Grafcet:
an Analysis Pattern for Event Driven Real-time Systems

Nathalie Gaertner Bernard Thirion

Laboratoire EEA, Groupe LSI
Université de Haute-Alsace
12, rue des frères Lumière
68093 Mulhouse Cedex, France
Tel.: +33 (0)3.89.33.69.72 – Fax: +33 (0)3.89.33.69.69
N.Gaertner@essaim.univ-mulhouse.fr B.Thirion@ essaim.univ-mulhouse.fr

Abstract: *In software developments, the simple succession of operations can easily be hard-coded. However, to control event driven systems, applications must conform to a well-defined sequence of operations, even simultaneous ones, that depends on a variable environment, on time constraints and on synchronization constraints between several sequences. In that kind of applications, too complex intertwined conditional statements must be avoided as well as blocking wait operations. Conditional statements quickly become complex and thus incomprehensible, particularly when several actions must be done simultaneously, whereas wait statements force all the application in a blocked state. Tasks can provide a solution for these kinds of problems. However, difficulties appear when no tasks support is provided by the chosen programming languages, when the number of parallel working tasks is limited, or when describing and implementing synchronization mechanisms are too complex. Moreover, the global behavior of the application is difficult to understand and deadlock situations can occur. Thus, specifying and reusing a flexible, reliable and on-line modifiable construction that defines sequences of different kinds of parallel actions is an easy, generic approach to this problem. The "Grafcet pattern" specifies such a construction. Its description in a pattern form not only provides an architectural solution but also a documentation about its use, the reasons for this particular solution, the benefits and drawbacks related to that pattern, the applicability context, etc.*

Keywords: Grafcet, business pattern, discrete event, sequential, design pattern.

1. Introduction

Synthesizing recurrent solutions through patterns originated from the pioneering work of architect Christopher Alexander at the end of the seventies [1]. The software community borrowed Alexander's idea of using patterns and pattern languages to adapt it to software developments. The now well-known design patterns first emerged from this adaptation [3, 5, 9, 14]. Analysis patterns [8] have introduced the concept of pattern in the analysis phase and provide reusable domain specific solutions. However, the word 'analysis pattern' does not point out that analysis and design solutions are provided. 'Business pattern', also used by R. Haugen at PLoP'97 [11], seems therefore a more appropriate word. Business patterns are a way to capture and communicate domain specific expert knowledge. They offer analysis and design reuse within their domain and take advantage of the pattern concept to document solutions of recurrent problems.

This paper describes such a kind of business pattern. The pattern is called "Grafcet" and is related to the domain of process control and more precisely to control for discrete event processes. The Grafcet [2, 4, 6, 12], close to Sequential Function Charts (SFC inherits from Grafcet), allows to control discrete event process. It provides a structure and behavior description about how and when performing control actions. Complete specifications of operations, which may be complicated and performed simultaneously, can thus be defined in an elegant flexible reliable manner.

The main differences between Petri nets [6] and Grafcet are that the later one is standardized and allows the definition of an overall description of a control system and may, via progressive levels of description, proceed to a description in which all the details are revealed. Statecharts [10] or the UML variant of statecharts [7] can also be used to describe control strategies of discrete event processes, but in the control field, Grafcet is the most widely used and known notation. Therefore, the Grafcet notation and vocabulary were used to facilitate the understandability and the mapping between Grafcet models and the implemented software.

As discrete event control is important and appears in many kinds of industries (manufacturing, automobile, chemistry, etc.), the pattern can be widely (re)used. When implemented, it is possible to achieve low level control, as well as supervision or fault diagnostics.

2. Grafcet Pattern

The "Grafcet pattern" provides an elegant solution in a pattern form to specify discrete event process controllers. The following section describes this business pattern by using a presentation format, called template, adapted from [9] and UML [7, 13] as modeling language.

Grafcet

`Intent`

Define extensible, flexible and dynamically modifiable sequences of actions (even simultaneous ones[1]) with time dependencies to achieve discrete event process control.

`Motivation`

Consider a simple chemical process that combines two reagents to produce the final result. It might work by first pouring enough of one reagent into a container to reach a particular level, then pouring enough of the second reagent until a second level is reached (while mixing both reagents), and then pouring out the product.

Figure 1 shows the system that is to be controlled.

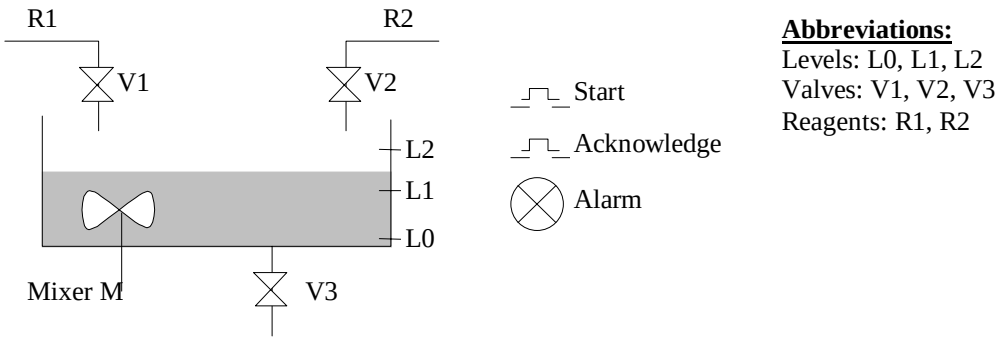

Figure 1: A chemical process example

The desired sequence of operations is the following:

1) When the start button is pressed, V1 should be opened until level L1 is reached.
2) When L1 is reached, the mixer should start mixing and simultaneously V2 should be opened.
3) When L2 is reached, the mixer should stop, V3 should be opened until the tank's level goes under L0.
4) If after 10 minutes the level of the tank is not under L0, an alarm is started. The "acquit" button stops the alarm and allows restarting the control process.

The grafcet of Figure 2 uses many Grafcet concepts to define a possible control sequence.

[1] Simultaneous actions can be performed with a single CPU, if the CPU processes discrete events and actions quickly enough to be "seen", from the outside, as simultaneous.

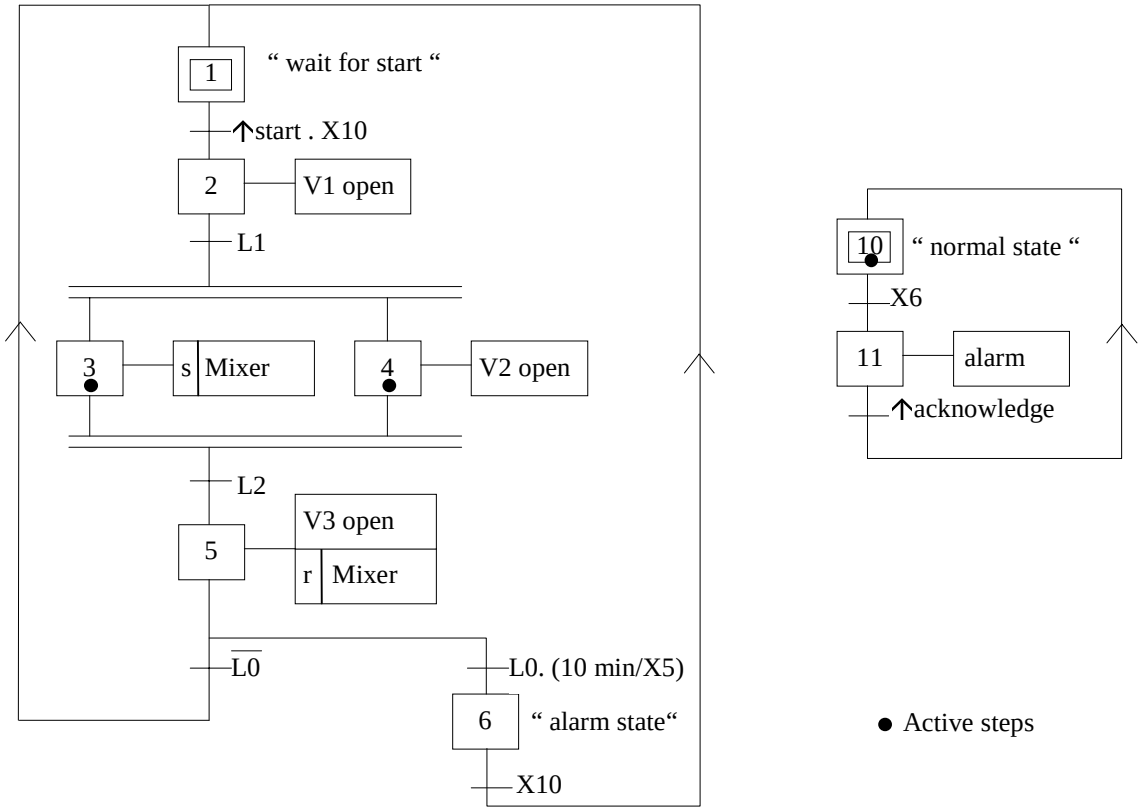

Figure 2: Grafcet of the tank filling example

The software that controls this chemical process must issue commands that depend on the system internal state, on time constraints, and on the environment's state. It might have to perform several commands simultaneously. The definition and implementation of the sequences of actions is carried out by the Grafcet pattern, without using tasks and complex synchronization mechanisms.

Domain

Grafcet is a standardized graphical language [12] and a modeling tool that can be applied to describe the behaviors of discrete event systems (dynamic systems that evolve according to the asynchronous occurrence of events). It was primarily developed for Programmable Logic Controllers (PLC), which are specialized computers used in industrial automation. Each PLC constructor provides specific tools to compile the grafcets and generate executable code for their computer. These tools range from interactive development environments to simple console applications.

Grafcet is based on a few key concepts: “steps”, “actions”, “transitions” and “transition conditions”. A Grafcet structure is divided into *steps*, separated by *transitions* (alternately). Each step has a set of *actions* that are performed if the step is active. At a given time during the execution of a grafcet, many steps can be active. Each transition has an associated logic *condition*; if the condition is true, the transition can be fired. The firing of transitions changes the set of active steps and thus indicates a modification of the controlled process state.

For example, figure 2 shows the Grafcet of the tank filling example shown in Figure 1. “Wait for start”, “V1 open”, “Mixer”, “2”, “5” are all steps. Step 3, 4, 10 are active at the same time. The action “V2 open” will not start until after “V1 open” is finished. L1 is the condition of the transition between “V1 open” and “V2 open”. If L1 is true the transition is fired.

Each *step* symbolizes a state or part of a state of the system (steady situations). A binary variable Xi is related to each step (in which i must be replaced by the step label) to record the active or inactive state of the step. The variable is true when its associated step is active, false otherwise. Initial steps can also be defined to specify the active steps at the beginning of the control process (initial state).

Actions are outputs, which send commands to the process to be controlled. Each action is related to a specific step. If the step is active, the associated actions are performed; respectively ignored when the step is idle. Several steps can

be active at the same time so that simultaneous actions can be performed. A “command pattern” [3, 9] can be used to model these abstractions. Decorations such as set or reset specifications can be added to define stored actions.

Transitions are associated to directed links. They link steps together and thus define possible evolutions of the system’s state. We use the words “downstream steps” of a transition to designate steps that are after the transition (according to the orientation of the transition), respectively “upstream steps” for steps before the transitions. Similarly, the words “upstream transitions” or ”downstream transitions” of a step can be used.

Each transition has an associated transition condition, which defines the condition that must be satisfied to clear / fire this transition. The clearing represents the evolution of the state of the system, as it invalidates all the immediate preceding steps and validates all the immediate following steps. To activate one or several downstream steps two conditions must be fulfilled:

1. the transition must be enabled: every immediate preceding step must be active.
2. it must be possible to fire the transition: the associated transition condition must be true.

If several steps are linked to a unique transition then we use the word “conjunction transition”, whereas when a step is linked to several transitions, we use the expression “disjunction transition”.

Transition conditions are logic values or logic expressions that can involve:

- input variables (measured by sensors for example)
- true (=1) or false (=0)
- temporal conditions
- step variables Xi.
- AND operators symbolized with “.”
- Or operators symbolized with “+”
- Not (expression) symbolized with expression
- Rising edge of an expression symbolized with ↑(expression)

For example, (X1 + (Speed >=2000). Stop) is a boolean expression that can be used as a condition.

Time dependencies allow taking time constraints into account. The standard defines the following notation: t1/Xi/t2. Xi is the step variable of the step named i; t1 defines the time to wait after step i became active; and t2 defines the delay after step i changes from active to inactive (the time unit must be specified).

Additional features exist in the Grafcet standard (conditional actions, delayed actions, …), but no details about them will be given here, because these specifications can be included in the actions/command pattern.

Grafcet notation:

Grafcet is based on a simple and easy to understand graphical notation.

- A step is represented by a square and is generally labeled with an integer. When a step is active, a dot is added in the square. A starting step is drawn with a double square.

- An action is specified by a written or symbolic statement inside a rectangle connected to a step. The set decoration is added writing an “S” in front of the action. The reset uses an “R”.

- Each transition is symbolized by a link, which associates steps together. Without any decoration, the link implies a top down direction; an arrow can be used to force a special direction. A vertical dash crossing the transition symbolizes the transition condition. Its associated logic expression is indicated beside this dash. A double horizontal line stands for “And transitions”, whereas simple lines are used for “Or transitions”.

The example given in the motivation section illustrates how to use and draw Grafcets.

Applicability

Use Grafcet to model the software when:

- it must respond to events from the outside world
- it must carry out complex sequences of actions
- it must control several individual elements at once
- time and synchronization constraints are involved and you don’t want to implement tasks
- controlling a process with different operating phases (each phase has a corresponding step in the Grafcet)
- flexibility is required, even at run-time, to change the Grafcet structure and thus the control behavior.

Structure

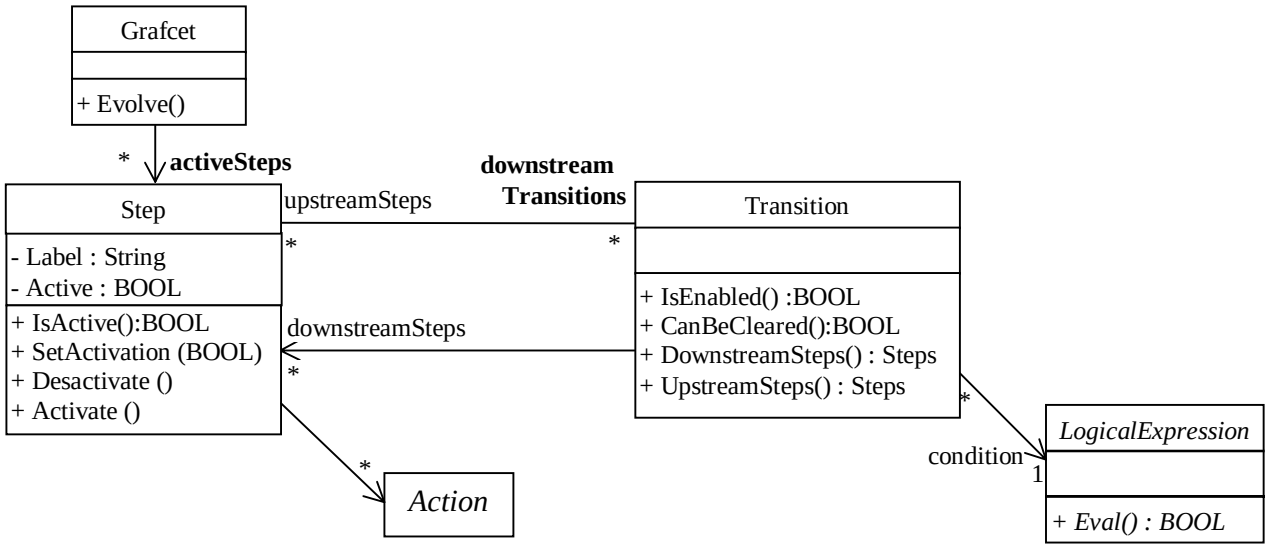

Figure 3: Grafcet's structures

Participants

❖ Grafcet

The control sequence uses the following cycle:

Inputs' acquisition & memorization
Evaluation of the evolution of the Grafcet
Perform actions

Thus, the evaluation undertakes the firing of the transitions, the step updates and defines the values of the actions to be performed.

❖ Step

This class contains information about steps (label, active or idle, …). The name of the object identifies and names the Step (S1: Step, for example).

❖ Transition

It provides a unique interface to declare transitions. Two kinds of transitions are possible: "And transitions" and "Or transitions". Inheritance can be avoided to create dynamically mutable transitions, because:

First case: AND Disjunction

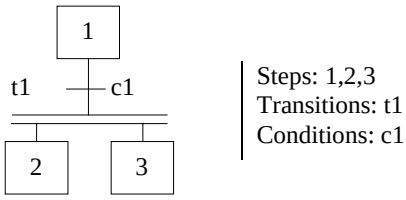

Figure 4: And Disjunction

If step 1 is active and c1 is true, then all the downstream steps of t1 are validated and the upstream steps are invalidated.

Second case: OR Disjunction

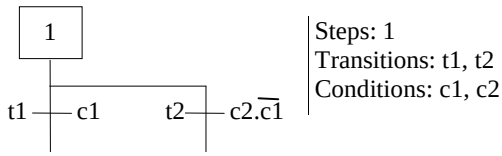

Figure 5: Or Disjunction

The associated condition of the t2 transition is modified to define an "effective" selection (c2 and not c1). Thus, if step 1 is active, then t1 and t2 can not be both cleared.

And/Or conjunctions are based on the same principle.

❖ LogicalExpression

It defines an abstract interface for simple or complex logical expressions. It can be based on an interpreter pattern [9] to define simple or complex expressions with intertwined AND, OR, NOT operators. Terminal expressions can be variables, values, step variables (Xi) or timers (for time dependencies).

❖ Action

This class specifies an abstract interface for different kinds of actions. It has a command pattern [3, 9] like design.

Collaboration

The sequence diagram in Figure 7 shows a simple evolution of the following example:

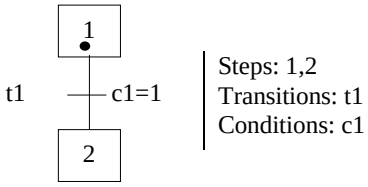

Figure 6: Simple Grafcet example

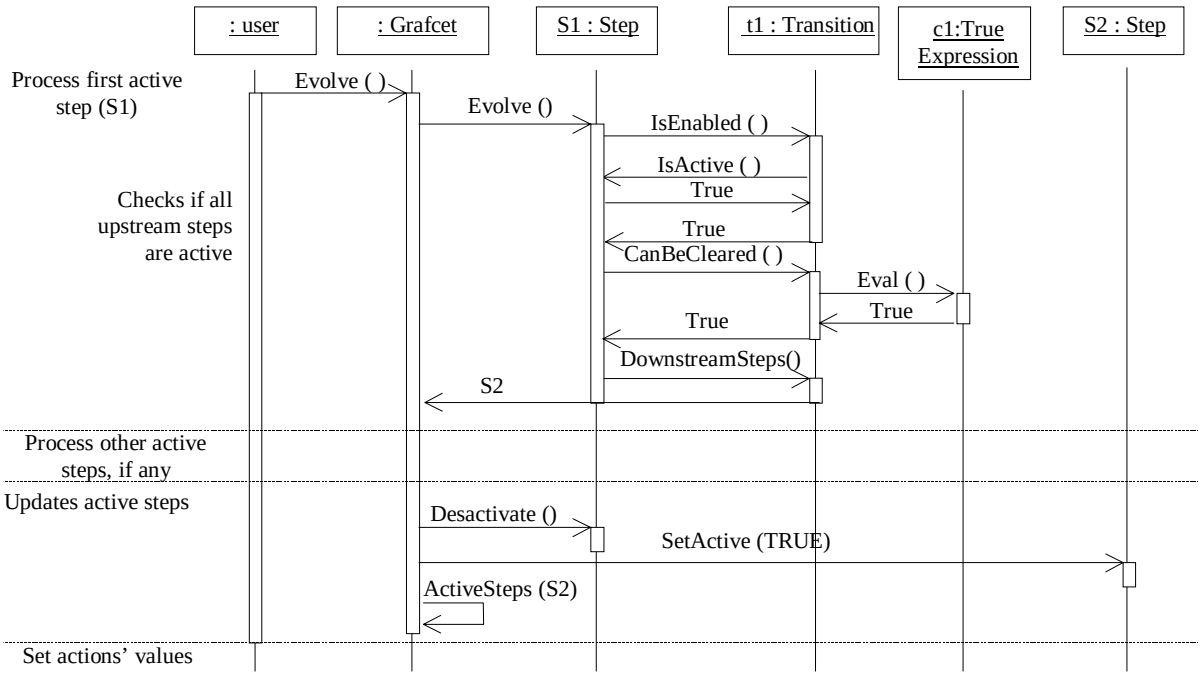

Figure 7: Sequence diagram using approach 2

The Grafcet memorizes all the active steps. To evolve, it delegates to each active step the responsibility to look for the new active steps. When all the active steps are processed, it updates its active steps (the new ones replace the previous ones).

Difficulties appear when "And conjunction" transitions are defined. The transition associated to an "And conjunction" must take into account the states of **all** the immediate upstream steps. Therefore, transitions must have a link to all their upstream steps and steps must know to which downstream transitions they are connected.

Lets take the following example to illustrate the collaborations between instances of the previous defined classes:

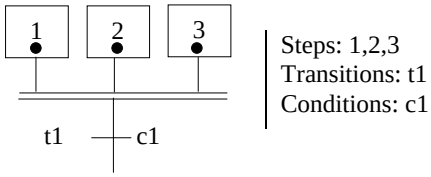

Figure 8: Clearing of an "And Conjunction"

The Grafcet object checks all its active steps. It starts with step 1. The downstream transition of step 1 is t1. To evaluate if the t1 transition can be cleared, steps 1, 2 and 3 must be active. To verify this, the t1 transition must check if all its upstream steps are active. Therefore, step 1 delegates this evaluation to t1. If all upstream steps of t1 are valid, then t1 evaluates its associated c1 condition.

Consequences

Succession of simple or complex actions, even simultaneous ones, and of any kind (thanks to unified approach of OT), can be defined.

Tasks and inter-process communication techniques supported by operating systems or high level languages, such as the Ada language, are avoided when using this pattern.

Steps, transitions or logical expressions can be created or modified dynamically. Thus, the behavior of the grafcet can be easily changed if needed. For example, lets us extend the example of the motivation section with adding a heating action while pouring reagent 1 and 2 (suppose that a heating mechanism exists). A possible grafcet is shown in Figure 9.

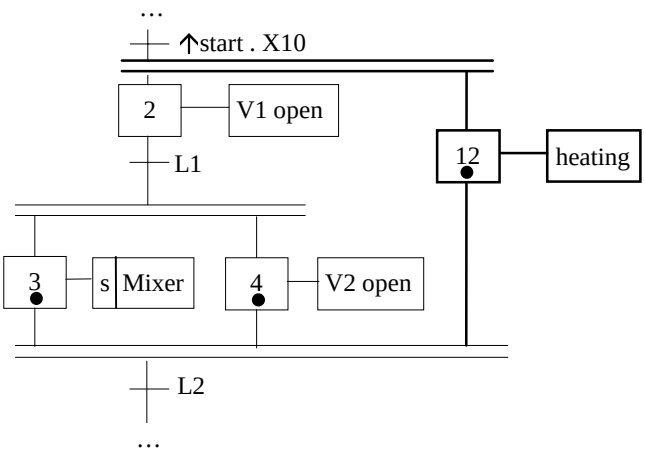

Figure 9: Extension of the filling tank example

The addition of a heating mechanism needs the creation of only 4 objets (one step, two transitions and one action) that are then connected to the existent grafcet objects. Although the heating should be performed in parallel with the tank filling and mixing, adding this new behavior is easy and can even be done at run-time.

Performances fluctuate depending on the Grafcet structure and more precisely on the number of steps connected to "and conjunctions". As active steps are memorized, only downstream transitions of active steps will be checked. For large and complex Grafcet objects with only few active steps, it is a beneficial approach. However, for convergent "And transitions", reduced performances are obtained. Indeed, lets assume that steps 1, 2 and 3 are active (see Fig. 8) and are upstream from t1 (And transition). For step 1, the Grafcet object will evaluate the t1 transition. As t1 has to check all its upstream steps, three checks are done (is step 1 active, is step 2 active and is step 3 active?). As the result is true, the condition associated to t1 will also be evaluated. Finally the transition can be cleared, and the downstream steps of t1 are memorized to replace, at the end of all the evaluations, the previous active steps. Then, step 2 (as it is also and active step) will be processed. The same verifications as for step 1 will be undertaken. For step 3, it will be also the same. Thus, with "And conjunction" transitions there are evaluation redundancies.

A variation of the Grafcet's structure avoids this kind of redundancies. The Grafcet class can keep track on all its transitions instead of its active steps. To evolve, a Grafcet object needs to delegate to each transition the responsibility to look if the transition is enabled and can be cleared/fired. Therefore, each transition knows its upstream steps, is able to check if these steps are active, and, in the case they are, it can evaluate its associated logic condition. If the transition is fired, the upstream steps are marked as invalid and the downstream steps as valid. Each transition will be evaluated only ones. However, **all** the transitions of the grafcet will be evaluated at each evolution; if the grafcet is big, time will be wasted while looking for the firing possibilities of each transition.

Performances are related to the chosen approach (initial structure or variation), the Grafcet's structure and the number of active steps. In most cases, the initial approach (the Grafcet object memorizes all its active steps) is better, because often there are:

- less active steps than transitions
- not so many "And conjunction" with lots of upstream steps.

Another consequence is the partial consistency checking of the Grafcet's structure. As the Grafcet pattern is based on a design that requires linking transitions to steps and steps to transitions, the alternation of step/transition is fixed.

User interfaces such as those for a coffee machines or ATM could use the Grafcet pattern. For instance, a coffee machine displays information and processes events according to the state of the machine (initial state, insertion of coins, payment ok, selection of coffee, cup down, …). However, not every event-driven application should use this pattern. For example, graphical user interfaces systems are event driven but the Grafcet pattern does not provide an effective solution for most of them. Indeed most of GUIs have not to record their state to validate or disable some actions. That means that they do not need to response to events with chronological constraints (a user should be allowed to click on a menu item, on another window, on any button, etc. with no fixed chronology).

Implementation

Consider the following issues when implementing the Grafcet pattern:

1. *Updating active steps*

   When evolving a `Grafcet` instance, updating the active steps should be done after checking all the downstream transitions of all active steps. Consider the following example to explain the problems when the update is done immediately.

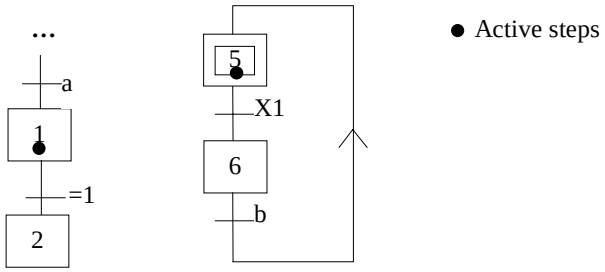

Figure 10: Simple example

   If processing step 1 involves setting step 1 as inactive and step 2 as active, then when step 5 will be processed it won't be possible to fire the downstream transition of step 5 although it should be.

To implement the `Evolve` method of the `Grafcet` class, a temporary list is used to memorize the next active steps.

```
void    CGrafcet::Evolve() {
   CStepList newActiveSteps;
   CListIterator<CStep*> iter(&m_activeSteps);

   for (iter.First(); ! iter.AtEnd(); iter.Next() ) {
        CStepList steps; //potential new active steps
        if (iter.Current()->CanEvolve(steps) ) {
              CListIterator<CStep*> iterNewActive(&steps);
              for (iterNewActive.First(); ! iterNewActive.AtEnd();
                                                    iterNewActive.Next() )
                    newActiveSteps.AddTail(iterNewActive.Current());
        }
   }
   UpdateActiveSteps();
   ProcessActions();
}
```

2. *Deleting all the steps and transitions when using a language without garbage collection*

   A `Grafcet` instance keeps track of all its active steps. To delete all the objects involved (steps, transitions, etc.), a solution is to design a `GrafcetFactory` that is responsible for managing the Grafcet's objects.

3. *Integrating step variables in transition conditions to synchronize different grafcets*

   It should be easy to create a condition transition based on a step variable. For example, as shown in Figure 11, to synchronize the G1 grafcet with the "normal state" of grafcet G2, the step variable X10 can be used in G1.

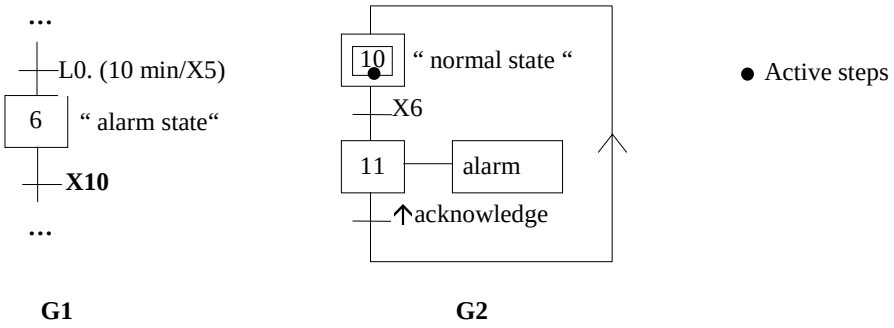

Figure 11 : Synchronization example

The `Step` class can be declared as a subclass of `LogicalExpression` and the `Eval` method overridden. As the `Step` class has an attribute that records the active or inactive state, the implementation is as easy as:

```
Bool   CStep::Eval() const { return m_active }
```

The declaration of the X10 condition transition is then done by writing:

```
CStep S10;
CTransition TdownS6;
TdownS6.Condition(&S10);
```

An alternative would be to use an adapter pattern. An intermediate `StepVariableLogicalExpression` class, that inherits from `LogicalExpression` and that is associated to the `Step` class, can be created; but then, when using this solution an additional object has to be created:

```
CStep S10;
CStepVariableLogicalExpression X10(&S10);
CTransition TdownX6;
TdownX6.Condition(&X10);
```

4. *Implementing time constraints like t1/Xi/t2 (cf. Domain section, Transition conditions) with no blocking wait statements*

   The simplest way to include delays in transition conditions is to use timer objects. A timer object is typically initialized with a value representing a duration and has at least three methods: one for starting the timer, one for stopping it and one to check if it is timed out. The implementation of these methods relies on the operating system API that gives the computer internal time or tick counts.

   The issue is then how and when starting timers and checking for the `IsTimedOut` method. The following C++ code fragments outlines one solution.

   - A specialized `LogicalExpression` class is created for delayed transition conditions:

```
class CTimedLogicalExpression : public CLogicalExpression {
public:
     CTimedLogicalExpression(CTimer t1,CStep*,CTimer t2=CTimer(0));
     CTimedLogicalExpression(CStep*,CTimer t2);
     // t1 : time to wait after the step became active
     // t2 : delay after the step changes from active to inactive

     virtual      ~CTimedLogicalExpression();
     virtual BOOL IsTrue() const {return (m_t1.IsTimedOut()&& m_t2.IsTimedOut()};
private:
     CTimer m_t1, m_t2;
     CStep* m_pStep;
};
```

   - Steps are linked to their dependent timers to be able to start and stop them.

```
class CStep {
public:
     ...
     SetActivation(BOOL);
     ...
private:
     CList<CTimer*>      m_t1Dependents, m_t2Dependents;
     ...
}

Void CStep::SetActivation (BOOL state) {
     if (m_active == state) return;
     m_active = state;
     CListIterator<CTimer*> iter1(&m_t1Dependants);
     for ( iter1.First(); !iter1.AtEnd(); iter1.Next() )
           if (m_active)
                 iter1.Current()->Start();
           else
                 iter1.Current()->Stop();

     CListIterator<CTimer*> iter2(&m_t2Dependants);
     for ( iter2.First(); !iter2.AtEnd(); iter2.Next() )
           if (m_active)
                 iter2.Current()->Stop();
           else
                 iter2.Current()->Start();
     }
}
```

   - The steps reference their dependent timers; these references are created when a delayed condition is created:

```
CTimedLogicalExpression ::CTimedLogicalExpression (CTimer t1, CStep* pStep,
                                                   CTimer t2)
{
     m_t1 = t1; m_t2 = t2;
     m_pStep = pStep;
     //register timers
     pStep->Addt1Dependent(&m_t1);
     pStep->Addt2Dependent(&m_t2);
}
```

5. A user-friendly interface can also be created to create Grafcet's structures and to generate corresponding objects, instances of the Grafcet pattern classes.

Related patterns

An interpreter pattern [9] can be used to design logical expressions.
A command pattern [3, 9] can model actions.
The state pattern [9] is another approach that can be used in conjunction with statecharts.

3. Acknowledgment

The authors thank Ralph E. Johnson for valuable comments, insights and contributions.

4. Conclusion

Software patterns are a way to transfer and to share expert knowledge by providing a solution to a general software problem in a particular context and by documenting the solution. Starting from this concept, business patterns can de defined. They are special kinds of software patterns related to a specific business or domain. Thus, a business pattern is a reliable and documented abstraction that solves a specific domain problem. It documents, explains and communicates expert domain-dependent knowledge, independently of any programming language.

We have used the business pattern concept to explain a solution to a common problem within the discrete event control domain. The solution is based upon the Grafcet standard. As Grafcet is suitable and widely used for solving real industry problems, this pattern can be integrated in existing projects or in new developments. The benefits rely in using a known notation from experts of this domain coupled with an object-oriented design, which allows the integration of different objects from other domains.

5. References

1. Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I. and Angel S. A Pattern Language. Oxford University Press, New York, 1977.
2. Baracos P. Grafcet step by step, Famic Inc. editor, 1992.
3. Bushmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M. Pattern-oriented software architecture : a system of patterns. Wiley, 1996.
4. Bouteille N., Brard P., Colombari G., Cotaina N. and Richet D. Le Grafcet, Toulouse: Cepaduès Editions, 1992.
5. Coad P. Object-oriented patterns. Communications of the ACM 35(9) pp. 152-159, September 1992.
6. David R. and Alla H. Petri Nets and Grafcet tools for modelling discrete event systems, Prentice Hall Ed., London, 1992.
7. Douglas B. P. Real-Time UML, developing Efficient Objects for Embedded Systems, Addison Wesley, Reading, MA, 1998.
8. Fowler M. Analysis Patterns: Reusable Object Models. Addison-Wesley, Reading MA, 1997.
9. Gamma E., Helm R., Johnson R. and Vlissides J. Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading MA, 1995.
10. Harel D. Statecharts: a visual formalism for complex systems, Science of computer programming, 8, 1987, p. 231-374.
11. Haugen R. Dependent Demand – a Business Pattern for Balancing Supply and Demand in the 4th Pattern Languages of Programming Conference, Washington University Technical Report 97-34.
12. IEC (International Electrotechnical Commission), Preparation of Function Charts for Control Systems, IEC 848, 1988.
13. Object Management Group. Unified Modeling Language – UML Notation Guide Version 1.1. Ad/97-08-05. Framingham, Mass. November 1997.
14. Pree W. Design Patterns for Object-Oriented Software Development. Addison-Wesley, 1995.