TOWARDS A VALIDATED OBJECT ORIENTED DESIGN APPROACH TO CONTROL SOFTWARE

Alban Rasse, Jean Marc Perronne, Bernard Thirion

MIPS Laboratory, LSI Group
University of Haute Alsace, ESSAIM
12 rue des frères lumière, 68093 Mulhouse cedex, France

Abstract: Due to their increasing complexity, the development of control software has become a difficult task. It is therefore necessary to consider a rigorous process for software design which integrates the different design phases in a unified manner. This is the aim of the present paper which proposes a coherent approach based on models that guarantee the development of validated applications. From an *analysis model*, the approach helps to obtain both a *validation model* which can be exploited with existing model checking tools and a specific *implementation model* which conforms to the validated model. *Copyright © 2005 IFAC*

Keywords: Software Engineering, Control System Synthesis, Object Modeling Technique, Models, Validation, Implementation

1. INTRODUCTION

The design of software systems has become more and more difficult due to their increasing complexity. It is the same for control software (Sanz, *et al.,* 2001) in which the constraints of concurrence, interaction and synchronization, make their comprehension awkward. To allow the development of reliable applications, such complexity must be controlled and design accuracy must be ensured at all levels. The bottom up Object Oriented methods, based on reusable entities, help to understand this complexity and to make the design process easier (Booch, 1994). Despite the undeniable advantage in terms of productivity and intelligibility, this approach does not guarantee a suitable global behavior. So, it becomes necessary to check and validate the software systems modeled in this way before their implementation into real systems.

A series of approaches try to introduce more formal aspects and semantics into the *Unified Modeling Language* (UML) (Object Management Group, 2003) specification of systems to allow their validation (Mikk, *et al.,* 1998; Apvrille, *et al.,* 2001). Since Object Oriented concepts and model checking techniques have matured, it is becoming possible to establish a design approach based on model driven engineering. The present approach is based on the composition and transformation of models to make checking and reliable implementation possible (figure1).

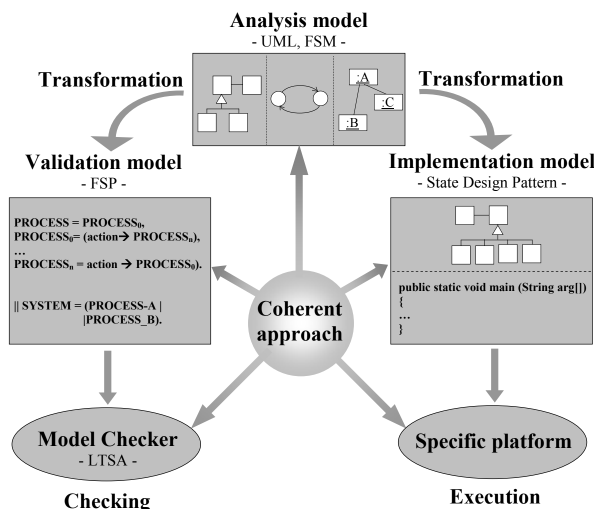

Figure 1: Conceptual representation of the approach proposed

The aim of the present paper is to propose a rigorous approach of design which simplifies, rationalizes, and validates the software design. To do so, this paper suggests to integrate, in a coherent manner, heterogeneous concepts and formalisms as Object Methods, Finite State Machines (FSM), Model Checking and Design Patterns. The present approach is based on:

- an object oriented analysis in which the structural aspect is divided into two conceptual levels: the resource objects and the behavior objects. This hierarchical decomposition provides a simple means to identify, then isolate – in distinct classes – the entities of a domain and the behaviors applied to these entities. The abstraction and organization of the behaviors obtained through this modeling process make them easier to understand and specify.
- a model of the dynamic aspect. In UML, Statecharts are commonly used to model the reactive behavior of entities. However, the organization obtained through structural modeling allows the use of simpler, more precise formalism. To do so, Finite State Machines present an appropriate notation to capture formally the behaviors associated with each behavioral class.
- a particular configuration of the system in order to obtain the specific behaviors required.
- a validation of the behavioral model. The FSMs are translated into process algebra called *Finite State Processes* (FSP) (Magee and Kramer, 1999). This leads to a *validation model* which can be exploited with the *Labeled Transition System Analyzer* (LTSA) model checking tool (Magee and Kramer, 1999).
- an implementation which agrees with the specifications. To this aim, this paper proposes a translation of the validated model based on the recurring use of Design Patterns (Gamma, *et al.,* 1995) to ensure reliable implementation.

This paper is divided into two main parts. The first part presents the proposed approach by describing the *analysis, validation* and *implementation models* respectively and the way to obtain a coherent translation between these models. The second section presents an example of a legged robot which illustrates the present approach.

2. THE PROPOSED APPROACH

2.1 Analysis model

The *analysis model* consists in finding a robust and adapted structure that fits the system to be modeled. This structure must be easy to handle and organized in such a way that a complex macroscopic behavior can be created. Since it was standardized by the *Object Management Group* (Object Management Group, 2003), the UML has become a standard for the description of software systems (Gomaa, 2000). So, in the present approach, the *analysis model* is based on an object oriented design formed of three aspects which represent respectively the *structure*, the *behavior* and the *configuration* of a software system. The structural aspect helps to organize the abstractions (classes) of the model. The behavioral one describes the behaviors associated with the structure and the configuration part establishes the links between the instances from the abstractions. The specification of these links is necessary for the description of a particular application. This separation helps the designer to focus his attention on a particular aspect to reduce the complexity of the conception.

Structural Aspects. Through different levels of abstraction, the specification of the structural aspects helps to better understand the organization and the interactions within a system. The UML class diagram is the representation which is best adapted to the structural organization of the abstractions. The structural aspect (figure 2.a) of the present models is based on a two-level conceptual model which allows a systematic separation between resources and their behaviors (Thiry, *et al.,* to appear). The classes describing the resources represent controlled entities of the system. The behavioral classes describe the behaviors which apply to the resources and control them in their state space. This separation helps to isolate and abstract the behaviors of the resources and so, to simplify their specification. This concept can be used in a recurrent manner for the design of complex systems, since a resource/behavior association can be considered as a new resource which is, itself, controlled by a higher level behavior (figure 2.b). So, each behavior becomes an object which has an internal state and which interacts with other behaviors. This hierarchical composition proposes a simple way to reuse and coordinate the entities within a system.

Behavioral aspects. Each behavioral class is associated with a Finite State Machine which specifies the dynamic aspect of each resource in the form of event/state sequences (figure 3). This choice has been motivated because this simple formalism which has formal semantics is usually used for the behavioral specification and can be easily integrated into a UML design. The independent evolution of the local behaviors describes the entire state-space of the system. In accordance with the supervision control architecture (Ramadge and Wonham, 1988), it is the aim of the higher level behaviors to coordinate local behaviors to obtain the global behavior of the system. This coordination is possible by a mechanism of sending/receiving events.

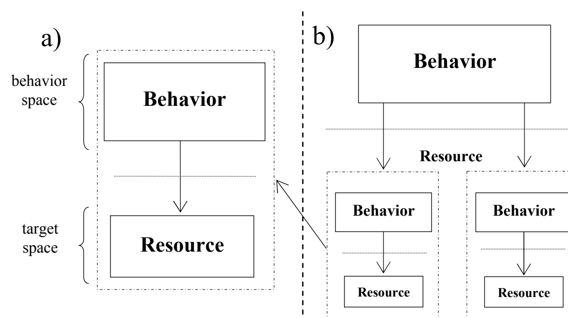

Figure 2: a) Structural aspect at two conceptual levels, b) Generalization of the concept

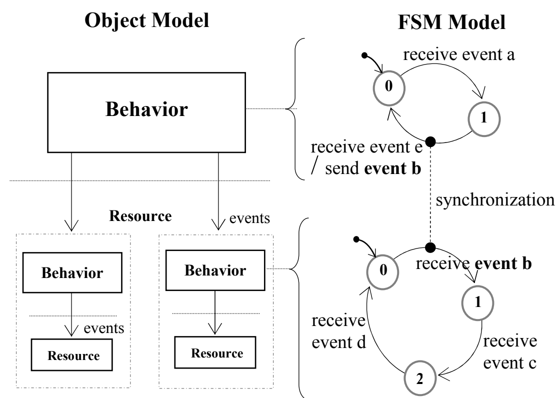

Figure 3: FSM associated with behavioral classes and hierarchy of behavior

So, local behaviors notify the global behavior of their change of state and so, according to the events received, the global behavior controls local behaviors in their state-space allowed. To make sure the global behavior meets the requirements, the behaviors which are isolated in the behavioral classes must be checked then implemented in accordance with the validation (section 2.2 and 2.3).

Configuration aspects. The dynamic and structural aspects which have been described so far, propose a family of potential configurations of a system in terms of behavior, class and interaction. However, software design usually requires the use of many instances of the same abstraction. Consequently, to describe a system, it is necessary to know the topology of these instances. In UML, this information is specified with an object diagram, so, this representation refines the structural aspects by specifying the links between the instances and refines the behavioral aspects by specifying the particular messages which are exchanged through these links (concept of synchronization). Thus specified, the configuration helps to understand complex systems and allows the design of a particular application whose behavior must meet the requirements.

The system is now fully specified and can be translated into the *validation model* to be checked and translated once again into an *implementation model* to be implemented in conformity with the validation. As will be seen in the next sections, FSMs find an equivalence with Design Patterns and with formal languages used for the implementation and validation. Consequently, their use makes the translation between the different models easier and so, simplifies the proposed approach.

2.2 Validation Model

The aim of all validation tools is to make software design reliable and to ensure designers that their specifications actually correspond to the requirements (Bérard, *et al.,* 2001). Among the checking methods, two major categories can be distinguished: simulation and model checking. These methods are not competitive but complementary. It is sensible to associate them within UML design, so as to bring an effective answer to the numerous checking problems. However, model checking methods require the use of formal methods which provide a mathematical context for the rigorous description of some aspects of software systems. In the present approach, *the validation model* will be expressed using a process algebra notation called *Finite State Processes* or *FSP* (Magee and Kramer, 1999) based on the semantics of the *Labeled Transition System (LTS)*. This formalism which is commonly used in the field of checking provides a clear and non ambiguous means to describe and analyze most aspects of finite state process systems (Arnold, 1994). It allows the use of the *LTSA* model checker (Magee and Kramer, 1999) in which a system is structured by a set of elementary components whose behavior is described in FSP. The present approach proposes to collect the behaviors specified with FSMs in the *analysis model,* then to translate them into FSP. The FSM formalism is also in accordance with the LTS's semantics; consequently they immediately find a correspondence with FSP (table 1).

Table 1: Mapping from the *analysis model* concepts to the *validation model*

Analysis model	***Validation model***
FSM state	local process *P=(a → P).*
FSM event	Action prefix *a →*
classes	processes
instances	process labeling *instance_name : type_name*
configuration	parallel composition *instance_1* || *instance_2*
synchronization	relabeling operator: a / b

The global behavior is obtained from all the instances of these elementary components and all their interactions within a particular configuration. These are executed concurrently and synchronized using the FSP *relabeling* operator to specify the set of valid states and the set of valid transitions of the software. It is this global behavior which is checked by LTSA.

2.3 Implementation Model

The formal specification techniques and the use of model checking tools do not prevent model mismatch during the development cycle; this is particularly true during the passage to the *implementation model*. The *Design Patterns* (Gamma, *et al.,* 1995) representing generic implementation models have been proposed to bring an explicit, proven solution to some recurring design problems. They reduce the development effort and increase software quality. Among these design patterns, the *State Design Pattern* gives an elegant solution which is commonly used for the implementation of Finite State Machines. The *State Design Pattern* in figure 4 specify and modify the behavior of an object when its state changes.

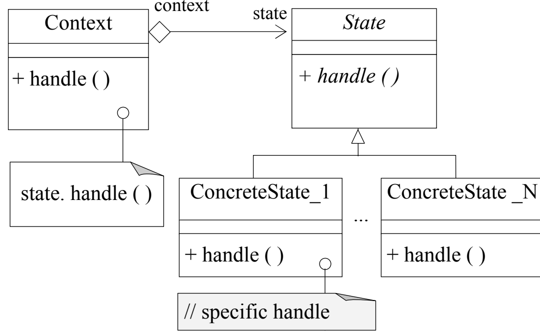

Figure 4: State Design Pattern

In the present *analysis model*, the behavior of each component is described as a Finite State Machine; it is therefore pertinent to translate these FSMs into an implementation which corresponds to the *State Design Pattern.* The *Context* class has a set of methods which represents the set of the FSM's actions. It manages the call of these methods to change the current state and delegates to each *ConcreteState* subclass the implementation of the task associated with the corresponding state. So, the passage from a *ConcreteState* class to another *ConcreteState* class evokes the change of state of a *Context* component. Its similarity with FSM helps to explicitly represent the concepts of states and transitions. So, it allows an implementation which is in accordance with the behaviors specified in the *analysis model* and checked in the *validation model.* This Design Pattern keeps the coherence of the approach in this last development phase. So, a valid implementation on specific platforms can be considered.

2.4 *Synthesis of the approach proposed*

In the present approach, the object oriented method is abstract enough to capture and integrate, within models, some heterogeneous formalisms which are usually used. To reduce the gap between models, each of them must respect the semantics of LTS. So, to ensure reliable software design, the present approach proposes - from *an analysis model* - to obtain a *validation model* specified with FSP and an *implementation model* based on Design Patterns. To illustrate this approach, the next section presents the example of a legged robot.

3. LEGGED ROBOT

3.1 *Presentation*

The system (figure 5.a) used to illustrate the present approach is an omnidirectional hexapod robot (Thirion and Thiry, 2002). This mobile platform requires efficient and appropriate control architecture for the integration of a number of coordinated functions. This system is the source of numerous problems concerning concurrence, synchronization, or decentralized control. Only the locomotion function will be considered here.

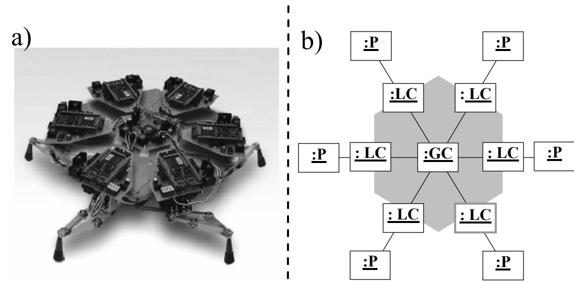

Figure 5 a) Mobile platform, b) Configuration

A leg moves in a cyclic way between two positions *aep* (anterior extreme position) and *pep* (posterior extreme position) (figure 6.c). A leg is in retraction when it rests on the ground and pushes the platform forward. It is in protraction when it resumes its *aep*. To ensure robustness and flexibility in locomotion, the control software must satisfy a set of progress and safety rules. According to the progress rules, all the legs must continue to move, whatever the possible execution traces of the system. According to the safety rules, all the legs must not be raised at the same time. So, the robot's control software is representative of a class of software systems which must be validated to avoid any problems in their exploitation.

3.2 *Analysis model*

Structure of the system in two conceptual levels. The control architecture is based on decentralized control: each local behavior obtained with a local controller (LC) is applied to a leg (L) and a global controller (GC) coordinates six local behaviors (figure 5.b).

Leg behavior. Figure 6.b shows the discrete behavior of a leg equipped with its local controller (figure 6.a). The beginning of the walking cycle is triggered by the occurrence of the *start* action. Each local controller is autonomous and their parallel execution describes the entire state-space of the system. However, to ensure reliable locomotion, only few states are allowed. It is the aim of the *global controller* to control the local behavior in the state-space allowed.

Global behavior. The global controller supervises (Ramadge and Wonham, 1988) each local controller by allowing (or not) its walking cycle (here, only the *start* action is controllable). It coordinates the legs and keeps the platform stable.

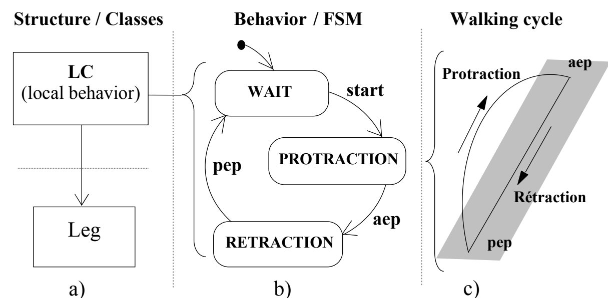

Figure 6: a) Architecture at two conceptual levels, b) Leg behavior and c) Walking cycle

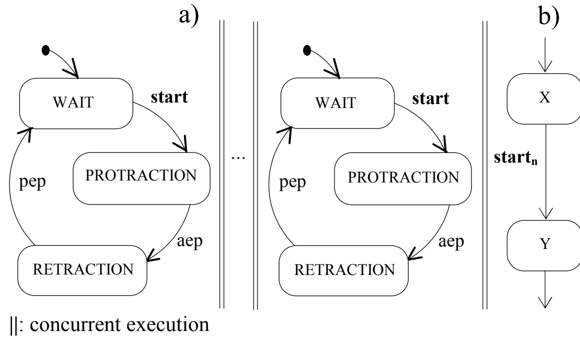

Figure 7: Parallel execution of a) the LCs and b) the GC

The global controller is not the goal of this example, so its specification (figure 7.b) will not be detailed here. It is the synchronization (sharing) of the *start* actions and the concurrent execution of all the local controllers (figure 7.a) and the global controller that provide the global behavior of the system. This global locomotion behavior must be validated to guarantee the above mentioned properties (section 3.1). Only the behavioral aspects which have been specified in this way will be validated, as will be seen in the next section.

3.3 Validation model

In the *validation model* proposed, attention will be given to progress properties which assert that "*something good eventually happens"* (Bérard, *et al.,* 2001). Indeed, adapted locomotion requires above all the recurrent motion of each leg. It must then be checked that each local controller will always be able to carry out its walking cycle. In agreement with the present approach, this property will be validated by the LTSA analyzer.

Specification of the behavioral model. The first step consists in specifying the behavior of the software components. In this model, the behavior of each component is defined as a Labelled Transition System (LTS) and modeled using process algebra notation FSP. So, the LC and GC elements which have a behavior in the *analysis model* are specified in FSP. Figure 8 represents the FSP translation of the behavioral classes of the local controller (LC) class graphically described by its Finite State Machine in figure 6.b. The global behavior is obtained from all the instances of these elementary components (LC, GC) and all their interactions within a particular configuration (figure 5.b).

```
LC           = WAIT,
WAIT         = ( start -> PROTRACTION ),
PROTRACTION  = ( aep   -> RETRACTION  ),
RETRACTION   = ( pep   -> WAIT        ).
```

Figure 8: Behavior description of an LC component in FSP

```
|| ROBOT = (lc1:LC || lc2:LC || ... || gc:GC)
/{
gc.start1 / lc1.start,
gc.start2 / lc2.start,
...
}.
```

Figure 9: Parallel composition and synchronization in FSP

In FSP, a process labeling (lc_i*:LC*) provides multiple instances of elementary components which are in accordance with the instances of the behavioral classes of the present *analysis model*. A set of six local controller processes (lc_i) is thus created, in which the labels of the actions (*start, aep, pep*) are prefixed with the label of the particular local controller (*lc1.aep, lc2.aep,...).* The global ROBOT behavior is expressed as a parallel composition (||) of the local (lc_i) and the global (*gc)* controllers. These are executed concurrently and synchronized on the *start* action (figure 9) using the FSP relabeling operator (/). This ROBOT behavior is validated by the LTSA model checker

Specification of the properties. In LTSA, the progress properties are expressed with the *progress* key word. The progress property previously stated (section 3.1) consists in checking the occurrence of the *start* action for each *local controller* and their infinitely repeated execution (figure 10).

Analyze of the model. The LTSA tool allows an interactive simulation of the different possible execution scenarios of the model specified. This exploration allows the user to improve his confidence in the coherence between the expected behaviors and the models which describe them. This first non exhaustive type of validation can be complemented by a search for property violations. If properties are violated by the model, the analyzer produces the sequence of actions that leads to the violations. The designer can then modify his model according to the results obtained.

3.4 Implementation model

The *State Design Pattern* proposes a simple way to implement the FSM. This implementation explicitly preserves the LTS concepts which are described in terms of states, actions and transitions. The implementation of the *State Design Pattern* for the behavior of a leg and of its local controller is shown in figure 11. The *LC* class can be in the state: *Wait*, *Protraction* or *Retraction* (figure 6.b) according to the current state and the occurrence of the event *aep*, *pep*, or *start*.

```
progress Cycle_lc1 = {lc1. start}.
...
progress Cycle_lc6 = {lc6. start}.
```

Figure 10: Progress properties in FSP

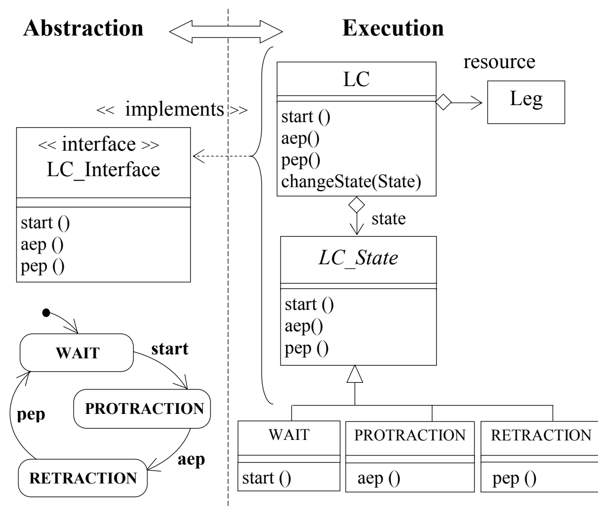

Figure 11: Implementation of a LC using the *State Design Pattern*

This implementation diagram consists of a number of elements including:

- the *LC_Interface* which defines all the possible actions of a component (alphabet of its Finite State Machine).
- the *Local Controller* class (*LC)* which exploits the abstraction of the leg as a resource by giving it a behavior described as a succession of states. It implements the *LC_Interface* and lets a local object called *state* perform the specific behaviors. This local object represents the current state of the *local controller* and changes according to the transitions inherent in its behavior (*aep, pep*, or *start*).
- the *LC_State* class which implements, in an abstract way, the behavioral *LC_Interface* and represents the parent class of all the states of a local behavior. Each particular state (*Wait, Protraction, Retraction)* implements the specific task associated with the state of the component. Each of these subclasses only defines the actions/transitions that are associated with them and the call of the corresponding methods causes the adaptation of the state of the local controller.

In this way, the *State Design Pattern* provides a safe means to produce the translation of an abstraction (the *analysis model)* into its implementation (the *implementation model*).

4. CONCLUSION AND PERSEPECTIVES

This paper has presented a rational method for software design by proposing a model based approach (*analysis, validation and implementation models*). It depends on an object oriented architecture with two conceptual levels and formal specifications based on Finite State Machines. From the information (structure, object, configuration and behavior) contained in the present *analysis model,* a *validation model* is obtained which fits the specified behavior. An *implementation model* adapted to this specification is obtained using the *State Design Pattern*, while conforming to the validation performed previously. Each model corresponds to the semantics of Finite State Machines which reduces the gaps between the different models. The present approach thus allows the coherent transition between heterogeneous models, ensuring the rational integration of the different phases in software development. The current work will be followed by the precise definition of transformation models aiming at a systematic or even automatic translation between the different models proposed. This automated model transformation will make the development process easier and more reliable.

REFERENCE

Apvrille, L., P. de Saqui-Sannes, C.Lohr, P. Sénac and J.P. Courtiat (2001). A new UML Profile for Real-time System Formal Design and Validation, *Proceedings of UML'2001*, Toronto, Canada, 287-301.

Arnold, A. (1994). *Finite Transition System*, Prentice Hall, Prentice Hall.

Bérard, B., M. Bidoit, A. Finkele, F. Laroussinie, A., Petitand P. Schnoebelen (2001). *Systems and Software Verification. Model Checking Techniques and Tools*. Springer, New York.

Booch, G. (1994). *Object-oriented Analysis and Design with Applications*, Second Edition, The Benjamin/Cummings Publishing Company Inc, Redwood City, California.

Gamma, E., R. Helm, R. Johnson and J. Vlissides (1995). *Design Patterns – Elements of Reusable Object Oriented Software*, Addison Wesley, Reading, Massachusetts.

Gomaa, H. (2000). *Designing Concurrent, Distributed and Real Time Application with UML*. Addison Wesley, Reading, Massachusetts.

Magee, J. and J. Kramer (1999). *Concurrency. State Models & Java Programs*. John Wiley & Sons, Chichester, UK.

Mikk, E., Y. Lakhnech, M. Siegel and G.J. Holzmann (1998). Implementing Statecharts in PROMELA/SPIN, *Proceedings of WIFT'98,* Boca Raton, FL, USA*,* 90-101.

Object Management Group, (2003). OMG, Unified Modeling Language Specification, Version 1.5, http://www.omg.org/docs/formal/03-03-01.pdf

Ramadge, P.J. and W. Wonham (1988). The control of Discrete Event Systems. *Proceeding of the IEEE*, **77**, 81-98.

Sanz, R., C. Pfister, W. Schaufelberger and A. De Atonio (2001). Software for Complex Controllers In: *Control Of Complex Systems* (Karl Astrom, P. Albertos, M. Blanke, A. Isidori, W. Schaufelberger, R. Sanz, Ed.). pp.143-164. Springer-Verlag, London.

Thirion, B. and L. Thiry (2002). Concurrent programming for the Control of Hexapode Walking, *ACM Ada letters*, **21**, 12-36.

Thiry, L., J.M. Perronne and B. Thirion (to appear). Patterns for Behavior Modeling and Integration, *Computer in Industry*, Elseier, Amsterdam.