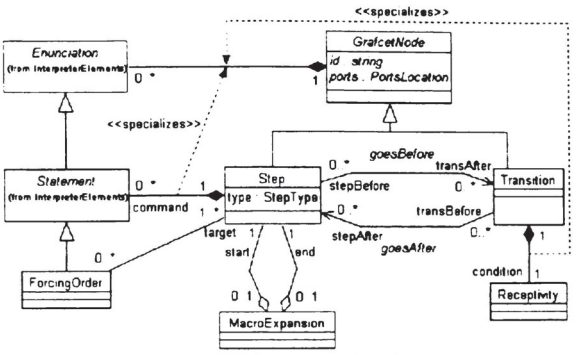


Fig. 7. Grafcet core package: main classes.

A macro step is the representation of a set of connected Grafcet nodes with a particular structure: there is only one initial step and only one final step. A question related to macro step comes forth: should it be considered a structural concept or, on the contrary, a graphical abbreviation and so left it outside the meta-model? It was decided to include it to model the constraint that Grafcet definition imposes in the macro expansion structure. So finally two classes (Fig. 8) have been added to the meta-model: *MacroStep*, that models macro step abbreviations, and *MacroExpansion*, that models macro step contents. For convenience, *MacroStep* has been defined as a constrained subclass of *Step*[2] and *MacroExpansion* as a subclass of *ConnectedGrafcet*. This solution is an application of the *Container* design pattern (Gamma, et al., 1995).

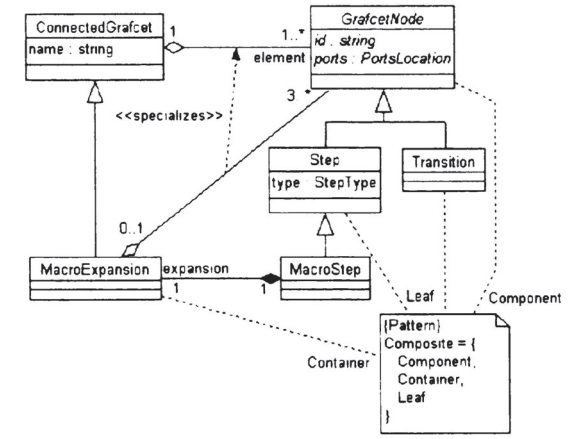


Fig. 8. Grafcet core package: macro-related classes.

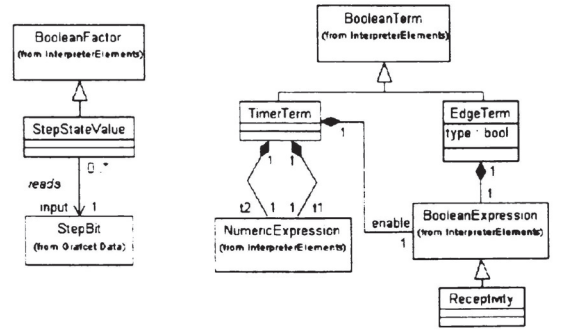


Fig. 9. Grafcet core package: Grafcet expressions.

[2] Macro steps don't have associated actions and they can't be forced (see section 3.4).

With regard to *Expression* related classes (Fig. 9), receptivities have been modeled with a *BooleanExpression* subclass, the *Receptivity* class. Also three types of terms have been considered: (1) *StepStateValue*, that models terms of the form $X_i$; (2) *TimerTerm*, that models expressions of the form $t_1\backslash expr\backslash t_2$, and (3) *EdgeTerm*, that models expressions of the form $\uparrow$(expr) and $\downarrow$(expr).

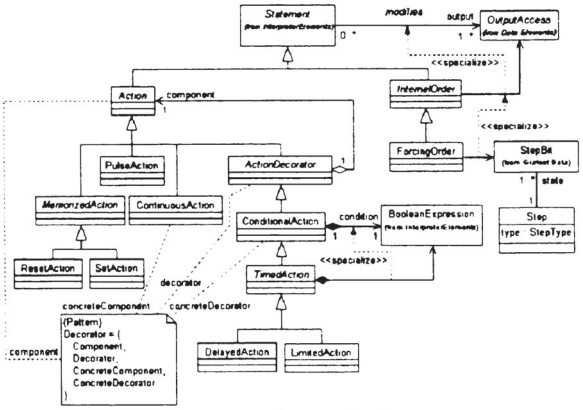


Fig. 10. Grafcet core package: Grafcet statements.

As for the *Statement* related classes (Fig. 10) two types have been modeled: forcing orders and actions. Because of forcing orders are internal orders and other could be defined in future Grafcet extensions, an *InternalOrder* abstract class has been added to the meta-model. This class constraints the *modifies* association between *Statement* and *OutputAccess* classes to allow internal orders to modify only output variables with internal scope. Therefore the *ForcingOrder* class has been defined as a subclass of the *InternalOrder* class that only modifies step bits. The modeling of actions is a bit more complicated. IEC (1993) defines several types of actions (e.g. N, P, R, S, C) that can be combined in different ways (e.g. SD, SL, DCL, CP, etc.) to more precisely describe the control action needed in each situation. To model this characteristic the decorator pattern has been applied (Gamma et al, 1995). Action types N, P, R and S have been considered basic actions which behaviour can be modified by C, D and L decorator actions[3].

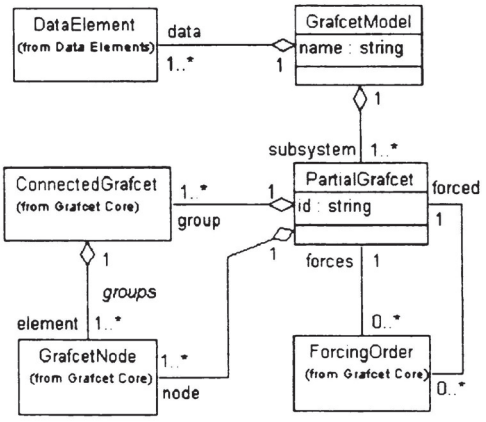


Fig. 11. Grafcet hierarchy package.

[3] The explanation of how the semantic of actions is implemented using this modelization is outside the scope of this article.

The Grafcet Hierarchy Package (Fig. 11) defines classes that model the two possible Grafcet hierarchies: the *containment hierarchy* and the *forcing hierarchy*. The containment hierarchy has been modeled by mean of aggregation associations among *GrafcetModel*, *PartialGrafcet* and *GrafcetNode* classes. With regard to the connected Grafcet concept, as Couffin, et al. (1997) have pointed out, it is more topological than structural. So at first it wasn't believed necessary to include it in the static meta-model. The reason because it has been finally included is that it constraints the structure a group of nodes can compose. This constraint is inherited by the *MacroExpansion* class guaranteeing syntactically well-formed macro steps. So at last it has been modeled with the *ConnectedGrafcet* class that is aggregated to the *PartialGrafcet* class and that has a constrained association with the *GrafcetNode* class. The forcing hierarchy has been modeled by means of the reified *ForcingOrder* association class[4] among *PartialGrafcet* instances. This association has been constrained to avoid loops in the forcing hierarchy.

### *3.3. Identification of Grafcet elements*

The identification strategy used to identify Grafcet elements is similar to the one first proposed by Couffin, et al. (1997). Each GrafcetModel has an unique alphanumeric identifier (attribute *name*). A Grafcet model contains a set of DataElements and a set of PartialGrafcets that have an unique *name* in the model. With regard to GrafcetNodes, each PartialGrafcet must have unique identifiers for steps, transitions and macro-steps, but it is legal for instance, to have in the same PartialGrafcet a transition with the same identifier as a step. The same could be said about ConnectedGrafcets. The solution proposed to identify macros is slightly different from the one proposed by Couffin, et al. (1997). Abbreviated references –usually of the form Mxx- are used as unique *identifiers* for macro abbreviations and macro contents textual descriptions are used as unique *names* for macro expansions.

### *3.4. Well-formedness rules*

To improve the meta-model semantics, the following OCL well-formedness rules have been defined:

model : GrafcetModel

(1) Partial Grafcets related by a forcing order pertains to the same global Grafcet.

```
model.subsystem->forAll(PartialGrafcet pg |
    pg.forces->forAll(ForcingOrder fo |
        model.subsystem->includes(fo.forced)))
```

(2) Forcing hierarchy has no cycles[5].

```
model.subsystem->forAll(PartialGrafcet pg |
    not(pg.forced*->includes(pg)))
```

(3) Global Grafcets have unique names.

```
model.AllInstances->forAll(GrafcetModel gm1, gm2 |
    gm1 <> gm2 implies gm1.name <> gm2.name)
```

(4) Partial Grafcets have unique names in each global Grafcet.

```
model.subsystem->forAll(PartialGrafcet pg1, pg2 |
    pg1 <> pg2 implies pg1.id <> pg2.id )
```

(5) Data elements have unique names in each global Grafcet.

```
model.data->forAll(DataElement de1, de2 |
    de1 <> de2 implies de1.name <> de2.name )
```

pg : PartialGrafcet

(1) Connected Grafcets have unique names in each partial Grafcet.

```
pg.group->forAll(ConnectedGrafcet cg1, cg2 |
    cg1 <> cg2 implies cg1.name <> cg2.name )
```

(2) Steps, transitions and macro abbreviations have unique identifiers in each partial Grafcet.

```
pg.node->forAll(GrafcetNode gn1, gn2 |
    (gn1 <> gn2 and gn1.type = gn2.type)
        implies gn1.id <> gn2.id )
```

cg : ConnectedGrafcet

(1) All nodes grouped by a connected Grafcet are connected among them.

```
cg.element->forAll(GrafcetNode node1, node2 |
    node1 <> node2 implies
        (node1.goesAfter*->includes(node2)
        or node1.goesBefore*->includes(node2)))
```

step : Step

(1) Multiplicity of goesBefore and goesAfter associations.

```
step.ports = #none        -- source and shaft step
    implies (step.transBefore->size = 0
            and step.transAfter->size = 0)
step.ports = #in          -- shaft step
    implies (step.transBefore->size > 0
            and step.transAfter->size = 0)
step.ports = #out         -- source step
    implies (step.transBefore->size = 0
            and step.transAfter->size > 0)
step.ports = #inOut       -- step
    implies (step.transBefore->size > 0
            and step.transAfter->size > 0)
```

trans : Transition

(1) Transitions can't be source and shaft at the same time.

```
trans.ports <> #none
```

[4] A reified association class is an association class modelled as a class. It is used when several association class instances are possible for any pair of related objects (Rumbaugh, et al., 1999).

[5] The notation *object.association** indicates the set of all objects that have a direct or indirect *association* with *object* (*association* transitive closure). This feature is not included in OCL.

(2) Multiplicity of goesBefore and goesAfter associations.

```
trans.ports = #in          -- shaft transition
    implies (trans.stepsBefore->size > 0
            and trans.stepsAfter->size = 0)
trans.ports = #out         -- source transition
    implies (trans.stepsBefore->size = 0
            and trans.stepsAfter->size > 0)
trans.ports = #inOut       -- transition
    implies (trans.stepsBefore->size > 0
            and trans.stepsAfter->size > 0)
```

macro : MacroStep

(1) Source and shaft macro abbreviations constraint the type of their expansion step and end steps.

```
macro.ports = #none        -- source and shaft macro
    implies (macro.expansion.start.ports = #out
            and macro.expansion.end.ports = #in)
macro.ports = #in          -- shaft macro
    implies (macro.expansion.start.ports = #inOut
            and macro.expansion.end.ports = #in)
macro.ports = #out         -- source acro
    implies (macro.expansion.start.ports = #out
            and macro.expansion.end.ports = #inOut)
macro.ports = #inOut       -- macro
    implies (macro.expansion.start.ports = #inOut
            and macro.expansion.end.ports = #inOut)
```

(2) Attribute *type* has always the value #macro.

```
macro.type = #macro
```

(3) Macros don't have actions.

```
macro.command->size = 0
```

macroContents: MacroContents

(1) Start and End steps are different.

```
macroContents.start <> macroContents.end
```

(2) Start step have at least an outgoing link.

```
macroContents.start.ports = #inOut or
macroContents.start.ports = #out
```

(3) End step have at least an incoming link.

```
macroContents.end.ports = #inOut or
macroContents.end.ports = #in
```

timerAction : TimerAction

(1) The condition associated to timer actions contains one and only timer term.

```
timerAction.condition.term->size = 1 and
timerAction.condition.term->asSequence->
            ->at(1).oclIsTypeOf(TimerTerm)
```

intOrder : InternalOrder

(1) Internal orders modify only output variables with internal scope.

```
intOrder.output->forAll(OutputAccess outputVar |
outputVar.oclIsKindOf(InternalScope))
```

forcingOrder : ForcingOrder

(1) Macros can't be forced.

```
forcingOrder.forced->forAll(Step step |
    step.type <> #macro )
```

(2) All forced steps are contained in the forced partial grafcet.

```
forcingOrder.target->forAll(Step step |
    forcingOrder.forced.node->includes(step))
```

action : ActionDecorator

(1) Actions has at most one decorator of each type.

```
action.component*->forAll(Action action1, action2 |
    action1.oclType <> action2.oclType)
```

## REFERENCES


Couffin, F., S. Lamperiere and J.M. Faure (1997). Contribution to the Grafcet formalisation: a static meta-model proposition. *Journal européen des systèmes automatisés*, **vol. 31, n° 4**, pp: 645-667.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Reading, MA.

IEC (1988). *Preparation of function charts for control systems*. International Electrotechnical Commission, publication 60848.

IEC (1993). Programmable controllers – part 3: their programming languages. International Electrotechnical Commission, publication 61131-3.

Lhoste, P., J.M: Faure, J.J. Lessage and J. Zaytoon (1997). Comportement temporel du Grafcet. *Journal européen des systèmes automatisés*, **vol. 31, n° 4**, pp: 695-711.

Pardo Martínez, X.C. and R. Ferreiro García (1999). SFC++: a tool for developing distributed real-time control software. *Microprocessors and microsystems journal*, **vol. 23, n° 2**, pp: 75-84.

Rumbaugh, J., I. Jacobson and G. Booch (1999). *The Unified Modeling Language reference manual*. Addison-Wesley, Reading, MA.

UTE (1992). *Function Charts Grafcet: Extensions of Basic Principles*. Union Technique de l'électricité, Document UTE C03-191.

Warmer, J.B. & Kleppe, A.G. *The Object Constraint Language: Precise Modeling with UML*. Addison-Wesley, Reading, Mass., 1999.

Zaytoon, J., J.J. Lesage, L. Marce, J.M. Faure and P. Lhoste (1997). Verification et validation du Grafcet. *Journal européen des systèmes automatisés*, **vol. 31-n°4**, pp: 713-740.