Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Mathematics and Computers in Simulation 70 (2006) 314–329
MATHEMATICS AND COMPUTERS IN SIMULATION
www.elsevier.com/locate/matcom

Architectural concepts and Design Patterns for behavior modeling and integration

Jean-Marc Perronne *, Laurent Thiry, Bernard Thirion

MIPS, Université de Haute Alsace, 12 rue des frères Lumière, 68093 Mulhouse, France

Available online 20 December 2005

Abstract

The design of the control software for complex systems is a difficult task. It requires the modeling, the simulation, the integration and the adaptation of a multitude of interconnected entities and behaviors. To tackle this complexity, the approach proposed consists in combining architectural concepts, Design Patterns and object-oriented modeling with unified modeling language (UML). In this context, the present paper describes a modeling framework to take greater advantage of these concepts and to design flexible, intelligible control software. It proposes to objectify the behaviors, which leads to a two-level architecture based on three concepts: resources software images of the controlled system-behaviors applied to these resources, and meta-behaviors, i.e. means for behavior integration and adaptation. Two Design Patterns are proposed to describe how to specify behaviors and define the means to combine and adapt them. The first pattern, Polymorphic Behavior, provides the means to define new behaviors for a system and to plug them dynamically. The second one, Structured Behavior, provides the means to use finite state machines for behavior switching. The originality of the framework is that it defines concepts, a UML-based notation and heuristics which specifies how to apply these concepts. To illustrate the elements mentioned, this paper uses the control software of a walking robot as a running example.
© 2005 IMACS. Published by Elsevier B.V. All rights reserved.

Keywords: Software architecture; Object-oriented modeling; Control software; Design Patterns; Complex behaviors

1. Introduction

The design of the control software for complex systems is a difficult task [29]. In particular, it requires means – i.e. concepts, notations and guides – for the integration and adaptation of a number of local behaviors within the framework of global control [36]. To tackle this complexity, one of the current approaches takes advantage of the know-how acquired from object-oriented software. In this context, the present paper proposes a modeling framework which explains how to capitalize this know-how in order to find a new way to design complex software systems which are controllers. The basic concept proposed by this paper is that of behavioral objects, which consists in reifying the behaviors of a subsystem. This founding principle opens an important field of investigation of complex systems. In particular, it helps to model all the elements considered (subsystems, control laws and interactions) in a uniform way with objects. The well-known principles of the object-oriented approach – classification, composition and delegation – can then be applied to the behavioral aspects. The notion of behavioral objects leads to an analysis guided by a two-level architecture that sets up three kinds of entities: resources, behaviors and meta-behaviors. These two levels

* Corresponding author. Tel.: +33 3 89 33 69 67; fax: +33 3 89 42 32 82.
E-mail address: jm.perronne@uha.fr (J.-M. Perronne).

0378-4754/$32.00 © 2005 IMACS. Published by Elsevier B.V. All rights reserved.
doi:10.1016/j.matcom.2005.11.004

must not be mistaken for the traditional notion of hierarchy. The first conceptual level includes entities which model resources, i.e. software images of the physical components. The resources help to model the structure of the controlled system and to specify the available services to make this structure evolve. The second conceptual level includes entities which model behaviors and allow the control of the previous elements. A behavioral object can then be considered as a resource for behavioral objects of a higher order. These behavioral objects, called meta-behaviors, help to integrate, adapt and coordinate other behaviors; they represent the third concept of the architecture. The heuristics associated with the present architecture matches a modeling step with each of the above-mentioned concept. The first step consists in modeling the controlled system with the different objects it is composed of and their relations. The second step determines the behaviors and the local laws which apply to each of the entities found. The last step uses meta-behaviors to coordinate the specified behaviors until the desired global control strategy is obtained.

The present paper is divided into three parts. The first part describes the main problems of modeling controllers which are particular software system. It explains how the complexity of the controlled systems is also to be found in the control software and presents the current ways to approach this complexity. The second part describes the modeling framework proposed. This framework includes concepts, notation and heuristics used to reduce the modeling efforts by describing the software components necessary for the global control of a complex system, the way to represent them and to organize them. The third part shows how the behaviors can be synthesized from Design Patterns adapted to control software. The control of a hexapod robot will be the example used throughout the whole study.

2. Software control of complex systems

2.1. Software and control

The control field includes the necessary know-how for the synthesizes of an algorithm dedicated to the control of a particular subsystem. For example, Astrom and Wittenmark [4] explain: (1) how to synthesize a control law with optimality and robustness constraints and (2) how to implement this law with an algorithm. So far, however, the control field has no general framework which would explain how to integrate the multitude of controllers necessary for the global control of a complex system, into a single software, in a flexible way. So, the software control of complex system leads to a software system which is also complex. To tackle this complexity, Van Bremen et al. [38] suggest to take advantage of the concepts from the field of multi-agent systems. Each controller is modeled by an agent which is a kind of active object performing a control law. The global behavior is then modeled as a society of agents which collaborate or are coordinated by agents of a higher level. Van Bremen and de Vries [37] present an implementation of these concepts for servo-controlled room temperature.

The present paper follows a similar approach. It shows how to take advantage of the object-oriented concepts to allow the easier software synthesis for complex control systems and it uses the example of the control of a legged robot [33] to illustrate the concepts.

This system, shown in Fig. 1, consists of a multitude of interdependent variables or entities which must be organized. To control it, it is necessary to integrate several types of controllers which perform each a particular behavior. In the present case, the control software must integrate local controllers to servo-control each leg and a global supervisor to synchronize the local motions and to set the walking speed. Each local controller can be decomposed into three simpler controllers: a retraction controller which allows the platform to move, a protraction controller which determines where and how to reposition a leg and a controller which coordinates the previous two controllers. The proposed modeling framework helps to tackle the complexity of such a system.

2.2. Object-oriented approach to control

One of the current ways to master the increasing complexity of control software consists in reusing concepts issued from software engineering. Sanz et al. [29] present the advantages and difficulties of such an approach. They explain that software engineering contains the necessary concepts to tackle the complexity and that using them helps to reduce the design efforts. However, their application to the control field requires new knowledge [18].

The elements from the software field which prove most promising in the control field are the object-oriented concepts – with the UML language [13] – and the architectural elements – with the Design Patterns [16]. Booch [6] and

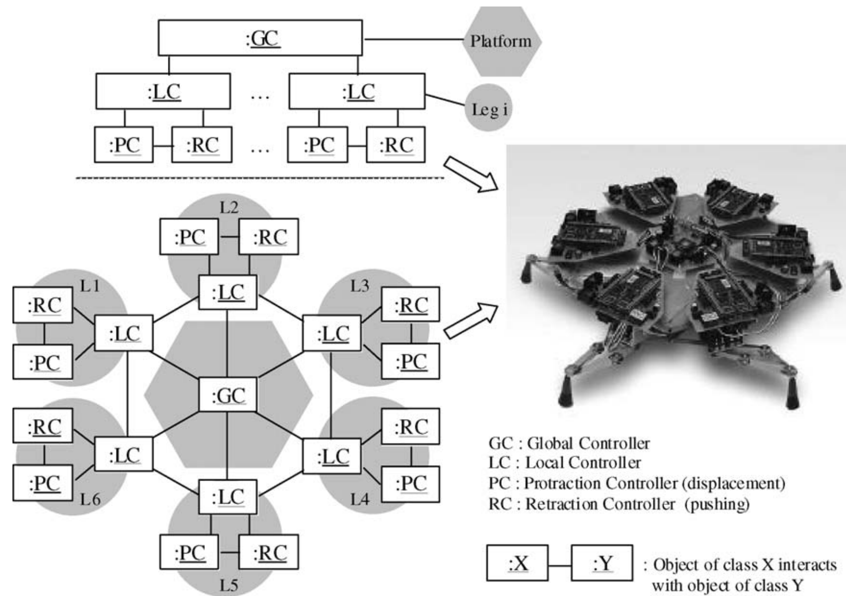

Fig. 1. Hexapod robot.

Rumbaugh et al. [27] present the fundamental principles of the object-oriented approach. They are relatively simple to understand:

- *Objects and messages*. Any entity is modelled by an object. One object collaborates with others through messages which correspond to requests or service requests. This point of view which is relatively different from the one used to study dynamic systems opens new prospects in terms of modeling. Fishwick [12] has shown the benefit from this change in viewpoints: greater intelligibility and adaptability of the models, in particular.
- *Classification and polymorphism*. Each object is associated with a class which specifies the properties of the object and the services it proposes. The classes are organized in the form of hierarchies within which specific classes inherit and refine the properties from parent classes. This helps to increase the abstraction level and the reusability of the models. The inheritance relation makes adaptation easier by allowing the replacement of an instance of the class C by any class which inherits from C. This leads to the principle of polymorphism.
- *Collaboration and delegation*. To perform a complex process, an object collaborates with others; by sending messages, it delegates part of its behavior to other objects.

The application of the above mentioned concepts to the control field has given a number of encouraging results. Yacoub and Ammar [39] propose an object-oriented model of servo-controlled systems. They then demonstrate that object-oriented concepts offer new mechanisms which are relatively different from the ones currently used in terms of adaptability. For example, they exploit polymorphism to adapt the control strategy; so there is no longer a parameter adaptation (classic case) but an algorithm adaptation. Dagermo and Knutsson [10] apply these object-oriented concepts to define an architecture dedicated to the control of vessels. They explain how object networks help to carry out this task. The notion of object is also used to integrate functional aspects with non-functional aspects. Functional aspects are defined by objects of the controller type. Non-functional aspects are characterized by other objects such as proxies, for example, for distributed control. Thiry and Thirion [35] show that object-oriented concepts lead to models which are relatively different from the one currently used and so gain in intelligibility and flexibility. They also make it clear that generic frameworks, which explain how to apply object-oriented concepts more rigorously, remain to be defined.

2.3. *Unified modeling language (UML)*

The unified modeling language has been proposed by the Object Management Group as a standard for the representation of systems in general. UML 2.0 includes 13 types of diagrams to represent, through different viewpoints,

the different aspects necessary for the comprehension or development of a system [13]. In particular, the collaboration diagram allows the representation of a given configuration by specifying the local interactions between the different objects: subsystems, controllers, users, sensors or actuators, etc. The sequence diagram is more interesting as it helps to model the temporal aspects in the interactions. The class diagram helps to specify, then organize, the types of objects at different abstraction levels in a system. Statecharts are state–transition diagrams used to describe the behavior of a class. These four types of diagram allow the representation of the structural and behavioral aspects. They also help to approach the complexity from different points of view [31]. The advantage of UML diagrams is that they can be understood by a large community and can be used within a large domain of application. In the control field, UML is mainly used to organize real-time aspects; extensions have been proposed with the UML-RT profile [21]. The use and advantages of UML at this level have been detailed by Douglas [11] whose uses UML to model control software architectures. In particular, he shows how to model, then implement the control strategy by considering the constraints related to the field: response time and shared resources, for example. The present paper describes a complementary use of UML. It shows how to take advantage of the main diagrams mentioned above to understand and organize the elements necessary for control: resources, behaviors, means for integration or adaptation, etc.

2.4. Design Patterns

The concept of Design Pattern was proposed by architect Alexander [3] in his book "*The Timeless Way of Building*". Later, the concept reappeared as an important concept in the field of software architectures to design applications by reusing generic design schemas established from successful and effective object-oriented solutions. Patterns were proposed after observing that a great number of software applications were based on the same principles and that their knowledge allowed design efforts to be reduced considerably. Today, the main patterns are described in a catalog [16], which presents 23 fundamental models to design systems that are more intelligible, more flexible and, above all, more generic. Buschmann et al. [7] proposes another important catalog of architectural patterns. These catalogs describe the styles of organization and interaction at a higher level of abstraction—by presenting layered architectures, for example.

The description of a Design Pattern includes:

- A name, to identify the pattern and to extend the vocabulary used to describe architecture.
- An intention, to summarize what the Design Pattern does and the conditions of use.
- A context (or motivation), to illustrate how the components of the pattern respond to the design problem.
- A detailed description of the pattern, with a graphical representation of its structure, the specification of the different components, directions for use, particularities of the pattern.
- A range of applicability, to present the situations in which the Design Pattern can be applied.
- An example of use.

Today, the concept of Design Pattern finds its place in other fields with, among others, management-, information- and control systems [8,14]. This convergence of the know-how from software engineering and that from system control engineering is possible as the models considered are described in an abstract way: each element is modeled by an object (processes, controllers, interactions, etc.).

Interesting examples of Design Patterns for the design of more intelligible and more flexible control architectures are given by:

- Sanz [28], who proposes a catalog of patterns to give more intelligence to control systems. Here, the patterns are used for the description of new mechanisms to adapt and integrate a set of controllers.
- Pont and Banner [24], who present a number of patterns for the design of embedded systems. These patterns explain how to study these systems more simply.
- Gaertner and Thirion [15], who propose Design Patterns for the control of discrete event systems using the Grafcet formalism. The patterns are used here to describe how to specify, then implement well-known models of the control field into object systems.
- Aarsten et al. [1], who present patterns for concurrent and distributed systems with applications in Computer Integrated Manufacturing (CIM). These patterns describe how to take account of non-functional aspects. They help

to link the control strategies with the particularities of the support architecture—i.e. several tasks distributed in a network.
- Selic [30], who proposes a pattern which explains how to specify, then create hierarchic control architectures that are more flexible and more robust.

The models proposed in this paper extend these works and can be considered as architectural patterns. They explain how to best use object concepts to model a controlled system (cf. two-level architecture), they describe the available means to combine (cf. combinators) and adapt (cf. adapters) a number of behaviors.

2.5. Architectural approach

The architecture of a system has been defined as a set of elements which are organized according to a given form and which meet a given motivation [23]. The advantage of the architectural approach has been known for a long time: from different points of view and through different abstraction levels, it helps to better understand the organization and interactions within a system. The notion of architecture is important in the field of systems in general, as it must help to tackle and master complexity [17]. As for control, the complexity generally lies in the multitude of local controllers which must be combined and adapted to give the controlled system the desired global behavior. There are some interesting examples of software architectures dedicated to control. Perronne and Hassenforder [22], present a model of software architecture to manage the different data flows necessary for control. This architecture helps, among other things, to abstract the hardware elements for greater intelligibility and to reconfigure the control strategies dynamically for greater flexibility. Zeigler et al. [40], propose an architecture based on the integration of models for the design of more autonomous systems. The behavior is specified by a set of models representing the different tasks to carry out. This idea of using models to study or design complex systems is the basis of Van Breemen's work [38]. It has the advantage of separating the different aspects of a system with the help of models of local controllers, models for the integration of these controllers and adaptation models; Narendra [20], Coste-Manière and Simmons [9] or Alami et al. [2] propose architecture models for the control of mobile robots.

This part has shown that the object-oriented approach, associated with UML notation and the notion of architecture helps to handle the complexity of software systems in general. That is why the approach is interesting in the particular case of control software where a multitude of interacting local controllers must be integrated and adapted so as to give the controlled system the desired behavior. The results are extended, by the present paper, to a unified modeling framework based on a given style of architecture and on heuristics.

3. Architectural concepts for behavior modeling

The proposed modeling framework can be described from three concepts associated with a two-level architecture style and three modeling steps. These concepts representing fine grain software abstractions are: (a) the resources, corresponding to the software images of the physical components; (b) the behavioral objects whose role is to control the resources in their state space; (c) the higher order behaviors (or meta-behaviors) whose role is to control other behavioral objects so as to combine them or adapt them, for example. The elements to consider – structures, behaviors and interactions – will be modeled by objects; so, the same notation will be used to represent the static and dynamic aspects.

3.1. Founding principles

In the field of control, a controller is defined to create and attach the desired behavior to a system. The reification of behaviors [5], also called objectification, helps to consider a behavior as an object; as such, it can be architectured. A behavior can be defined as a series of transformations applied to the state of a system; the state corresponds to an observable configuration of the system. With the object-oriented approach, an entity is modeled by attributes and methods. The attributes represent the state of the system and the methods correspond to the different transformations to pass from one state into another. The reification of behaviors then consists in capturing – in the form of an object – the series of calls for the methods so that the controlled system reaches the desired state. This reification has numerous advantages. As the behavior is an object, it can also receive messages and be controlled by objects of a higher level;

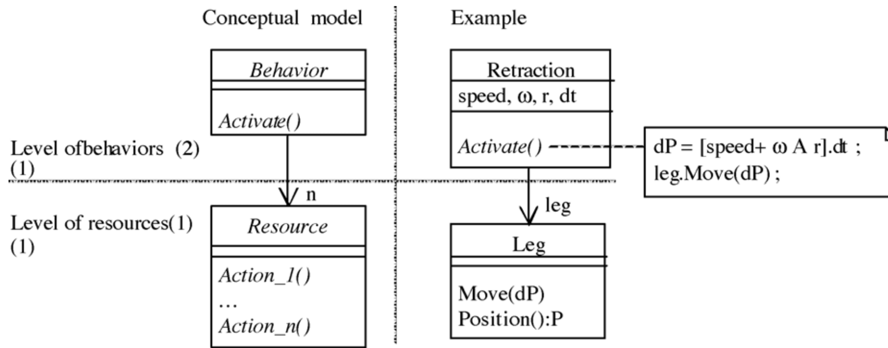

Fig. 2. Behavior-resource conceptual model.

this property is used by meta-behaviors to integrate or adapt different behaviors. The behaviors can be classified as hierarchies of increasing abstraction. Polymorphism will help to dynamically adapt the behavior of a system. Finally, another interesting property is the ability to represent static and dynamic elements on the same diagram.

3.2. Behavioral objects

The reification of behaviors leads to the concept of behavioral objects which represents the most important element in the proposed modeling framework. The term "behavioral object" was introduced to allow the easier modeling and synthesis of complex dynamic systems [34]. Indeed, the dominant element in the field of control is the behavior. In order to model and integrate it in an object-oriented software, a natural choice is that this behavior becomes an object. This definition of behavioral objects leads to modeling at two conceptual levels of analysis (Fig. 2). The first conceptual level includes all the resources evolving in any state space; resources typically model physical entities. The second conceptual level corresponds to the behavioral objects whose role is to control the resources in their state space. In this schema, the laws, the evolution rules or the constraints are systematically separated from the objects governed by these laws. For instance, in Fig. 2, the retraction controller describes the control law to move a leg object which itself represents the organ as such.

3.3. Organization of behavioral objects

The advantage of the proposed concept is to allow the use of the founding principles of object-oriented modeling [6,26] for the description and *the organization of the behavior space*. The relations of classification and aggregation in particular prove important as they help to structure this space and allow the easier synthesis of flexible control software:

- Inheritance allows classification of various behavioral objects; it will help to organize behavior categories by abstraction levels. It will also be used to define Polymorphic Behaviors that can change dynamically.
- Aggregation helps to obtain behavior diversity and to decompose complex behaviors into simpler behaviors.

These two relations allow better mastery of complexity as they provide the means to obtain diversity by successive refinements (inheritance) and by combination (aggregation). They also help to identify and organize the behaviors of a system so as to make them easier to understand or specify. Hence, a network of behavioral objects which provide a structure for the space of behaviors, as shown in Fig. 1.

Fig. 3 proposes an extract of the organization of the behaviors necessary for the control of the legged robot. It notably shows the four main behavior/controller classes used to control this complex system:

- the global controller sets and controls the global motion (GM) of the platform;
- the local controller controls the leg motion (LM) according to the retraction and protraction phases;
- the retraction controller controls the leg at rest and so contributes to the global motion;

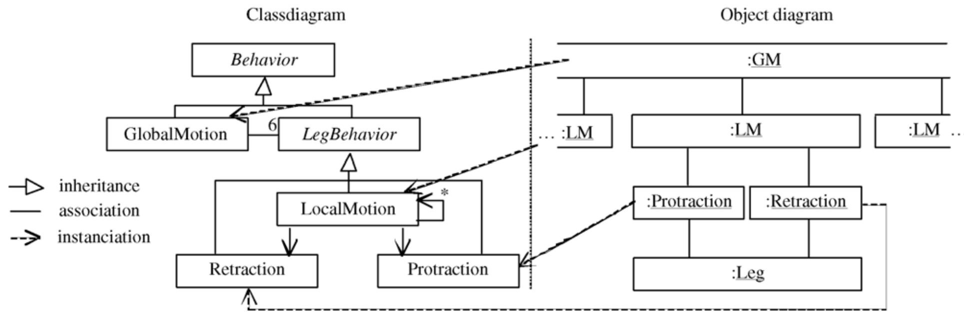

Fig. 3. Hierarchic organization of abstractions and compositions.

- the protraction controller brings the upheld leg into the position where it can contribute again to the motion of the platform.

The local behavior results from the aggregation of a local controller, a retraction controller and a protraction controller. The global behavior of the robot is then obtained through the combination of a global controller with six local controllers. The configuration used for global control is shown in the right-hand part of Fig. 3, while the left-hand part shows the hierarchy of the behaviors used.

3.4. Meta-behaviors

The previous part has explained how to organize local the behaviors that are necessary for the control of a complex system. This part now presents the composition model for the integration of these local behaviors into the context of more global control. It starts from the observation that a behavior is an object; in this sense, it can be considered as a resource and controlled by other behavioral objects (Fig. 4). This corresponds to the third concept of the reference architecture and to an additional modeling step: after modeling the physical components with resource objects and their dynamic with behavioral objects, the last step consists in modeling the means for the integration of these higher order behaviors (meta-behaviors). Behavioral objects are used to define the dynamic (i.e. control laws) which specifies how the internal state of a resource evolves within a given running mode; meta-behaviors are used to define how the behaviors themselves evolve.

A meta-behavior is a behavioral object which does not specify a particular control law but a general mechanism that can be used in a large range of applications in order to activate, inhibit, organize, combine or adapt behaviors. So two categories of meta-behaviors are currently identified: combinators and adapters.

3.4.1. Combinators

Combinators are particular behavioral objects in the sense that they can only be applied to behaviors. The most conventional combinators are given by sequential, parallel, repetitive or conditional behaviors, as shown in Figs. 5 and 6.

Fig. 5 proposes a family of combinators that can be used for the integration of behavioral objects. These fall into five categories:

- *Basic behaviors* which represent “behavior atoms” to be combined for more complex activities. In Fig. 5, retraction and protraction belong to this category. They represent control laws that are assumed to be indivisible. In general,

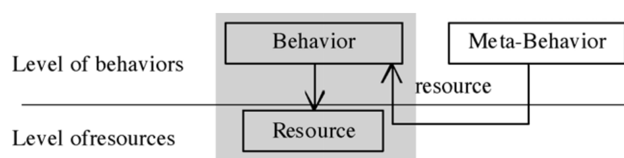

Fig. 4. Meta-behavior.

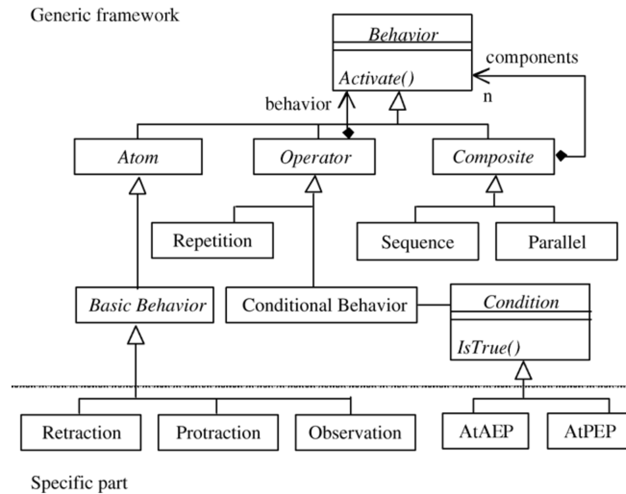

Fig. 5. Combinators for the integration of behavioral objects.

a basic behavior is a behavioral object which does not describe a combination and which is directly applied to a resource.

- *Repetition behaviors* are currently used to define cyclic activities. This is the case of controls which must run constantly. For example, the locomotion behavior consists in repeating two actions – protraction and retraction – successively.
- *Conditional behaviors* help to add an activation condition (or guard) to a behavior. For example, it must be possible to suspend the retraction behavior when the leg stretch degree reaches its maximum threshold.
- *Sequence behaviors* allow the simple combination of behaviors following the rule: "if a behavior is completed, then activate the next behavior". For example, the protraction behavior of a leg follows the retraction behavior (Fig. 10).
- *Parallel behaviors* allow several behaviors to be performed simultaneously. They play an important part in the control of complex systems where several activities must be carried out in parallel. They allow a modular design of control software. For example, the global motion of the platform is obtained by combining, in parallel, the motion of the six legs.

The configuration described in Fig. 6 represents the local behavior of a leg (Fig. 10) in the form of a behavior tree. Complex behaviors are obtained by combining the behaviors of the lower level. The basis of the hierarchy includes the simplest behaviors that are most specific to the target system considered.

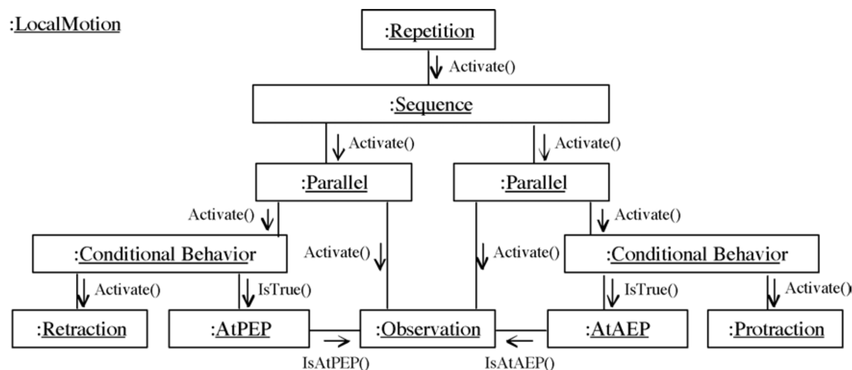

Fig. 6. Configuration of behavioral objects specifying the local motion.

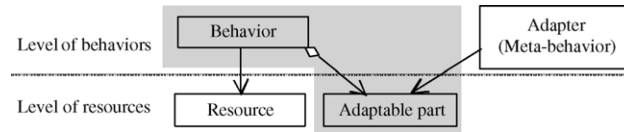

Fig. 7. Meta-behavior adapter.

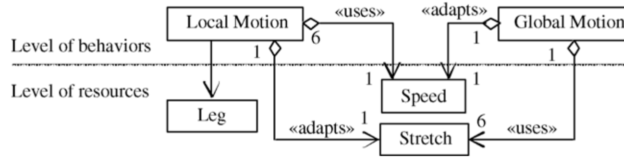

Fig. 8. Adaptation of the speed for the hexapod.

3.4.2. Adapters

Combinators – as behaviors of a higher order – are used to compose behavioral objects and to obtain new, more and more complex behaviors, always following the same schema (repetitions, conditional behaviors, composition, etc.). This concept can be generalized for the adaptation or reconfiguration of a behavioral object; that is why architecture extension is proposed. The model of behavioral objects must be refined so that a behavior can adapt/reconfigure another behavior. A behavioral object can be decomposed into two parts: a constant part and a variable or adaptable part. This leads to the reification of the adaptable part of the behavior. The conceptual model in Fig. 4 can be declined and a "folding" process allows the easier definition of a behavior which exports an adaptable part used as a resource by an "adapter" meta-behavior (Fig. 7). It must be noted that, if need be, the adaptable part can be shared by several behaviors which would then benefit from any adaptation action.

A first illustration of this principle is the management of the robot's motion. A global controller (GlobalMotion) adapts the global speed of the platform according to the performance of the six local controllers (LocalMotion) (Fig. 8). The present case is in fact rather complex as it sets up a continuous process which adapts the performance of six hybrid processes.

- The global controller uses the stretch degree provided by the leg controllers to adapt the global speed of the platform. Here, speed is the adaptable part of the six local controllers.
- The six leg controllers use the global speed of the platform adapted by the global controller in order to calculate and apply the control parameters of their legs.

This example has illustrated the case where a process of a higher level adapts a parametric process at a lower level.

A second illustration of the adapter principle is the design of a controller (behavior) which controls a resource using a strategy (adaptable part) that describes a control algorithm. The strategy defines the actions to execute in a given situation; the controller then just applies these actions to the resource it controls and provides a configuration model. This principle may be applied to the robot so as to adapt its locomotion by changing the walking strategy. The robot would then be able to switch from a free walking gait to a rhythmic walking gait, for example.

4. Design Patterns for behavior modeling

The above proposed modeling framework has shown how to determine and organize – conceptually – the abstractions necessary for the modeling and design of a control software system. To create them, an implementation phase is necessary; it may take advantage of the Design Pattern principle so as to benefit from a proven means for the obtention of well-built software. According to this principle, an implementation support for the synthesis of behaviors with two Design Patterns will be proposed. It will not cover the whole architecture, but give indications for the integration and description of the behaviors.

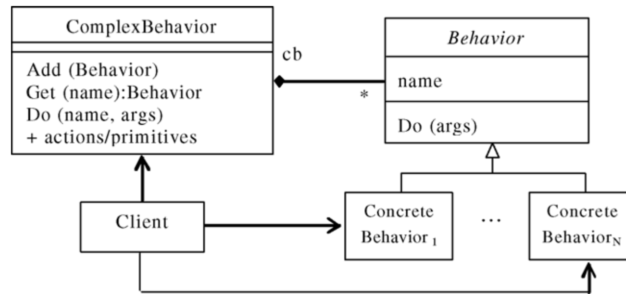

Fig. 9. Structure of the Polymorphic Behavior pattern.

The first Design Pattern, Polymorphic Behavior (PB), provides the means to define and to plug new basic behaviors into a system dynamically. The second Design Pattern, Structured Behavior (SB), allows the description and the discrete control of complex behaviors using finite state machines (FSMs).

4.1. Polymorphic Behavior pattern

To be reusable, patterns are generally described according to a typical schema ([16], Section 2.4) specifying the Design Pattern's intention, its context of use, a detailed description, etc. The two patterns presented here conform to this schema.

4.1.1. Intention

Polymorphic Behavior allows the definition, the integration and the execution of new behaviors for an object or a software component.

4.1.2. Context

The design of flexible and upgradeable systems is based on the capability of defining, attaching and using new behaviors or new functionalities dynamically. However, upgradeability becomes really interesting when there is no need to change the initial structure of the system. Such upgradeability will allow the addition and use of increasingly complex functionalities that will make the system efficient, more robust, more autonomous, etc.

4.1.3. Description

A behavior is represented as a dynamic element (i.e. an activity) linked to a complex behavior called ComplexBehavior as shown in Fig. 9. If the first element defines the interface for any dynamic behavior, then the ComplexBehavior is characterized basically by a generic method Do. Consequently, the ComplexBehavior may be integrated into a structure without specifying all the services and behaviors it proposes. Then, a client uses the services proposed by the ComplexBehavior to build new behaviors which will be added in a recursive way—so as to give it high-level functionalities.

The structure–behavior or ComplexBehavior–behavior relation is similar to the class–method relation of the object meta-model and is the basis of this pattern.

Structure: see Fig. 9.
Components:

ComplexBehavior
- proposes a set of primitive services (actions) which represent the basic components used to build other behaviors;
- defines specific methods to manage these behaviors (Add, Get and Do).

Behavior
- represents the interface for an executable behavior. The latter will be started by calling the Do method and can be parameterized.

ConcreteBehavior
- represents a concrete behavior to be integrated into the ComplexBehavior.

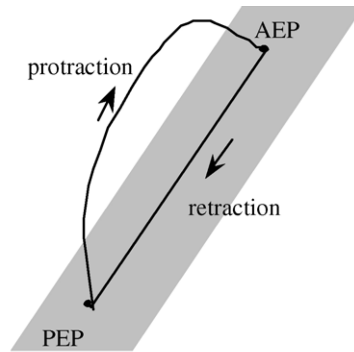

Fig. 10. Basic cycle of a leg.

Use:

The designer

- declares a set of new behaviors by subclassing behavior;
- specifies the Do method by using all the possible services proposed by ComplexBehavior, i.e. using basic actions or using other behaviors.

The client

- integrates new behaviors into the ComplexBehavior with the Add method;
- executes a behavior with the message "cb.Do (behaviorName, behaviorArgs)".

Particularities:

- behavior and structure can be considered separately. The ComplexBehavior can be integrated as a basic component into any structure; then, when a new need or behavior is required, a method is defined and attached to it;
- if the generic method "Do(name, args)" manages unknown names, i.e. behaviors that are not defined yet, default behaviors can be used. For instance, a composite behavior can be added to the ComplexBehavior and the (sub) behaviors used by this behavior can be added later;
- the system becomes dynamic, i.e. all components are ComplexBehavior instances, and consequently, the notion of type disappears and the system becomes more flexible;
- to use a behavior, the client must know its name and its parameters.

4.1.4. Applicability

This pattern provides the means: (1) to handle an entity, the ComplexBehavior, whose behavior is not yet entirely defined and (2) to define and plug dynamically various behaviors into this entity. The pattern proposed is an extension of the well-known Command Pattern [16]; this implies that all the specificities of this pattern can be used in SB. The Command Pattern is used by the Command Processor pattern proposed by Sommerland [32]; it is used to centralize and control the commands applied to a system. The specificity of the pattern presented is that it allows the integration of a reflexive level into an object-oriented application: it allows the dynamic modification of the behavior.

4.1.5. Example

The PB pattern helps to design systems with great flexibility: each control unit is represented as a behavior that is defined, modified, parameterized and integrated into the system independently of the others. For the hexapod, a hierarchy of behaviors is defined and attached to the platform and leg.

There are two kinds of behaviors: global behaviors applied to the platform and local behaviors applied to a leg. The control architecture has a global behavior called global motion (Fig. 11). To some extent, it is the interface or the view of the controlled system. The role of the client is to define the mission of the controlled system. According to a given speed and direction, at any moment, the global motion controller computes the adapted command which must be applied to the platform. From this single component, specific behaviors such as rotating or moving forward can be deduced.

The local leg behaviors allow platform stability and platform motion. The main behavior of a leg is defined by a repetition of two phases (Fig. 10): retraction and protraction. The retraction behavior uses the global speed to compute

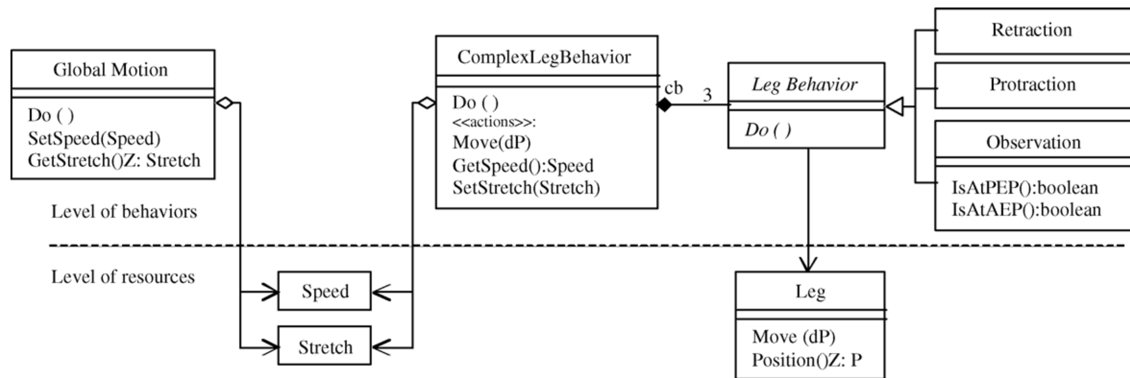

Fig. 11. Behaviors hierarchy for modeling leg behaviors.

the local speed and moves the extremity of the leg towards the posterior extreme position (PEP). The protraction behavior consists in moving the extremity of the leg to the anterior extreme position (AEP), at maximum speed and in conformance with the global motion.

The limit positions are computed using the working space, i.e. the space of all possible positions that a leg can reach. PEP is the position of the leg when it goes out of this space; AEP is the symmetrical image of PEP in this space. An observation behavior is used to compute theses two predicates. So, in order to set up the local leg behavior, three basic behaviors must be implemented: the protraction behavior, the retraction behavior and the observation behavior (Fig. 11).

The particularities of the PB pattern improve the resulting architecture:

- the separation between behaviors and ComplexBehaviors helps the design of a controller which can be applied either to a simulated system or to a concrete system;
- the behaviors are plugged and adapted dynamically (i.e. during runtime);
- considering structural Design Patterns (for example, composite or adapter), sequential behaviors or adaptable behaviors can also be set up; this highlights the extensibility aspect of the proposed pattern.

4.2. *Structured Behavior*

4.2.1. *Intention*

Structured Behavior provides the means to define, control and attach a finite state machine to a complex behavior. Satecharts are commonly used to model the reactive behavior of complex systems; in Yacoub and Ammar [39] a very complete Pattern Language of Statecharts is detailed. However, the organization obtained through structural modeling allows the use of a simple formalism such as FSM. Structured Behavior is a simple extension of the State Pattern [16] by integrating the concept of transition and by describing the evolution of the FSM explicitly.

4.2.2. *Context*

Finite state machines are simple but efficient tools to describe systems whose behavior is complex. Indeed, if they describe the evolution of the internal state of a system, they can also represent the behavior evolution of this system; in this case, the activation/deactivation of a state is attached to a start/stop of a behavior.

FSMs are currently used to represent complex behaviors with, for instance, real-time systems presented by Douglas [11], or Magee and Kramer [19]. But the known patterns which help to pass from the description to the implementation are either:

- *Incomplete*: The State Pattern, for example, does not explicitly represent the transitions between the states. Some extensions have been proposed by Ran [25].
- *Very complete*, *yet difficult to use*: For example, the Statechart model presented by Yacoub and Ammar [39] which is composed of a large number of classes.

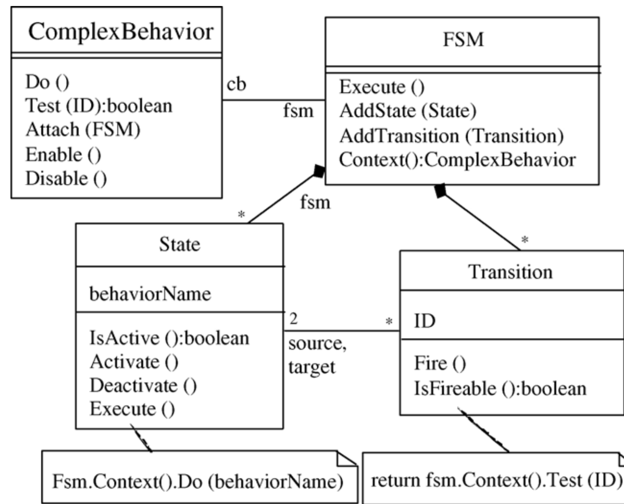

Fig. 12. The Structured Behavior pattern for FSM modeling.

4.2.3. Description

The pattern presented has four classes (Fig. 12):

- States are associated with behaviors. The state activation/deactivation will start/stop the corresponding behavior.
- Transitions between states characterize switchings between associated behaviors. The fire of a transition, which can be controlled by a guard, deactivates the source state and activates the target state.
- The FSM fires the fireable transitions, deactivates the source states and activates the target states.
- The ComplexBehavior defines the specific context of the FSM. This class provides the behaviors that can be performed and the information that can be used.

Structure: see Fig. 12.
Components:
ComplexBehavior
- models an entity whose behavior switching is abstracted using an FSM. This FSM is defined separately using states and transitions, and is integrated into ComplexBehavior with the Attach method;
- provides the logical conditions and behaviors used by the concrete transitions and the concrete states.

State
- abstracts an atom of behavior and can be activated/deactivated. The behavior is executed while the state is active.

Transition
- establishes an elementary relation between two states. When a transition is fireable, the source state is deactivated whereas the target state is activated, i.e. a behavior is started while another one is stopped.

FSM
- controls the global behavior, i.e. the activation of the various states and fires the transitions;
- provides the means to manage (add, remove, get, etc.) a set of states and transitions.

Use:
Designer
- declares a ComplexBehavior with: (1) a set of behaviors and (2) a set of conditions;
- defines the concrete states by creating an instance of, or by subclassing, state;
- defines the concrete transitions by creating an instance of, or by subclassing, transition.

The client
- defines an FSM with instances of concrete states and transitions;
- attaches this FSM to the ComplexBehavior.

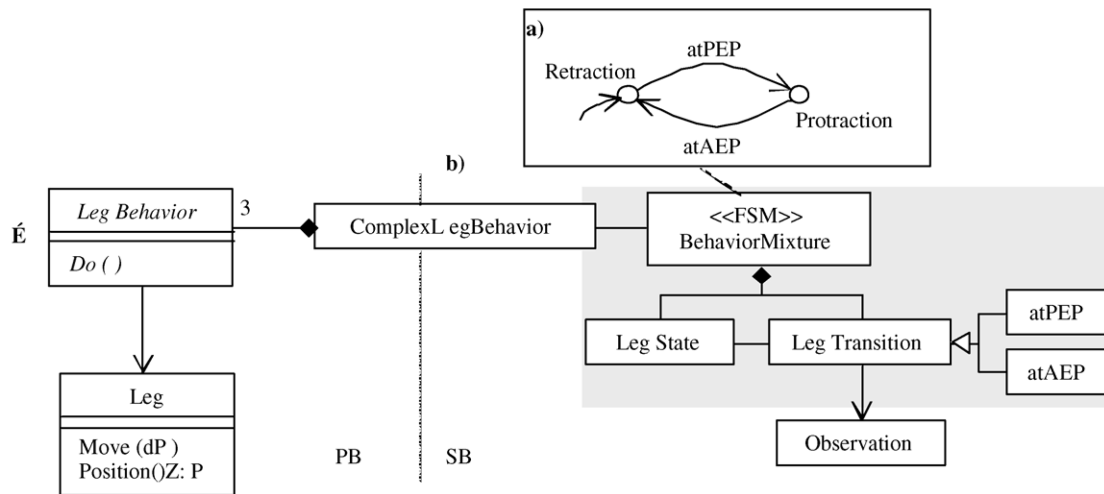

Fig. 13. (a and b) Integration of behaviors using PB and SB.

Particularities:

- the evolution of the FSM ((de-)activation of states and firing of transitions) is clearly defined whatever the structure of the FSM;
- the definition of new states, or transitions, is easy;
- the ComplexBehavior is abstract; a specific behavior mixture is performed by the FSM;
- like PB, the Structured Behavior pattern provides the means to define and modify dynamically complex behavior switching, using an FSM.

4.2.4. Applicability

Fig. 12 represents a simple, clear and easy to use meta-model for finite state machines. This model specifies the main components (state, transition, behavior, etc.) and describes the dynamic evolution of an FSM. The proposed structure is an extension of the State Pattern which is limited to the concept of state for which the designer has to implement the strategy for the evolution between the various states.

This pattern can easily be extended by adding: (1) composite states which execute an internal FSM when they are activated and (2) other elements such as disjunction and conjunction, for instance; see Gaertner and Thirion [15] for more details.

4.2.5. Example

To obtain the leg motion of the hexapod, it is necessary to compose the basic behaviors presented above. This task is carried out using the Structured Behavior pattern. The mixture of "behaviors" and FSMs will produce combined behaviors corresponding to a higher level behavior. Thus, the basic behaviors describe the actions necessary for walking and the FSM describes the switching of these behaviors. Therefore, the FSM describes, at a high level of abstraction, the global behavior of a component and the SB explains how to implement and execute it.

The BehaviorMixture (Fig. 13b) represents the fundamental behavior of a leg. It describes an FSM with two states that represent the two phases of the walking cycle (Fig. 13a). The observation behavior is used to update the conditions associated to each transition. The conditions are: atAEP which means that a leg reaches its AEP, atPEP which means that a leg reaches its PEP. This basic model of BehaviorMixture is then refined to add the synchronization rules and the constraints necessary to the stability of the platform. The use of the SB pattern makes this modification easy. Indeed, with the SB Design Pattern, state and transition are objects that can be handled, assembled, or even adapted. For instance, a condition between two states can, by subclassing the class transition, be suited to a more complex condition of transition. This ability is used to synchronize the local motions of the six legs.

To ensure the stability of the platform, two neighboring legs cannot be in protraction at the same time. Therefore, the state of a leg must be known, i.e. protraction or retraction. This information can be given by the BehaviorMixture

behavior. To take this information into account, the transitions of the FSM can be refined by subclassing atPEP and atAEP.

5. Conclusion

This paper has presented a modeling framework to define control software for complex systems using the example of a controlled hexapod robot to illustrate the notions considered. The modeling framework offers concepts, a notation and heuristics that allow the use of object-oriented concepts to "reduce" complexity and to synthesize intelligible, flexible control software. It is based on an architecture model where objects are organized according to two conceptual levels—one for resources and one for behaviors. This proposal reduces the apparent complexity of a system by separating the nature of the entities it is composed of from their dynamic. The object-oriented concepts then allow the identification and organization of the different behavior classes. More particularly, inheritance helps to organize behaviors by abstraction levels and aggregation helps to obtain complex behaviors by combining simpler ones. Finally, some behavioral objects can be considered as resources for behaviors of a higher order (meta-behaviors) so that they can be integrated, combined or adapted. The same notation may then be used to represent the static, dynamic and meta-dynamic aspects of a system. The notation chosen in this paper is UML which has the advantage of being adapted and understood by a large community.

To provide a support for behavior implementation, two Design Patterns have been presented. The Polymorphic Behavior pattern allows the definition, the integration and the execution of new behaviors for a system. The Structured Behavior pattern provides the means to define and attach a finite state machine to more complex behaviors in order to model behavior switching. These two patterns extend the range of application of two well-known patterns: command and state. The two patterns are quite complementary, indeed, PB is generally used to analyze a complex system and to architecture the behaviors whereas SB is essentially used to compose, implement and execute these behaviors.

Finally, the necessity to consider a rigorous process for software design which integrates the different design phases of modeling has been highlighted. To complete the proposed modeling framework, a coherent model-based approach supported by model-checking tools, that ensures the development of validated applications, can be considered [26].

References

[1] A. Aarsten, D. Brugali, G. Menga, Designing concurrent and distributed control systems, Commun. ACM 39 (10) (1996) 50–58.
[2] R. Alami, R. Chatila, S. Fleury, M. Ghallab, F. Ingrand, An architecture for autonomy, Int. J. Robot. Res. 17 (4) (1998) 315–337.
[3] C. Alexander, The Timeless Way of Building, Oxford University Press, NY, 1979.
[4] K.J. Astrom, B. Wittenmark, Computer-Controlled Systems, Prentice Hall, Upper Saddle River, NJ, 1997.
[5] M. Beaudouin-Lafon, W.E. Mackay, Reification, polymorphism and reuse: three principles for designing visual interface, in: Proceedings of the Working Conference on Advanced Visual Interfaces, Palermo, Italy, 2000, pp. 102–109.
[6] G. Booch, Object-Oriented Analysis and Design with Applications, Addison-Wesley, Reading, MA, 1994.
[7] R. Buschmann, R. Meunier, P. Rohnoert, M. Sommerland, Pattern-Oriented Architecture—A system of Patterns, John Wiley and Sons Ltd., Chichester, 1996.
[8] J.O. Coplien, D.C. Schmidt, Pattern Language of Program Design, Addison-Wesley, Reading, MA, 1995.
[9] C. Coste-Manière, R. Simmons, Architecture—the backbone of robotic systems, in: Proceedings of the IEEE International Conference on Robotics and Automation, ICRA'00, 2000, pp. 67–72.
[10] P. Dagermo, J. Knutsson, Development of an Object-Oriented Framework for Vessel Control Systems, Technical Report, ESPRIT III/ESSI/DOVER #10496, 1996.
[11] B.P. Douglas, Real Time UML—Advances in the UML for Real-Time Systems, Addison-Wesley, Reading, MA, 2004.
[12] P. Fishwick, Toward a Convergence of Systems and Software Engineering, Technical Report 005, Department of Computer and Information Science and Engineering, University of Florida, 1996.
[13] M. Fowler, UML Distilled, Addison-Wesley, Reading, MA, 2003.
[14] N. Gaertner, Patterns métier et architectures génériques pour la commande et la supervision de processus, Ph.D. Thesis, Université de Haute Alsace, France, July 1999.
[15] N. Gaertner, B. Thirion, Grafcet: an analysis pattern for event driven real-time systems, in: Proceedings of Pattern Languages of Program, PLoP1999 Conference, Urbana, IL, 1999.
[16] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.
[17] D. Garlan, M. Shaw, An Introduction to Software Architecture, Series on Software Engineering and Knowledge Engineering 2, World Scientific Publishing Company, Singapore, 1993, pp. 1–39.

[18] C.P. Jobling, P.W. Grant, H.A. Barker, P. Townsend, Object-oriented programming for computer-aided control engineering, Automatica 30 (1994) 1221–1261.

[19] J. Magee, J. Kramer, Concurrency. State Models & Java Programs, John Wiley & Sons Ltd., Chichester, 1999.

[20] K. Narendra, Adaptation and learning using multiple models, switching and tuning, IEEE Control Systems Magazine (1995) 37–51.

[21] OMG, UML Profile for Schedulability, Performance, and Time Specification, OMG Document ptc/02-03-02, 2002.

[22] J.M. Perronne, M. Hassenforder, An Object Architecture for Advanced Controller Software, SAFEPROCESS'2000, 4th Symposium on Fault Detection and Safety for Technical Processes, vol. 2, Budapest, Hungary, 2000, pp. 751–755.

[23] D. Perry, A. Wolf, Foundations for the study of software architecture, SIGSOFT Software Eng. Notes 17 (4) (1992) 40–52.

[24] M.J. Pont, M.P. Banner, Designing embedded systems using patterns: a case study, J. Syst. Software 71 (2004) 201–213.

[25] A. Ran, MOODS: models for object-oriented design of state, in: J. Vlissides, J.O. Coplien, N.L. Kerth (Eds.), Pattern Languages of Program Design—2, Addison-Wesley, Reading, MA, 1996, pp. 119–142.

[26] A. Rasse, J.M. Perronne, B. Thirion, Design and validation of object-oriented software via model integration, in: Proceedings of the EuroSim, Paris, France, September 6–10, 2004, 6p.

[27] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice Hall, Englewood Cliffs, NJ, 1991.

[28] R. Sanz, Design patterns for intelligent control systems, in: Proceedings of IFAC Congress, Beijing, China, 1999.

[29] R. Sanz, C. Pfister, W. Schaufelberger, A. De Atonio, Software for complex controllers, in: K. Astrom, P. Albertos, M. Blanke, A. Isidori, W. Schaufelberger, R. Sanz (Eds.), Control of Complex Systems, Springer-Verlag, London, 2001, pp. 143–164.

[30] B. Selic, Recursive control, in: R. Martin, et al. (Eds.), Patterns Languages of Program Design—3, Addison-Wesley, 1998, pp. 147–162.

[31] B. Selic, J. Rumbaugh, Using UML for modeling complex real time systems, White paper, ObjecTime Ltd., RationalSoftware Corporation, March 1998.

[32] P. Sommerland, Command processor, in: J.M. Vlissides, J.O. Coplien, N.L. Kerth (Eds.), Pattern Languages of Program Design—2, Addison-Wesley, MA, 1996, pp. 63–74.

[33] B. Thirion, L. Thiry, Concurrent programming for the control of hexapod walking, ACM Ada Lett. 21 (1) (2002) 12–36.

[34] L. Thiry, Modèles, métamodèles et Objets Comportementaux pour les systèmes dynamiques complexes, Ph.D. Thesis, Université de Haute Alsace, France, December 2002.

[35] L. Thiry, B. Thirion, Object-oriented modeling and simulation of complex control systems, in: Proceedings of the European Simulation Multiconference, ESM'02, Darmstadt, Germany, June 2002, pp. 115–120.

[36] A.J.N. Van Breemen, Agent-based multi-controller systems—a design framework for complex control problems, Ph.D. Thesis, University of Twente, Enschede, The Netherlands, 2001.

[37] A.J.N. Van Breemen, T.J.A. de Vries, Design and implementation of a room thermostat using an agent-based approach, Control Eng. Practice 9 (3) (2001) 233–248.

[38] A.J.N. Van Breemen, T.J.A. de Vries, J.B. Striper, An agent-based framework for local model approaches, in: Proceedings of the 16th IMACS World Congress 2000 on Scientific Computation, Applied Mathematics and Simulation, EPFL Lausanne, Switzerland, August 2000.

[39] S. Yacoub, H. Ammar, A pattern language of Statecharts, in: Proceedings of Fifth Annual Conference on the Pattern Languages of Programs, TR #WUCS-98-25, PLoP'98, Illinois, August 1998.

[40] B.P. Zeigler, S.D. Chi, F.E. Cellier, Model-based architecture for high autonomy systems, in: Proceedings of the European Robotics and Intelligent Systems Conference, EURISCON'91, Corfu, Greece, 1991, pp. 3–22.