Proceedings of the 17th World Congress
The International Federation of Automatic Control
Seoul, Korea, July 6-11, 2008

Functional (Meta)Models for the Development of Control Software

Laurent Thiry*, Bernard Thirion*

** ENSISA - 12, rue des frères Lumière - 68093 MULHOUSE Cedex (France)*
{laurent.thiry, bernard.thirion}@uha.fr

Abstract: The development of control software is a complex task: it requires the integration of many descriptions and tools for specification, design, deployment, etc. To reduce the gap between the various models used to describe a system, this paper proposes the concept of functional metamodels, i.e. precise descriptions of a modeling language embedded into a functional language. The interest of the concept is the ability to define frameworks to simplify the development process. In particular, the paper proposes a framework for specification, design and deployment and applies it to the example of a legged robot.

Keywords: Software engineering, (Meta)Modeling, Programming approach

1. INTRODUCTION

The development of software for control requires the integration of models and tools ; each one being dedicated to a stage of the development (Sanz and Arzen, 2003). To simplify the integration of models, this paper defines the concept of functional metamodel. Indeed, combining functions with descriptions of models (called metamodels) makes possible the design of simple tools that can be integrated in the development process of software for control, with modeling, checking, and deployment. The tools proposed as an illustration are written in the functional language Haskell and make software analysis and design easier ; each tool captures a modeling language with its syntax and semantics. Thus, a framework for developing safety critical systems is presented ; the framework is based on Labeled Transition Systems to describe and simulate a system's behavior, Linear Temporal Logic to verify that a behavior satisfies a temporal property, and a prototype of concurrent language to implement checked behaviors. As an example, the paper presents the development of a control software for a legged robot, Fig. 1.
This paper is divided into three parts. The first part presents the development process as a family of expertise to integrate, the difficulties to do this integration and the solution brought by the Model Driven Development (MDD) community ; see (Mellor *et al.*, 2003). The second part presents the advantages of functional languages combined with model driven principles and proposes a framework to model software for control, check their properties and get the code to deploy. The third part concludes by summarizing the results which were obtained and the perspectives which where considered.

2. SOFTWARE FOR CONTROL

2.1 Development process: from multi-model to metamodels

Overview. Nowadays, the development of software dedicated to control systems becomes increasingly complex. In particular, such software must be produced in a shorter time and at a lower cost, and must be of higher quality (Henzinger and Sifakis, 2006). The problem is that the design process requires a set of various descriptions, or formalisms, to integrate ; see (Thiry *et al.*, 2003). So, the methodologies and tools which are used have to be considered with a great attention.
The development process is the rendez-vous of many actors or specialists, each one uses its own languages and tools, and a lot of energy/time is lost integrating these various points of view ; see (Mosterman *et al.*, 2004). In the field of software for control, the main steps that compose the development are specification, design and deployment. The specification consists in translating the informal requirements expressed by a client into expressions in a formal language ; it describes what must be done with the main components that characterize a software system. Indeed, control systems usually consist of a set of entities, performing a set of tasks, distributed on a network and embedded on dedicated processor. The global behavior resulting from the parallel composition of these entities/tasks must satisfy requirements and formal models are the means of proving this satisfaction ; see (Magee and Kramer, 2006). Design describes a possible solution that can satisfy the specification and the requirements. This step requires creativity and must be compatible with the previous step ; the models which are used have to be compared to specification models for formal proofs. The creativity consists in identifying the various configurations of a system and the paths, or sequences of actions, between them. Deployment is a translation of the design models into code that can be compiled and deployed in the physical architecture. This part requires an interpretation of the concepts used by models into programming constructs. The difficulty here is that design models do not always integrate programming facilities such as processes, channels, rendez-vous, etc. Then, the development process is characterized by a family of modeling languages necessary for specification, design and deployment.

Sample development. To illustrate the development process, this section describes the various models used to design a control software for a legged hexapod ; see (Thirion and Thiry, 2002), Fig. 1.

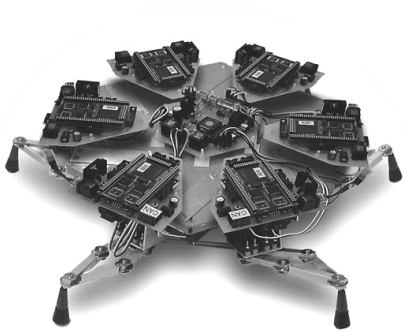

Fig. 1. Example of complex system to control.

The specification expresses the properties required, by the way of logical formulae. The formulae give a formal model of the requirements expressed by a client. For instance, the following expressions imply that a leg can not go up and move if the preceeding or following leg is up (for stability reasons), and each leg contributes to the global movement of the platform (for equity reason), i.e. globally (`G`) if a leg i moves then (`=>`) in the future (`F`) it will push.

```
forall i in [0..6],
 G (not (Move(i) /\ Move(i+1 mod n)))
 G (Move(i) => F Push(i))
```

The design proposes a state model representing the order of actions to be performed . The main models used for a design are Finite State Machines (FSM) ; see (Antoniotti and Mishra, 1995). Fig. 2 gives an overview of the robot behavior. The proposed model must satisfy the previous formulae ; this is done by tests or model checking ; see (Clarke *et al.*, 2000).

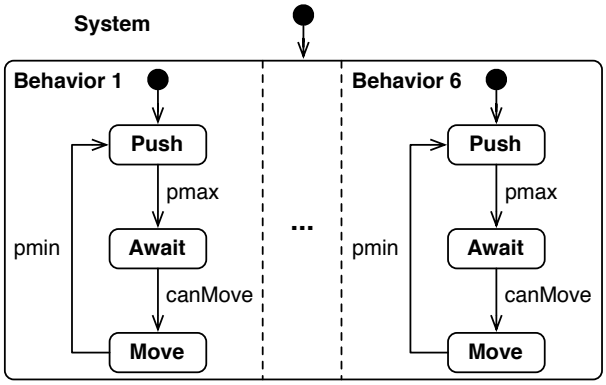

Fig. 2. The movement of the robot results from the parallel composition of the legs' behavior.

Using a programming language, the deployment gives the code to put in the various controllers. Most often, this step uses dedicated platform with threads and networks ; it is important to be able to interpret design models with these concepts. The following code gives a translation of the design model into an imperative/concurrent style, and it is the basis for the control software. The code is based on basic actions (`pos`, `update` to access or update shared variables as the position `p` of a leg i), control structures (forever) and combinators (; for sequence and || for parallel composition).

```
behavior i = forever do { push i;
 await i; move i }
push i = do { p<-pos i;
 if p<pmax then update i (p+delta)
  else return () }
main = (behavior 1)||...||(behavior 6)
```

Conclusion. Developing software for control is a difficult task and current works try to propose concepts, methodologies and tools for the integration of many modeling languages ; see (Henzinger and Sifakis, 2006). An elegant solution has been brought by the Model Driven Development (MDD) community described in the next part.

2.2 Model Driven Development (MDD)

Overview. The MDD community defines concepts and tools to capture and then integrate families of modeling languages ; see (Mellor *et al.*, 2003). Each modeling language is represented by way of a metamodel, usually an UML class diagram extended by OCL constraints, and is often called Domain Specific Language (DSL) ; see (Deursen *et al.*, 2000). The metamodels are used to configure metamodeling tools, i.e. generic tools used to edit, to save/restore, or to translate into code models conformed to the specified metamodel. Sample tools are the Generic Modeling Environment GME (Ledeczi *et al.*, 2005) or Atom3 (Lara and Vangheluwe, 2002). Concerning model transformations, they are usually based on graph transformations, i.e. rewriting rules between two (meta)models.

Sample metamodel. To understand the concept of metamodel, this section presents a metamodel for dataflow models, Fig. 3. Dataflow models are used to model continuous system with signals and functions on these signals ; Fig. 4 presents an example of dataflow model. The dataflow models can be presented graphically: a function is represented by a box (called block) with inputs similar to the arguments of the function represented, and the resulting outputs. The composition of functions is represented by linking inputs to outputs. Blocks can be classified into two categories: Basic blocks, to map inputs to outputs by an equation, and Composite blocks, to define hierarchy inside the models. All the concepts describing dataflow models can be represented by a metamodel, itself represented graphically by an UML class diagram. The Unified Modeling Language (UML) is a standard to model software architectures ; see (OMG, 2007).
The concept of functional metamodels, presented in part 3, consists in replacing the metamodel description language UML by a functional language to exploit the notions of (meta)model and transformation, and then simplify the development of control software. For instance, Fig. 5 proposes the UML metamodel for Labeled Transition Systems (LTS) and the code below the figure its translation into the functional programming language Haskell.

Well formed dataflow models are defined by the previous metamodel completed by logical expressions in the Object Constraint Language (OCL). For instance, a rule saying that an input can be connected to one, and just one, output is formalized by the OCL expression:

```
context i:Input
inv OneConnection: i.output->size()==1
```

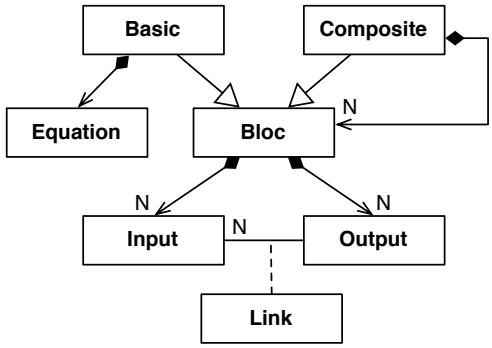

Fig. 3. Metamodel specifying dataflow models (see Fig. 4 as an example).

Conclusion. UML is described using class diagrams and OCL expressions in a reference document (OMG, 2007) of a thousand pages ; what can be considered a bit too complicated to be really useable. Moreover, in specific fields (real-time applications for instance), the notation is not sufficient and must be extended by using profiles (i.e. extension of UML metamodel). Finally, UML/OCL are considered as semi-formal in the sense that their are described with themselves. This is a problem in the field of critical systems where it must be possible to formally prove that a system satisfies requirements. Thus, the standard approach proposed by the MDD community which uses generic tools and UML can be described as too complicated.

This paper proposes an approach to exceed these limitations, and the following part shows how the functional paradigm can make the concepts of metamodel and transformation practicable. Indeed, the concepts of Domain Specific Language (DSL) or "meta" appeared early in the history of functional languages, and it is interesting to show how advances in this field combined with metamodels and transformations can be profitable for the development of control software.

3. FUNCTIONAL METAMODELS

3.1 Functional models

Foundation. The functional paradigm is based upon the lambda calculus, a metamodel of computation based on the concepts of function, application of function to an argument and rewriting rules from an expression of a language into another expression ; see (Hudak, 1989). Modern functional languages, such as Haskell, add to this metamodel the concepts of type and constant to increase intelligibility and to reduce errors. A type is defined by a set of functions. For instance, lists are defined with a constant (the empty list `[]`), and the operator (`:`) which adds an element at the beginning of a list. Then, `1:2:3:[]` is a list of integers, and can be written `[1,2,3]`. A special construct of a language with no special semantics is called syntactic sugar. Then, the first advantage of Functional Models (FM) is their simplicity (i.e. complex computations can be done with short expressions) and their ability to compute easily a collection of objects with lists (such as matrixes with eventually symbolic expressions): for instance, Haskell is built upon a simple grammar and a standard library with a set of generic functions ; see (Peyton Jones, 2003).

The second advantage of FM is their ability to model computations, i.e. the way an expression is reduced or evaluated. Contrary to imperative models, functional models support lazy evaluation and, implicitly, dataflow models with infinite streams ; see (Mathaikutty, 2005). As an illustration, the following model declares a continuous dynamical system and its simulation. Such a model is used to compute trajectories of the legged robot (Fig. 1).

```
h  = 0.01; f  = 1; c  = 2*cos(2*pi*f*h)
minus  = zipWith (-) -- Combinators
mult k = zipWith (*)
y =0.0:y1 -- Infinite Stream
y1=sin (2*pi*f*h):((repeat c `mult` y1)
                   `minus` y)
y = take 100 y
-- [0.0, 0.06, 0.12, 0.18, ...]
```

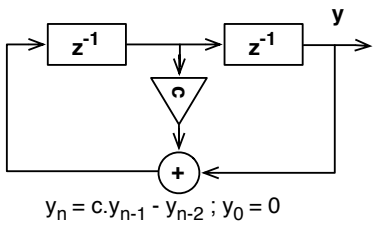

Fig. 4. Dataflow model of an oscillator.

Finally, the third advantage of FM is their ability to easily build embedded languages, or Domain Specific Language (DSL). For instance, Pembeci *et al.* (2002) have proposed a functional model for reactive/control systems with robotic and vision parts. A DSL can be defined by a datatype playing the role of a metamodel, and functions playing the role of transformations. Contrary to the standard MDD approach using UML as a foundation, the combination of functional paradigm with the MDD concepts can lead, as shown in part 3.2, to a simple but complete framework integrating a family of languages for specification, design, deployment, and so on.

Modeling actions. Programming features of more classical programming languages can be easily captured using monadic types ; see (Peyton Jones, 2001). In particular, an action is defined by a type `IO a = s->(a,s)` ; where s is the execution environment before, then after, the computation, and a is the type of the value computed. The primitive `getChar` and `putChar` are built upon this definition and have respectively the type `IO Char` (say an action that computes a character) and `IO ()`. Similar definitions are used to define mutable variables (monad State), optional results or errors (monad Maybe), alternative results (monad Either), concurrency (monad Continuation), etc. Monads are associated to special functions: "return" to create a monadic value, and "bind" combinator (>>) to compose monads. Bind is used with IO to create sequences of actions and exchange values between them. The concept of sequential composition of actions is described precisely and in a simple way. Moreover, functional actions can be put into lists and can be passed as argument or result of functions, etc. Then, monads make possible the use of imperative aspects inside declarative/functional model ; the metamodel of code used for deployment is based on the concept of monadic action and is presented in the next part.

Conclusion. Combining functions with MDD concepts makes possible the definition and use of specific languages for control. The next part exploits the presented elements to propose

an other framework for specification, validation and implementation of discrete dynamical systems. As an illustration, this part detailed the development environment used for the control software of the legged robot, Fig. 1.

3.2 Functional metamodels in control

The development process described in part 2.1 is based on a set of languages, each one being adapted to a step of the process. As shown in part 2.2, a modeling language can be described by a metamodel. Part 3.1 presents functional languages as a mean to design easily embedded languages (called DSL) ; each DSL is based upon a metamodel and a set of transformations. Using the functional paradigm, this part proposes DSLs (i.e. metamodels) adapted to the various steps of the development process identified.

Specification and validation. Properties are usually formalized with logical expressions. Expressions are modeled by a grammar that is interpreted, in Haskell, by a datatype definition. The following model specifies a subset of temporal logic with: constant (`FTrue`), basic properties, operators of classical logic (`Not`, `Or`) and temporal operators (`neXt`, `Until`). The semantics of these constructs, represented by s, is based on traces t, i.e. sequences of states. ; each state corresponding to a set of properties. This model also describes how to build syntactic sugars (`globally` and `imply`) on the top of the previous elements.

```
type Trace e = [Set e]
data Form e = FTrue | Property e
            | Not (Form e)
            | Or (Form e) (Form e)
            | X (Form e)
            | U (Form e) (Form e)

s :: Trace e -> Form e -> Bool
s t FTrue = True
s (ps:pss) (Property p) = member p ps
s t (Not f) = not (s t f)
s t (Or f f')  = (s t f) || (s t f')
s (ps:pss) (X f) = s pss f
s t@(ps:pss) g@(U f f') = ((s t f) &&
      (s pss g)) || (s t f')
s _ _ = False

globally f = U f FTrue
imply f g = Or (Not f) g
```

The datatype definition can be described by an UML metamodel, similar to the one of Fig. 3: Form becomes a parametric class with an operation `s(t:Trace):Boolean` and `FTrue`, `Property`, `Not`, etc. become subclasses ; the implementation is described by the rules under data definition. This establishes the compatibility between the elements proposed and the standard MDD approach ; tools cited in part 2.1 can be exploited. However, one can wonder about the interest of having a UML visual model for a formulae such as `(Push => X Move) U Stop)`? The following model gives an example of test for this expression using the framework proposed.

```
t = [["Push"],["Move"],["Stop"]]
f = U (imply (Property "Push")
             (X (Property "Move")))
      (Property "Stop")
r = s t f -- is True
```

Anyone can copy the functional metamodel below, and then get a simple model checker for the bounded LTL to check that a finite trace t satisfies the formulae f (see the result r). To continue the example, it is necessary to define a model of design. The next section proposes a metamodel for the Labeled Transition Systems (LTS) and a function to run an lts and obtain a trace useable with the function s ; testing all possible traces of the lts with a property p will prove that the lts satisfies p.

Design. The discrete behavior of a system is generally described with a set of configurations, or states, and a set of transitions between these configurations ; transitions are generally controlled either by an event received by the system, or by an action in which the system is engaged. More precise definitions of such structures are Finite State Machine - FSM (for events) and Labeled Transition Systems - LTS (for actions). LTS were proposed by Plotkin to formally describe the semantics of programming languages ; in particular, they are used to describe process algebra which are modeling languages for concurrent communicating processes. The Finite State Process (FSP) proposed by Magee and Kramer (2006), for instance, is a process algebra built upon LTS and is used to describe systems behaviors, and properties to check before deployment. As the other languages (CSP, CCS or ACP), FSP defines syntactic sugars to facilitate the definition of complex behaviors integrating hierarchy (i.e. an LTS inside a state), concurrency and synchronization. Thus, LTS, as lambda calculus in functional models, represent a formal kernel language for concurrent systems and are well suited for design. Moreover, they can be easily integrated into the specification model of the previous part to prove that a behavior proposed is in accordance with the requirements. Fig. 5 presents the UML metamodel based on the mathematical definition of the LTS and the code below gives the (second) functional metamodel: classes are translated into data structures and transformations to functions on these structures.

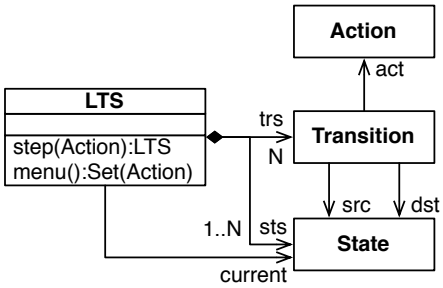

Fig. 5. Metamodel of Labeled Transition Systems for behavior and properties specification.

An example of model is given by the legs' behavior of Fig. 2 ; the system behavior is obtained by composing local behaviors into a single LTS. In the functional metamodel, the semantic of the LTS is captured by the function `step` and auxiliary function `menu` which returns the possible actions a system can engage into ; a potential deadlock is then an

empty menu `[]`. The `step` function describes the evolution of a LTS from state to state and corresponds to a model transformation. A run corresponds to a sequence of actions and can be used to generate a trace. Then, it is possible, using the specification model and function s, to check if this trace satisfies a set of properties.
The following code proposes an implementation of the LTS metamodel, an example of model (lts) and the transformations onto the metamodel (step and run).

```
data LTS s a = -- Metamodel
  LTS { current     :: s,
        states      :: [s],
        transitions :: [(s,a,s)] }

lts = LTS "Push" -- Sample model
  ["Move","Push","Await","Stop"]
  [("Push","pmax","Await"),
   ("Await","canMove","Move"),
   ("Move","pmin","Push"),
   ("Push","pmax","Await")]

menu lts =[a|(s,a,_)<-transitions lts,
   s==current lts]
lock lts = menu lts == []

step (LTS cur sts trs) act =
 LTS cur' sts trs
   where cur' = head [d|(s,a,d) <- trs,
     s==cur, a==act]

run lts trs = do
  if not (lock lts) then do {
    putStrLn (show (menu lts));
    act <- getLine;
    run (step lts act) (act:trs)}
  else return (reverse trs)

compose :: (Lts s a) -> (Lts s a) -> Lts s a
```

As for LTL, the implementation of the LTS is simple and can be used to define design models, test (with run) or check (with s) them. Then, it is possible to define syntactic sugars to define lts more easily with product of two lts (for parallelism), relabeling of actions (for synchronization), or copy. Concerning the product (see function compose) of two LTS, used to model concurrency, it returns a new global lts for the whole system and is consequently not interesting for deployment: target platform/language generally integrates dedicated constructs for concurrency and distribution. Moreover, modeled actions must be translated into concrete actions for control (beginning with read/write values). The next section proposes an embedded language into Haskell for concurrency. This language is defined by a third functional metamodel and can be use to describe design model with imperative and concurrency features before the deployment to the target architecture.

Deployment. Deployment consists in translating the design models into a specific programming language and platform. In the field of functional languages, concurrency can be easily modeled with the Continuation monad: each action (see IO, part 3.1) is extended by the rest of the computation, called continuation. As for the specification part, the metamodel for concurrency is better described with a textual metamodel ; see the following code. In the tool proposed for deployment, continuations are integrated to IO actions to mimic interleaving of, possibly infinite, sequences of actions. The parallel combinator (||) makes the interleaving and is detailed by (Peyton, 2001). The function schedule captures the semantic of round robin used in the domain of multitasking. The communication between processes is based on the State monad which plays the role of a shared memory ; this concept is detailed by the previous author.

```
data Action = Atom (IO Action)
  | Par Action Action
  | Stop

schedule [] = return ()
schedule (Atom m:rem) = m >>= (\m' ->
  schedule (rem++[m']))
schedule (Par m1 m2:rem) =
  schedule (m1:m2:rem)
schedule (Stop:rem) = schedule rem

exec m = schedule [m stop]
```

The previous metamodel/language is then used to run the deployment model for the legged robot, simply with `exec main` ; see code after Fig. 2. This code is obtained by translating/transforming the LTS , presented in the previous section, into statements of an imperative language: if_then_else or forever, for instance. This transformation from design model to a deployment model is difficult to describe ; so, the approach proposed here consists rather in giving the type IO Action to the functions step and run. Thus, design models can be integrated into deployment models in a simple/elegant manner. For instance, the behaviors presented in the example of part II.1 is replace by (run lts_i) and the behavior of the robot is given by:

```
main=schedule [Par (run lts1) (run lts2) ...]
```

The previous expression can be executed on a single processor ; or it can be broken into sub-expressions that whill be executed by many processors to increase performances. Distribution on a network is the next step of the development process and belongs to the perspectives of this work.

Synthesis. Part 3.2 presents a framework based on the concept of functional metamodels and applies it to the analysis and design of software for control. Each expertise required by the development process is captured by an Haskell (meta)model and is naturally integrated to the others. Consequently, there is no need for extra tools to model, check and run complex systems such as the legged robot used as an illustration, Fig. 1.

4. CONCLUSION

This paper has proposed the concept of functional metamodels which consists in describing metamodels in a functional language. The interest of the concept is the possibility to integrate modeling paradigms in a common framework. As an illustration, the paper has detailed a family of metamodels

implemented with Haskell for specification, design, simulation and checking, and deployment. The resulting framework has been applied to the development of a control software for a legged robot, Fig. 1.

More precisely, the use of datatypes and functions, representing metamodels and model transformations, can capture the various points of view, knowledge and methods, necessary in the field of control software. As an example and an application of the concept of functional metamodels, the paper has presented three tools to model systems behavior with Labeled Transitions Systems (LTS), to check properties modeled by formulae from Linear Temporal Logic (LTL), and a prototype of concurrent language to implement checked behaviors.

The perspectives envisaged with these works will consist mainly in refining the development process and to identify its other advantages/limits, in particular with the integration of continuous and discrete (meta)models, and the model of the target architecture (with resources, networks, sensors/ actuators, etc.).

REFERENCES

Antoniotti M. and Mishra B. (1995). *Discrete Event Models + Temporal Logic = Supervisory Controller: Automatic Synthesis of Locomotion Controllers*. Research Report, NYU

Clarke E.M., Grumberg O. and Peled D.A. (2000). *Model Checking*. The MIT Press, Cambridge

Deursen A., Klint P., Visser J. (2000), Domain-Specific Languages: An Annotated Bibliography, In: *SIGPLAN Notices*, Vol. 35, No. 6, pp. 26–36

Henzinger T.A. and Sifakis J. (2006), The Embedded Systems Design Challenge. In: *the 14th International Symposium on Formal Methods (FM)*, Lecture Notes in Computer Science, Springer

Hudak P. (1989), Conception, Evolution, and Application of Functional Programming Languages. In: *ACM Computing Surveys*, Vol. 21, Issue 3, pp. 359-411

Lara J. and Vangheluwe J. (2002). AToM3: A Tool for Multi-formalism and Meta-modelling. In: *the 5th International Conference on Fundamental Approaches to Software Engineering*, pp. 174-188

Ledeczi A., Balogh G., Molnar Z., Volgyesi P., Maroti M. (2005). Model Integrated Computing in the Large, In: *IEEE Aerospace,* pp. 1-8

Magee J. and Kramer J. (2006). *Concurrency: State Models & Java Programs*, John Wiley and Sons Eds, Chichester, UK

Mathaikutty D.A. (2005). *Functional Programming and Metamodeling frameworks for System Design*. PhD Thesis, Faculty of Virginia Polytechnic Institute and State University

Mellor S.J., Clark A.N., Futagami T. (2003). Guest Editors' Introduction: Model-driven development. In: *IEEE Software*. Vol. 20, No 5, pp.14-18

Mosterman P.J., Sztipanovits J., Engell S. (2004). Computer automated multi-paradigm modeling in control systems technology. In: *IEEE Transactions on Control System Technology*, Vol. 12

OMG (2007). UML 2.1.1 Superstructure Specification. Available from www.uml.org

Pembeci I., Nilsson H., Hager G., Burschka D., Peterson J. (2002). Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. In: *Principles and Practice of Declarative Programming*, PPDP'02, Pittsburgh, Pennsylvania, USA

Peyton Jones S. (2001), Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In: *Engineering theories of software construction*, Tony Hoare, Manfred Broy, Ralf Steinbruggen Eds, IOS Press, 2001, pp. 47-96

Peyton Jones S. (2003). *Haskell 98 Language and Libraries.* Cambridge U. Press, 2003.

Sanz R. and Arzen K.E. (2003). Trends in software and control. In: *IEEE Control Systems Magazine*. Vol. 23. No. 3, pp. 12-15

Thirion B. and Thiry L. (2002). Concurrent Programming for the Control of Hexapod Walking. In: *ACM Ada Letters*, Vol. 21, N°1, pp. 12-36

Thiry L., Perronne J.M., Thirion B. (2003). Patterns for Behavior Modeling and Integration, In: *Computers in Industry*, Elsevier Ed, pp. 225-237