DESIGN AND VALIDATION OF OBJECT-ORIENTED SOFTWARE VIA MODEL INTEGRATION

A.Rasse, J-M.Perronne, B.Thirion

MIPS Laboratory, LSI Group, UHA
ESSAIM, 12 rue des frères Lumière, 68093 Mulhouse Cedex, France
{a.rasse, jm.perronne, b.thirion}@uha.fr

ABSTRACT

Due to their increasing complexity, software development and maintenance have 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 *analysis models*, the approach helps to obtain both *validation models* which can be exploited with existing model checking tools and specific *implementation models* which conform to the validated models.

KEYWORDS

Object Oriented Modeling, UML, Model Transformation, Model Checking, Design Patterns

1 INTRODUCTION

Modeling control software systems has become more and more difficult due to their increasing complexity. 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 [4]. Despite the undeniable advantage in terms of productivity and intelligibility, the occurrence of emerging behaviors makes reliable software production complicated. It then becomes necessary to check and validate the software systems modeled in this way, since their failures may have economic, material or human consequences. A series of approaches try to introduce more formal aspects and semantics into the *Unified Modeling Language* (UML) [10] specification of systems [7,8,1] to allow their validation. 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 planned approach depends on the composition and transformation of models to make checking and reliable implementation possible. (figure 1).

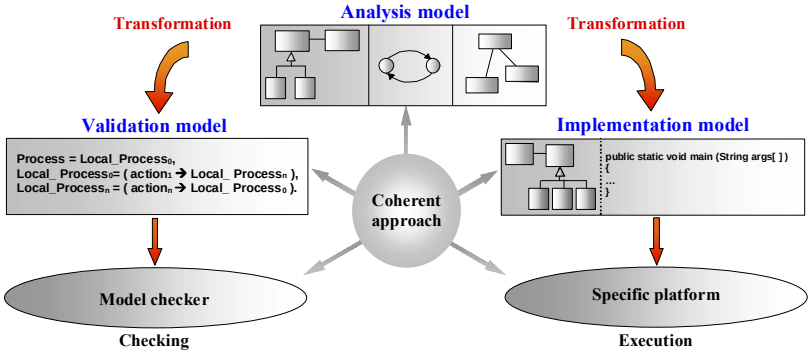

figure 1: Conceptual representation of the approach proposed

The present approach consists of five parts:

- an object oriented analysis in which the structural aspect is divided into two conceptual levels: the resource objects and the behavioral objects. This 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 organisation of the behaviors obtained through this modeling process make them easier to understand and specify.
- a model of the dynamic aspect. In UML [10], Statecharts are commonly used to model the reactive behaviour of models. However, the organisation obtained through structural modeling allows the use of simpler, more precise formalism. Here, finite state machines present an appropriate notation to capture, formally the behaviors associated with each behavioural class.
- a particular configuration of the system in order to obtain the specific behaviors required.
- the validation of the behavioral model obtained earlier. The finite state machines are translated into process algebra called *Finite State Processes* (FSP) [7]. This leads to a validation model which can be exploited with the *Labeled Transition System Analyzer* (LTSA) model checking tool [7].
- 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 [5] to ensure reliable implementation.

This paper is divided into four parts. The first part presents the running example of a legged robot which will be used for the present approach. The next sections describe the *analysis, validation* and *implementation models* respectively.

2 RUNNING EXAMPLE

The system used to illustrate the present approach is an omnidirectional hexapod robot called Bunny [11] (figure 2.a). This mobile platform requires an efficient appropriate control architecture for the integration of a number of coordinated functions. Only the locomotion function will be considered here: a leg moves in a cyclic way between two positions *aep* (anterior extreme position) and *pep* (posterior extreme position) (figure 2.b). 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*. The control architecture is based on decentralised control; the local behaviors obtained with local controllers (LC) are applied to a leg (L) and a global controller (GC) coordinates the local behaviors.

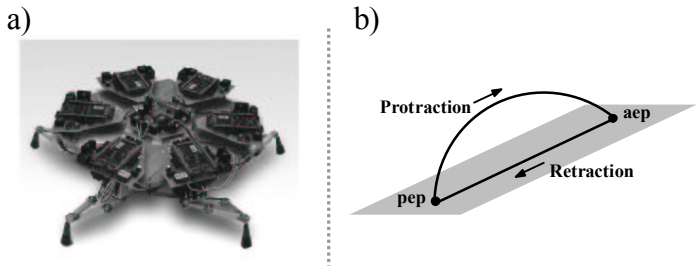

figure 2 : a) Bunny b) cycle of a leg

This system is the source of a number of problems concerning concurrence, synchronization, or decentralized control. So, to ensure robustness and flexibility in locomotion, Bunny 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. Bunny's control software is representative of a class of software systems which must be validated to avoid any problems in their exploitation.

3 ANALYSIS MODEL

The *analysis model* consists in finding a robust and adapted structure that fits the system to be modelled. Based on simple, key entities, 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* (OMG) [9], the UML has become a standard for the description of software systems [6]. In the present approach, *the analysis model* consists of three parts (figure 3): a structural, a behavioral and a configuration part. 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.

3.1 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 of the present models is based on a two-level conceptual model [12], where so-called behavioral objects control objects considered as resources (figure 3.a). The classes describing the resources represent the elements necessary for the description of a system. The classes describing the behaviors control the resources in their state space. This separation helps to isolate and abstract the behaviors of the objects to which they apply and so, to simplify their specification. The concept can also be generalized insofar as a behavior/resource association can be considered as a new resource which is, itself, controlled by another behavior.

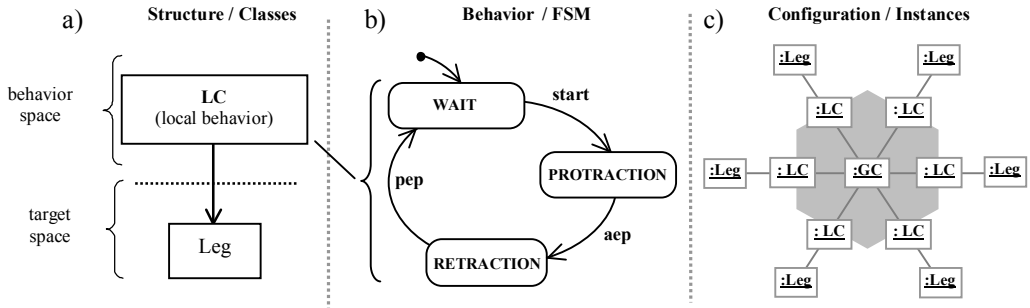

figure 3: analysis model

3.2 Behavioral aspects

Each behavioral class is associated with a finite state machine. These finite state machines specify the dynamic aspect of each elementary entity of the system in the form of event/action sequences. Figure 3.b shows the discrete behavior of a leg equipped with its local controller. To coordinate the legs and so keep the platform stable, the global controller supervises each local controller by allowing (or not) its walking cycle. It is the synchronization of the *start* actions and the concurrent execution of all the local controllers and of the global controller which provide the global behavior of the system. The specification of the global controller will not be detailed here. The locomotion algorithm which has been obtained must be validated to guarantee the above mentioned properties (§ 2). Only the behavioral aspects which have been specified in this way will be validated, as will be seen in the next section.

3.3 Configuration aspects

From the structural and behavioral aspects, the specification of a configuration helps to define a particular application. The instances from the abstractions (classes) are composed in such a way that they provide the specific behavior defined by the requirements. During this composition, the behaviors (finite state machines) are combined and synchronized. The UML object diagram in figure 3.c illustrates the object configuration used for the locomotion function.

4 VALIDATION MODEL

The aim of all validation tools is to make software design reliable and to assure the designers that their specifications actually correspond to the requirements [3]. 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. 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 with a process algebra notation called *Finite State Processes* (*FSP)* [7] based on the semantics of the *labeled transition system* (LTS). So, a system is structured by a set of components whose behaviors are defined as finite state machines. This formalism which is commonly used in the field of checking provides a clear and non ambiguous means to describe and analyse most aspects of finite state process systems [2,3]; it allows the use of the *LTSA* model checker [7]. The behaviors of the present *analysis model,* described in the form of finite state machines, immediately find a correspondence (figure 4.a) with the FSP notation and their translation then becomes obvious. The local behaviors of the *validation model* result from all the behavioral classes and all the associated finite state machines. Figure 4.b represents the FSP translation of the behavior of the *local controller* (LC) class graphically described by its finite state machine in figure 3.a.

Analysis model	Validation model
configuration	parallel composition *instance_1* ‖ *instance_2*
classes	processes
instances	process labeling *instance_name : type_name*
FSM event	action *a* →
FSM state	local process *P=(a → P).*

a) mapping from analysis model concepts to validation model

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

b) behavioral description of a LC in FSP

figure 4: FSP translation

The global behavior is obtained from all the instances of these elementary components and all their interactions within a particular configuration (§ 3.3). In FSP, a process labeling (lc_i:*LC*) provides multiple instances of elementary components, which agree with the instances of the behavioral classes of the present *analysis model*. A set of six local controllers processes (lc_i) is thus created, in which the labels of the actions (*start, aep, pep*) will be prefixed with the label of the particular local controller. The *ROBOT* global behavior (figure 5) is expressed as a parallel composition (||) of the local (lc_i) and global (*gc*) controllers. These are executed concurrently and synchronized on the *start* action using the FSP relabeling operator (/).

```
||ROBOT =  ( lc1:LC || lc2:LC || ...|| gc:GC ) / {gc.start_lc1 / lc1.start,.....}.
```

figure 5: global behavior in FSP

This *ROBOT* behavioral model will be validated by the LTSA model checker. This tool allows the interactive simulation of the different possible execution scenarios of the model specified. To do so, the user selects the different actions he wishes to control. This interactive exploration allows him 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 properties violation. In the *validation model* proposed, great attention will be given to the progress properties which asserts that "*something good eventually happens*". 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. To do so, it is necessary to check the occurrence of the *start* actions for each local controller and their infinitely repeated execution. This property is called *progress Cycle_lc1 = {lc1.start }* in LTSA. If this property is violated by the model, the analyzer produces the sequence of actions that leads to the violation. The designer can then modify his model according to the results obtained.

5 IMPLEMENTATION MODEL

The formal specification techniques and the use of model checking tools do not prevent model mismatch during the development cycle. The *Design Patterns* [5] representing generic implementation models have been proposed to bring an explicit, proven solution to some recurring design problems. Among these design patterns, the *State pattern* gives a solution which is commonly used for the implementation of finite state machines. In the present *analysis model*, the behavior of each component is described as a finite state machine. It is therefore pertinent to associate each component with the implementation which corresponds to the *State design pattern*. Because it is similar to finite state machines, this pattern helps to obtain a code which conforms to the behaviors specified in the *analysis model* and checked by the *validation model*. The implementation of the *State design pattern* for the behavior of a leg and of its local controller is shown in figure 6.

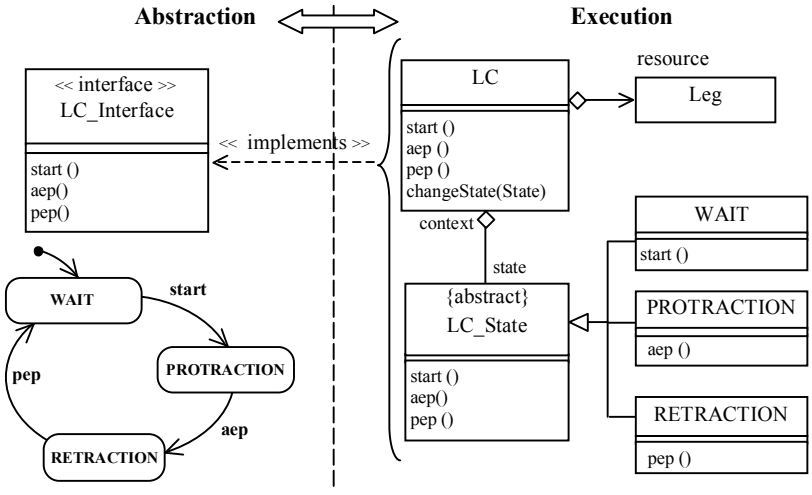

figure 6: implementation of a LC using the *State design pattern*

This implementation diagram consists of a number of elements including:

- An *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 behaviour (*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 behavior 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 a translation of an abstraction (the *analysis model*) into its implementation (the *implementation model*).

6 CONCLUSION AND PERSPECTIVES

This paper has shown the feasibility of 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 (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 implementation of transformation models aiming at a systematic, or even automatic translation, between the different models proposed. The concepts of model transformation will eventually make the development process easier and even more reliable.

REFERENCES

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

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

[3] B.Bérard, M.Bidoit, A.Finkele, F.Laroussinie, A.Petit, L.Petrucci, and P.Schnoebelen (2001). *Systems and Software Verification. Model-Checking Techniques and Tools*. Springer

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

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

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

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

[8] E.Mikk, Y.Lakhnech, M.Siegel, G.J.Holzmann (1998). Implementing statecharts in PROMELA/SPIN, *in proceedings of 2nd IEEE workshop on Industrial-Strength Formal Specification Techniques,* IEEE Computer Society Press, p 90-101.

[9] Object Management Group, http://www.omg.org/

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

[11] B.Thirion, L.Thiry (2002). Concurrent programming for the Control of Hexapode Walking, *ACM Ada letters*, vol. 21, N°1, pp.12-36.

[12] L.Thiry, J.M.Perronne, B.Thirion. Patterns for Behavior Modeling and Integration, *Computer in Industry*, Elserier Ed, to appear.