Using Process Algebra to Validate Behavioral Aspects of Object-Oriented Models

Alban Rasse*, Jean-Marc Perronne*, Pierre-Alain Muller, Bernard Thirion***

* MIPS, ESSAIM, Université de Haute Alsace
12 rue des frères Lumière, 68093 Mulhouse, France
{Alban.Rasse, Jean-Marc.Perronne, Bernard.Thirion}@uha.fr

** IRISA / INRIA Rennes, Campus Universitaire de Beaulieu
Avenue du Général Leclerc, 35042 Rennes, France
pierre-alain.muller@irisa.fr,

Abstract. *We present in this paper a rigorous and automated based approach for the behavioral validation of control software systems. This approach relies on metamodeling, model-transformations and process algebra and combines semi-formal object-oriented models with formal validation. We perform the validation of behavioral aspects of object-oriented models by using a projection into a well-defined formal technical space (Finite State Process algebra) where model-checkers are available (we use LTSA; a model checker for Labeled Transition Systems). We then target an implementation platform, which conforms to the semantics of the formal technical space; in turn, this ensure conformance of the final application to the validated specification.*

1 Inroduction

The increasing complexity of control software systems makes their comprehension and their construction more and more difficult [11]. The approach proposed in this paper (figure 1) simplifies the reliable design of these software systems through a complete software development cycle (from the specification to the code) in a coherent and automated way. It is based on existing techniques, from different fields of software engineering, and integrates:

- a specification phase based on object-oriented decomposition.
- a validation phase based on formal methods and model-checking tools, so as to provide software designers with checking techniques that improve their design quality.
- an implementation phase to ensure the coherence of the generated code according to both the validation and specification phases.
- a model-based software engineering process in accordance with Model-Driven Engineering (MDE) [4], which allows - through a metamodel architecture - the integration of the specification, the validation and the implementation phases into a coherent software development cycle. Moreover, model transformation – a key concept in MDE – helps to go from

one modeling field to another, which, in turn, helps to obtain automatically, from a source model, models that are adapted to a particular technical space. These transformations make the software designer's tasks easier by hiding, as far as possible, the complexity of formal tools which often require an important learning effort.

As the whole approach cannot be described in this paper, only the *specification* and *validation* phases, with the associated transformations, will be considered here (dark gray in figure 1).

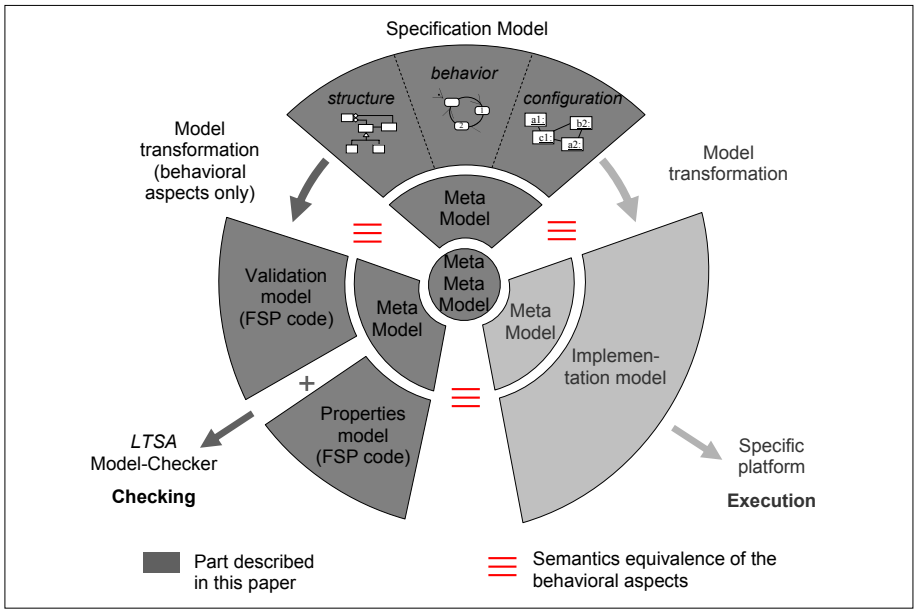

Figure 1. Projection of the behavioral aspects into a process algebra technical space

The approach is based on a *specification model* which represents an abstraction of the control software. This model is specified using classes, objects and Finite State Machines (FSM) so as to describe the different aspects (structure, behavior, and configuration) of the system under study. FSMs have been chosen as this formalism is based on known semantics [8] which can be interpreted in terms of Labeled Transition System (LTS) [1]. The precisely defined semantics is necessary - on one hand - to allow the easier use of model transformation techniques and - on other hand - to ensure the coherence of the approach, since the behavioral aspects of the proposed models (*specification, validation and implementation*) are also based on semantics that can be described in term of LTS. The FSMs are translated into a process algebra [3] called Finite State Processes (FSP) [8]. This leads to a *validation*

model which can be analyzed with the *Labeled Transition System Analyzer* (LTSA) model checking tool [8].

This paper is divided into four parts. The first part presents the running example which will be used to illustrate the proposed approach. The second and third sections describe an overview of the *specification model* and the *validation model* respectively. Finally, the fourth section presents the model transformation concepts necessary for the generation of the *validation model*.

2 Running Example

The system used to illustrate the present approach is a control software whose role is to manage the locomotion function of an hexapod robot [12] (figure 2.a). A leg moves in a cyclic way between two positions aep (anterior extreme position) and pep (posterior extreme position) (figure 2.b). The control architecture is based on decentralized control [7]; the walking cycle of a *leg* (L) is obtained with *local controllers* (LC) and the global behavior is obtained with six *local supervisors* (LS) which coordinate the *local controllers* (figure 2.c).

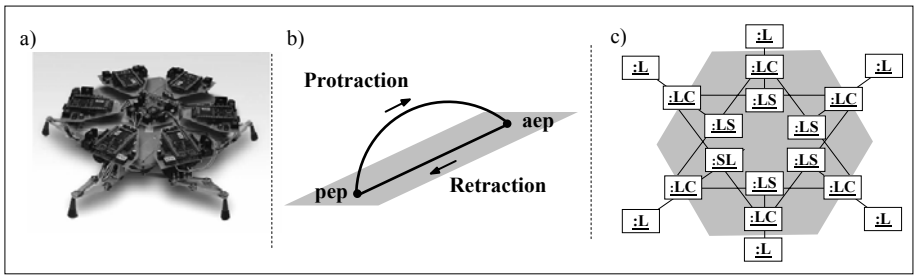

Figure 2. a) Mobile platform, b) Walking cycle, c) Control architecture

To ensure flexible and robust locomotion, this system must satisfy a set of liveness and safety properties. As an example, one of these liveness properties says that all the legs must always execute their walking cycle, whatever the possible execution trace of the system. And in accordance with the safety properties, one leg can only be raised if its two neighbors remain on the ground (static stability). The control software of this robot is a typical example of the software systems which must be validated to avoid severe dysfunctions at runtime.

3 Specification Model

The *specification model*, based on object-oriented models, represents an abstraction of the control software and includes three complementary aspects which represent, respectively, its structure, its behavior and its configuration.

3.1 Specification of the structural aspects

To describe the different types of entities present in control systems, we specify the structural aspects in the form of two conceptual levels [9]. The first level models the *passive* objects which must be controlled, while the second level corresponds to *behavioral* objects (active entities) whose role is to control *passive* objects in their state space (figure 3.a). This explicit representation of behaviors allows these to be considered as full objects and so, to be manipulated and organized within an object-oriented architecture. Moreover, the systematic separation of *passive* objects from *behavioral* objects helps to abstract and isolate them and thus to simplify their specification. This organization can also be generalized since a *passive/behavior* association can be considered as a new (passive) object which is, itself, supervised by another *behavior* (figure 3.a).

3.2 Specification of the behavioral aspects

We model the dynamic aspects of control systems by associating each *behavioral* class with a Finite State Machine (figure 3.b). Figure 3.b models the discrete behavior of a leg controlled by its *local controller*, which is itself coordinated by its *local supervisor*. Once specified in this way, the behavioral objects execute an elementary task, in an autonomous and independent manner, and their concurrent execution describes the entire state space of the six legs.

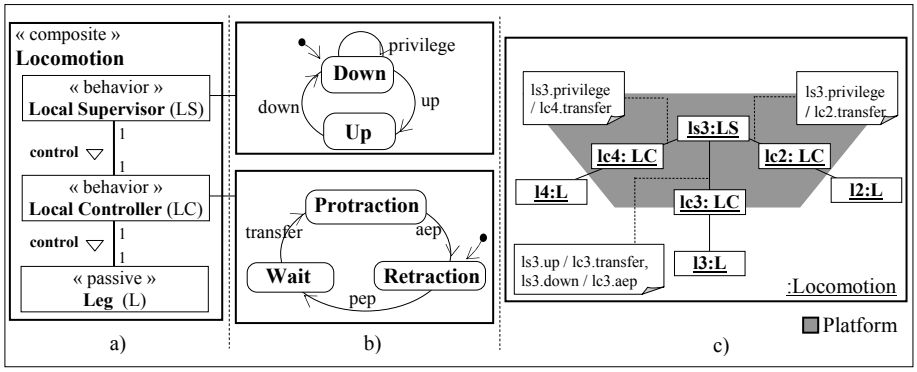

Figure 3: *Specification model* of the *Locomotion* function: a) structural aspects, b) behavioral aspects, c) configuration aspects

To ensure reliable locomotion, some of these states - for example the state in which all the legs are raised at the same moment - must be prohibited. To restrict the entire state-space to the allowed state-space, we allow (or not) some transitions to be fired by synchronizing the actions of the LC instances with those of the LS instances. These synchronizations (or shared actions) are detailed in the configuration aspects. Moreover, we propose to combine *behavioral* and *passive* objects together in a *composite* object (figure 3.a), so as to explicitly represent a modeled software function (here the *Locomotion*). To make design easier and development effort profitable these *composite* can be manipulated and (re)used to model more complex software functions in a hierarchic and modular way.

3.3 Configuration aspects

The previously described behavioral and structural aspects specify a set of possible configurations of a family of software systems in terms of classes, interactions and behaviors. Consequently, modeling a particular software system of this family requires the description of a particular configuration. This particular configuration, which is represented with an object diagram (figure 3.c) helps to better define the structural aspects by specifying the topology and interactions of the instances which make up the software system. Moreover, it also helps to better define the behavioral aspects by specifying - in the form of relabeling annotations [8], (*instance1.actionA /instance2.actionB*) - the actions which are shared between these instances. These shared actions allow to synchronize instances in order to obtain the desired behavior. The object diagram in figure 3.c illustrates part of the configuration of the mobile platform. This diagram shows, in accordance with the previously mentioned safety property, how the *local supervisor* ls3 allows the evolving of *local controller* lc3 according to the position of the two neighboring *legs* l2 and l4. Indeed if *legs* l2 and l4 are raised (lc2 and lc4 receive the *privilege* to do their protraction: *ls3.privilege/lc4.transfer* or *ls3.privilege/lc2.transfer*) then *leg* l3 can only be in the *Down* state (figure 3.b). Conversely, if the *legs* l2 and l4 remain on the ground, *leg* l3 can be allowed to rise (*ls3.up/lc3.transfer*) which will then preempt the *privilege* of its neighbors.

This last specification phase helps to complete the *specification model* whose global behavior (*Locomotion* function) must be validated so as to make sure that its specification respects the expected properties.

4 Validation Model

Simulation and model-checking techniques aim to make software reliable by ensuring designers that their models meet their requirements [2, 5]. The integration of these complementary methods into object-oriented constructions seems pertinent as they allow the efficient validation of software systems. In the proposed approach,

the *validation model* is described in the form of process algebra called *Finite State Process* (FSP) [8] in order to use LTSA [8]. The advantage of LTSA is that it allows both the simulation and the checking of behavioral models.

4.1 Specification of the validation model using FSP

In LTSA, a system is structured using a set of primitive processes, whose behavior is modeled in FSP in the form of expressions combining local processes and actions. The representation of the global behavior of systems is obtained with the composition of instances of these processes (*instance: Process*) and with the representation of their interactions through shared actions within a composite process. So similarly to the *specification model*, modeling a composite process allows the specification of a complex system in a modular, hierarchic way; the instances of composite processes are potentially reused in another composite. To specify the *validation model*, we collect the entities contained in the *specification model* (states, actions, relabeling annotations, …) to transform these entities into FSP (i.e. section 5). Thus, as shown in figure 4.a, for the *local controller* (LC), the behavior of a behavioral class, graphically described by its FSM (figure 3.b), is used to obtain the primitive process (LC) in FSP.

a)

```
LC          = Retraction,
Retraction  = ( pep      -> Wait       ),
Wait        = ( transfert -> Protraction ),
Protraction = ( aep      -> Retraction ).
```

b)

```
|| Locomotion = ( lc1 : LC || lc2 : LC || …
                 || ls1 : LS || ls2 : LS || … )
                 / {
                 ls3.privilege / lc2.transfer,
                 ls3.up / lc3.transfer,
                 …}.
```

Figure 4. Behavioral description in FSP, a) of the LC primitive process, b) of the *Locomotion* composite process

In a second step, the composite type instances which are presented in the configuration aspects (figure 3.c) are used to generate the composite processes in FSP (figure 4.b). As an example, the *Locomotion* behavior is obtained from a set of six instances (lci) of the primitive process *local controller* (LC) and six instances (lsi) of primitive processes *local supervisor* (LS). These instances are composed in a parallel way (||), then synchronized (/) using their shared actions - thanks to the annotation (*ls3.privilege/lc2.transfer, ls3.up/lc3.transfer*, etc…) - included in the *Locomotion* composite object (figure 3.c). This *Locomotion* behavioral model is then checked using LTSA.

4.2 Analysis of the validation model

LTSA allows the interactive simulation of the different execution traces of the specified model to ensure that the latter satisfies the expected behavior. Simulation, which is a non-exhaustive validation, can be completed with a search for violation of liveness and safety properties. In the *validation model* proposed here, only the liveness properties will be presented. A liveness property asserts that « something good eventually happens » [2]. In LTSA, liveness properties are expressed with the keyword progress. The liveness property mentioned earlier (at the end of section 2) consists in checking that each *local controller* (lci) can always execute its walking cycle, which results in the recurrent detection of the *transfer* action for each *local controller* (figure 5).

```
progress Leg1_Cycle = {lc1.transfer },…, progress Leg6_Cycle = {lc6.transfer }.
```

Figure 5. Liveness properties in FSP

If a property is violated by the *validation model*, LTSA produces the sequence of actions leading to this violation. The designer can then modify his/her model according to the obtained results.

5 Model Transformation

Model-Driven Engineering [4] aims to unify software activities from the specification down to the executable code production, through the integration of heterogeneous models into coherent software developments. This coherent integration is only possible - according to MDE - through a formally defined metamodeling architecture which allows - through different levels of abstraction (models, metamodels, metametamodel) - the precise definition of the concepts used to characterize a particular type of (meta)model. In this architecture, metamodels describe all the concepts necessary for the definition of a specific type of models, while the metametamodel specifies the concepts that are common to the metamodels used. So, from these common concepts, a set of relations between the entities of the metamodels can be deduced. Figure 6.a describes the correspondence of the concepts of the *specification metamodel* and those of the *validation metamodel*. The transformation rules which can be deduced from these relations are applied to the entities of a source model (here, the *specification model*) in order to obtain the entities of the target model (here, the *validation model*) in a systematic way. Moreover, the explicit representation of the metamodels and transformation rules allows the use of model transformation tools for the automated generation of specific target models (figure 6.b). In accordance with MDE, the present approach is based on the concepts of models, metamodels and model transformations and has

been prototyped with a metamodeling environment – MetaEdit [6] - in order to transform the *specification model* into a *validation model* (FSP code). The FSP code obtained in this way can directly be analyzed with the LTSA tool. As the proposed models respect the LTS semantics, the semantic gap between these models is reduced, which makes the transformation between models easier. Moreover, the use of model transformation tools makes the proposed approach even more reliable by avoiding the errors that would be caused by manual transcriptions.

a)

Specification metamodel	Validation metamodel
Behavior classes	Primitive processes (Pp)
Instances	Instance of Pp
FSM states	Local processes
FSM action	Action prefix
Guard	Guard
Composite classes	Composite processes
Shared action	Relabeling
…	…

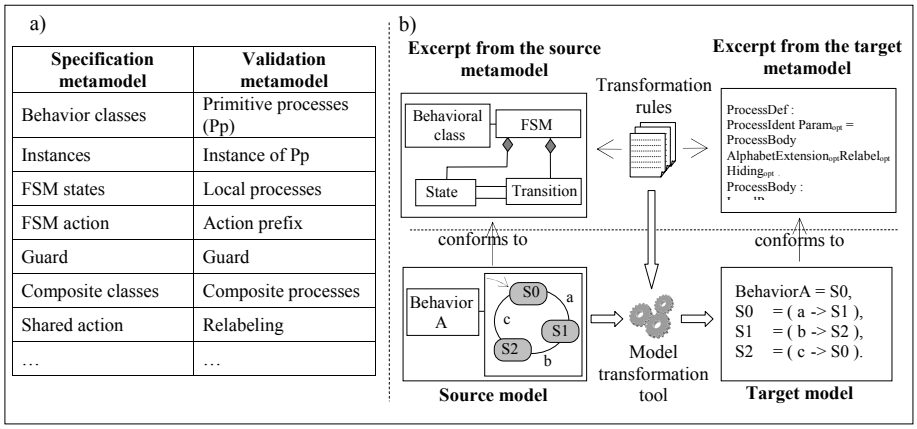

Figure 6.a). Correspondence between the *specification* and *validation metamodel*, b) Conceptual representation of metamodeling

As said in the introduction, the aim of the present approach is to produce an executable code for the implementation of validated control software. However, even if the joint use of object-oriented techniques, checking tools and model transformation techniques makes software development easier and more reliable, it does not guarantee that the implementation conforms with the validation. That is why, the approach presented in this paper is part of a global software development (figure 1) in which the use of a framework and a runtime platform – also in conformity with LTS semantics – helps to reduce the semantic gap between the models and thus allows the easier generation of a code in accordance with the *specification* and *validation models* [10]. So, this approach allows the creation of a coherent software development cycle that integrates specification, validation and implementation phases.

Conclusion and Perspective

This paper has presented an approach combining object-oriented techniques with formal validation and MDE, to ensure the validated specification of control software. In a first step, it proposes an object-oriented specification completed with FSM for the modeling of software systems. The *specification model* thus obtained is

sufficiently precise to be used as a source model for automated software generation. It can be transformed into a process algebra so as to be validated with a model-checking tool. This approach which has been applied on a locomotion software system has the advantage of making the conception of software systems easier while increasing their reliability and also of being integrated in a coherent global development ranging from the specification to the implementation. We will continue this work, in a first step, by the checking of other liveness and safety properties to validate more effectively the *Locomotion* function of the robot. In a second step, we plan to implement the approach on a number of various applications to test its robustness.

Reference

[1] Arnold, A., Finite Transition System, Prentice Hall, Prentice Hall, 1994.

[2] Bérard, B. et al. Systems and Software verification. Model-Checking Techniques and Tools, Springer, 2001.

[3] Bergstra, J.A., Ponse, A. and Smolka, S.A. editors, Handbook of Process Algebra. Elsevier Science, Amsterdam, 2001.

[4] Bézivin. In search of a Basic Principle for Model-Driven Engineering, Novatica Journal, Special Issue, March 2004.

[5] Clarke, E.M., Grumberg, O. and Peled, D. Model checking, The MIT Press, Cambridge, Mass., 1999.

[6] Domain Specific Modeling with MetaEdit+, January 2005, http://www.metacase.com/

[7] Lin, F., and Wonham W.M., Decentralized Control and Coordination of Discrete-Event Systems with Partial Observation. IEEE Transactions on Automatic Control, vol.35, n°12, p.1330-1337, 1990.

[8] Magee, J. and Kramer, J., Concurrency. State Models & Java Programs. John Wiley & Sons, Chichester, UK, 1999.

[9] Perronne, J.M., Rasse, A., Thiry, L., Thirion, B., A Modeling Framework for Complex Behavior Modeling and Integration, Proceedings of IADIS'05, Algrave, Portugal, 2005.

[10] Rasse, A., Perronne JM., Thirion, B. Toward a Validated Object-Oriented Design Approach to Control Software. Proceedings of 16th IFAC World Congress, Prague, Czech Republic, 3-8 July, 2005.

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

[12] Thirion, B. and Thiry, L., Concurrent programming for the Control of Hexapode Walking, ACM Ada letters, n°21, 2002, p.12-36.