Functional metamodels for systems and software

Laurent Thiry, Bernard Thirion

Functional metamodels for systems and software

Laurent Thiry, Bernard Thirion
ENSISA, 12 rue des frères Lumière - 68093 Mulhouse (France)

Abstract

The modeling, analysis and design of systems is generally based on many formalisms to describe discrete and/or continuous behaviors, and to map these descriptions into a specific platform. In this context, the article proposes the concept of functional metamodeling to capture, then to integrate modeling languages. The concept offers an alternative to standard Model Driven Engineering (MDE) and is well adapted to mathematical descriptions such as the ones found in system modeling. As an application, a set of functional metamodels is proposed for dataflows (usable to model continuous behaviors), state-transition systems (usable to model discrete behaviors) and a metamodel for actions (to model interactions with a target platform and concurrent execution). A model of a control architecture for a legged robot is proposed as an application of these modeling languages.

Key words: System modeling, functional metamodeling, Model Driven Engineering (MDE)

1. Introduction

The analytical and computational modeling of systems generally requires the composition of various models and formalisms for their study, their analysis or their design, Vangheluwe and de Lara (2003). For example, dataflows, used to describe continuous behaviors, must be composed with discrete elements to capture operating modes, and with more software oriented elements, Henzinger and Sifakis (2007). Moreover, computational models can be described using an imperative language, an object oriented language, a functional language, etc.

The Model Driven Engineering (MDE) community has brought a reflection on what "models" of systems, or models of modeling languages also called metamodels are and what the relations between models also called model transformations are, Mellor et al. (2003). MDE is based on three points of view, Figure 1. The most concrete is the one of systems experts that model, analyze or design systems. To do this, they need tools that are developed by software experts ; the tools have generally a "formalism inside" (e.g. matrix algebra for matlab). At a more generic level, recurrent elements, on which the preceding tools are based, are capitalized by models experts ; in particular, the latter provide concepts and means to manipulate modeling languages and models that will be used by software experts to develop more easily/quickly specific tools, that will be used by systems experts.

As a result, MDE proposes a set of dedicated languages: Meta Object Facility (MOF) to specify metamodels, Object Constraint Language (OCL) to add semantics or constraints on modeling elements, Query/View/Transformation (QVT) to relate (meta)models, etc. The description of formalisms inside MDE is generally object-oriented. The paper proposes an alternative approach that is function-oriented and is named *functional metamodeling*. This approach has an impact on the three layers of Figure 1 and it proposes new means to

Email addresses: laurent.thiry@uha.fr (Laurent Thiry), bernard.thirion@uha.fr (Bernard Thirion).

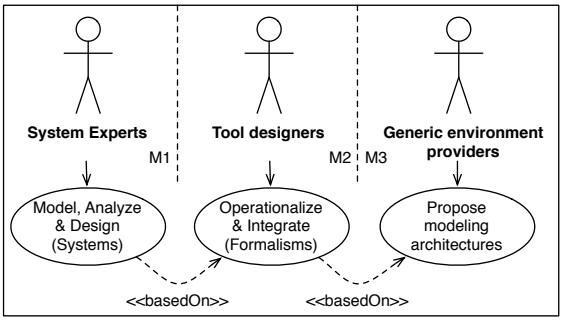

Fig. 1. Abstraction layers in MDE.

specify and to relate modeling languages ; these elements will be detailed in part 3. The concept leads to particular tools ; examples will be given in part 4. The specificity of the tools leads to another way of modeling systems ; an illustration will be presented in part 5.

The advantages of functional descriptions are their formal/mathematical foundation, and their ability to describe other formalisms in a compact and rational manner. Despite these interests, functional models and languages are rarely being considered in the domains of modeling, simulation, control, etc. Thus, to facilitate the use of functional languages, the paper proposes a framework that explains how to describe, then to integrate, behavioral models (with continuous and discrete parts) and how to combine them with implementation models (necessary for the deployment on a target platform). The work presented also explains how to profit from the modern functional programming language Haskell, with high level constructs, to model dynamical systems differently, Thiry and Thirion (2008).

So, with our proposition, a metamodel will correspond to a set of functions that can be composed to built expressions that are, de facto, models. Transformations are functions too and are based on the previous definition. As an application, a set of *functional metamodels* is proposed for continuous behaviors (i.e. dataflow models), for discrete behaviors (i.e. state-transition models) and for more software oriented elements (i.e. with concurrent actions). As a complement, the article explains how to 1) get either a numerical representation, or an analytical representation, of continuous dataflow models (using Series, Mc Ilroy (1998)) ; 2) extend discrete behavioral models with temporal logic formulae representing abstract specifications usable for model checking, Clarke et al. (2000). Described in a compact and comprehensible way, the elements presented allow the study of a wide range of systems such as the legged robot of part 5. And in contrast to dedicated tools like Matlab/Simulink [1] , the framework proposed can be easily adapted or extended.

The paper is divided into six parts. Part 2 introduces the concepts of systems, models and metamodels. Part 3 details the main contribution of the paper with *functional modeling* ; i.e. what it is and what the benefits obtained are. Part 4 proposes an application of the concept to systems. More precisely, this part describes five functional metamodels and a set of tools for the modeling/simulation of continuous, discrete and concurrent systems. Continuous behaviors can be modeled using either dataflows (such as the ones found in Matlab/Simulink), or mathematical series. Discrete behaviors are modeled by state-transition systems and logical formula expressing constraints that have to be satisfied/checked. Part 5 proposes a use of the framework to model the control architecture of a legged robot. Part 6 concludes by summing up the main elements of the paper and presents the perspectives considered.

2. Systems, models and metamodels

2.1. *Dynamical systems*

From a mathematical point of view, a *system* is modeled by a time T, a set of states X, a set of inputs U, and a transition function $F: X \times U \rightarrow X$, Lee and Varaiya (2003). Figure 2 presents a classification of system models according to T and X, Maler (1998). A *behavior* is defined by an initial state $x_0 \in X$, an input $u: T \rightarrow U$, a function $x: T \rightarrow X$ that satisfies $x(0) = x_0$ and $x(i+1) = F(x(i), u(i))$ for a discrete time T, and $dx(t)/dt = F(x(t), u(t))$ for a continuous time T. The output of a system is given either by a function $y: T \times X \times U \rightarrow Y$, where Y is the set of outputs, or by completing the transition function $F: X \times U \rightarrow (X \times Y)$. U and Y may be numerical values (continuous case) and/or symbolic values called events (discrete case).

The models above are used to describe most dynamical systems. However, for the implementation, other elements have to be considered such as the model of computation, Jantsch and Sander (2005). Particularly, most of the time a system is decom-

[1] www.mathworks.fr

X / T	Continuous (IR)	Discrete (IN)
Continuous (IR^{IN})	Differential Algebraic Equations	Finite Difference Equations
Discrete	Naive Physics	State-Transition Systems

Fig. 2. Classification of system models.

posed into many sub-systems, whose behavior is given by one of the previous models, and that have to evolve concurrently. As a consequence of this decomposition, it is necessary to precise the model of computation considered, i.e. how time is shared between the various sub-systems and how these latter communicate. For continuous behaviors, a synchronous model of computation is mostly used ; for discrete behaviors, an asynchronous model of execution is privileged. In both cases, a scheduler sets the order of execution for each component. In the synchronous model, global time T is divided into instants and each component makes a step in an instant. In the asynchronous model, each component can make many steps during an instant (i.e. each sub-system has its own time basis T_i). Thus, in this model, it is necessary to add software elements such as communicating channels (or shared variables) to allow data exchanges or to synchronize the sub-systems.

The previous description shows that systems modeling is based on various kinds of models with continuous or discrete behaviors on the one hand and other parts representing implementation details. These parts are complementary and have to be integrated. Figure 3 presents an example of system combining a continuous element (P) and a discrete element (Q) that have to evolve in parallel (i.e. $P \mid\mid Q$). This example will be used in part 4 to illustrate the functional metamodels proposed.

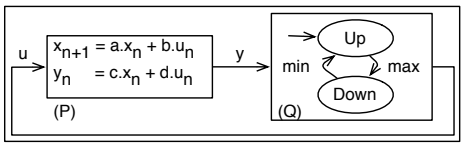

Fig. 3. Composition of continuous, discrete and concurrent models.

2.2. *Model Driven Engineering (MDE)*

Model Driven Engineering (MDE) proposes concepts and tools to capture, then to integrate, modeling languages, Mellor et al. (2003). Each modeling language is defined by a metamodel, generally a MOF class diagram [2] extended with logical constraints, Varro and Pataricza (2003). Figure 4 presents the conceptual framework on which MDE is based on. A system is modeled using a *modeling language* ; a *model* is an element of the modeling language which is the set of all models conforming to a *metamodel*, Favre (2004), and Varro (2002). A model *transformation* represents a relation between two modeling languages and is defined by a set of *mappings* between the corresponding metamodels. As an illustration, the behavior of an oscillator can be modeled by a regular expression (min.max)*. The metamodel defines the syntax of regular expressions using basic elements min/max, sequences (.), repetitions (*), etc. The modeling language is then the set of all valid regular expressions. The translation from a regular expression into a finite state model is an example of transformation ; the target metamodel is defined by states and transitions and the target model has two states, e.g. 0/1, and two transitions (0,max,1) and (1,min,0), Figure 3.

Other examples of metamodels for systems are given by Breton and Bezivin (2001) for Petri nets, and Denckla and Mosterman (2005) or Mathaikutty (2005) for block diagrams. Metamodels can also be defined as a Domain Specific Language (DSL), Deursen et al. (2000). DSLs can also be embedded in programming languages ; more precisely, the components of a metamodel are mapped into data types and sets of functions to create and to manipulate models. For example, Hudak et al. (2003) provide a DSL embedded into the functional programming language Haskell and dedicated to robotics and vision.

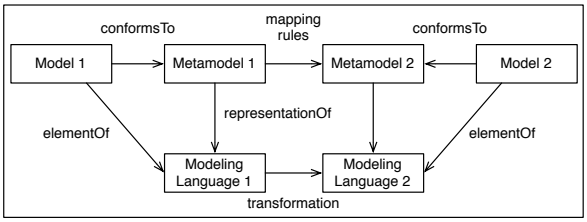

Fig. 4. Extract of MDE megamodel.

Metamodels can also be used to configure generic (meta) modeling environments, proposed by the MDE community, in order to get tools dedicated to a specific formalisms (i.e. to create and to ma-

[2] www.omg.org/mof/

nipulate models for these formalisms). Examples of metamodeling tools dedicated to systems are the Generic Modeling Environment GME, Dubey (2005), and Atom3, Vangheluwe and de Lara (2003).

To explain the concept of metamodel, Figure 5 specifies an example of a possible metamodel for block diagrams. Block diagrams are used to model continuous dynamical systems with signals and transfer functions on these signals. The diagram of Figure 3 shows an example of block diagram that conforms to this metamodel. A transfer function f is represented graphically by a block with inputs (corresponding to the arguments of f), and outputs (corresponding to the results of f). Transfer function composition is obtained by linking the output of a block to the entry of another block. Blocks can be classified into two categories: basic blocks, which relate entries to outputs with algebraic equations (Figure 2) and composite blocks whose internal structure is given by interconnecting more simpler blocks. Composites are useful to organize models in a hierarchical way. The concepts that appear in this description can be represented on a class diagram (Figure 5): modeling elements/concepts are represented by classes and relationships between these concepts are represented by associations (between the corresponding classes), or compositions (marked by a diamond).

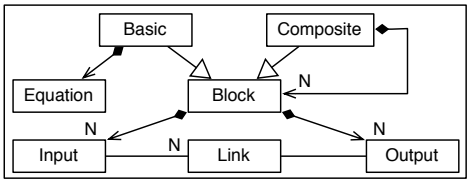

Fig. 5. Metamodel for block diagrams.

Well-formed block diagrams are defined by the previous metamodel plus a set of logical expressions expressed, for instance, with the Object Constraint Language (OCL). For instance, a rule saying that an input can be connected to one, and only one, output is formalized by the OCL expression:

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

The languages used to specify meta-models (i.e. here MOF and OCL) are described by reference documents of a thousand pages ; which makes them difficult to use in practice. As a consequence, other languages (such as Essential MOF - EMOF, or Eclipse Modeling Framework - EMF), based on a restricted set of concepts, are preferred. Moreover, all these languages are not sufficient to describe other parts of MDE such as concrete syntaxes of the models or means to transform models ; so, other languages have to be considered (e.g. the Query/View/Transform - QVT, for model transformations). The use of a restricted set of concepts to specify models, metamodels and model transformations will be helpful and is the main concern of the article. In contrast to classical metamodeling, which exposes the structure of the abstract syntax of a modeling language through MOF, EMOF or UML diagrams, the proposed approach tries: 1 - to hide the structure following the concept of abstract data types, Guttag (2002), and 2 - to specify transformation models by sets of equations.

Before presenting the concept of *functional metamodeling* in part 3, part 2.3 describes what is called functional modeling.

2.3. *Functional modeling*

As shown in the previous sections, systems and models can be described in various manners. In particular, systems and models can be mathematically specified by sets and relations (or functions) that can be represented visually by graphs. For instance, the behavior of a system can be modeled by a set of states and transitions, and the structure by a set of objects and links. The functional paradigm, used by functional programming languages, is based on this kind of description. More precisely, a function f will correspond to a particular relation between two sets, also called data types (A, B) and will be written $f : A \rightarrow B$. A datatype will correspond to a set of functions to build values, to compose values, or to map values into values of another type. A function is associated to a definition $f(x) = y$ and to applications $f(z)$ whose semantics consists in evaluating the expression y with the occurrences of x replaced with z. The expression y consists of (constant) values or other functions. Functions can be composed with a "dot" operator defined by $(f \circ g)(x) = f(g(x))$.

Functional programming languages are based on the previous elements plus *a set of, generally unrecognized, features*, Wadler (1996). The most important ones are:

– The definition/use of functions, called *constructors*, that are not necessarily attached to an expression. In particular, all constants (e.g. $1 : \mathbb{N}$) can be modeled by constructors.

– The possibility *to pass functions as arguments or as result of other functions*. The derivative operator is an example of such a function: if f is a function from $\mathbb{R}$ to $\mathbb{R}$ (i.e. $\mathbb{R} \rightarrow \mathbb{R}$) then the derivate is a function $derivate : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$.
– The capability *to evaluate a function partially*, i.e. a function with two arguments $f : (A, B) \rightarrow C$ can be replaced by a function with one argument that returns another function of one parameter, i.e. $f' : A \rightarrow (B \rightarrow C)$. Thus, if f requires two arguments (a, b) then f' requires only one arguments a ; the result can be applied to another argument b when necessary. This property can be used *to define generic functions* that can generate a set of more specific functions.
– If data types can be composed (e.g. with the cartesian product $A \times B$, or (A, B)), then functions can be composed too ; e.g. if f, g are function then (f, g) is also a function and $(f, g)(x) = (f(x), g(x))$. This possibility is interesting to realize parallel computations, Harisson (2006).

As a consequence of these powerful capabilities, most functional programming languages are based on a few constructs (to define new functions or to use previously defined functions) plus a library of generic functions. All the possibilities offered by the functional paradigm will be used in the rest of the paper to describe (and to integrate) modeling elements and will be called *functional (meta)modeling*. The advantage of the approach is to be based on a small number of concepts and to be well adapted to formalisms based on mathematical descriptions (like the ones found in system modeling (2.1)).

Another advantage of the concept is to be naturally supported by functional programming languages. In particular, the functional models (and metamodels) presented can be translated into the modern functional programming language Haskell, Hudak et al. (2007). Haskell is based on very few constructs, Bird (1998). A data type corresponds to a set of constructor functions and functions on this data type are defined by pattern matching, i.e. one rule for each constructor.

As an illustration, basic arithmetic expressions (Exp) can be specified following the principle of abstract data types, Guttag (2002). $Value$ lifts $Integers$ into Exp, and $plus/mult$ composes two expressions.

$$value : Integer \rightarrow Exp$$
$$plus : Exp \times Exp \rightarrow Exp$$
$$mult : Exp \times Exp \rightarrow Exp$$

Expressions are defined using function applications. For example, a model of $e = 1+(2\times3)$ will be simply:

$$e = plus(value(1), mult(value(2), value(3)))$$

Figure 6 shows a possible object-oriented model for the example. The type Exp and the three constructors are mapped into classes ; the values used by the constructors are mapped to attributes or compositions (e.g. $left/right$ expressions for binary operators $Plus/Mult$). The figure also presents an association (f) to model a transformation (and a function) from expressions to any type T.

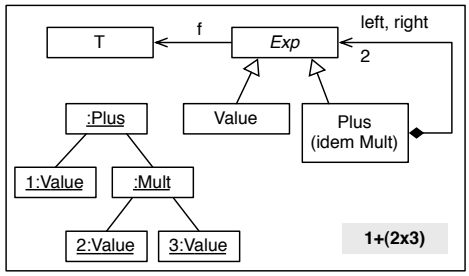

Fig. 6. Models of arithmetic expressions.

The example can be translated into Haskell using a type definition (with the keyword *data*) that groups together the three constructors (constructors have a name followed by a set of parameters ; the result is to the left side of =, here Exp):

```
data Exp = Value Integer
       | Plus  Exp Exp
       | Mult  Exp Exp

-- This definition is equivalent to
-- Value :: Integer -> Exp
-- Plus  :: Exp    ->(Exp -> Exp)
-- Mult  :: Exp    ->(Exp -> Exp)

e = Plus (Value 1) (Mult (Value 2) (Value 3))
```

Notes. First, as a convention, mathematical models will be in italics (e.g. $value$, $plus$, Exp, etc.) and Haskell programs will be written using monospace font as above. Second, parenthesis will be used to avoid ambiguities and $f(x) \equiv f\ x \equiv (f\ x)$.

Now, with this model, all functions on arithmetic expressions, i.e. $f : Exp \rightarrow T$, can be defined by three equations (i.e. one for each constructor) and three parameters : $g : \mathbb{N} \rightarrow T$ (used to transform $Value$) and $h, i : T \times T \rightarrow T$ (to transform Plus and Mult). As an example, the evaluation of arithmetic

expressions can be specified by the function $eval : Exp \rightarrow \mathbb{N}$:

```
eval (Value v) = id v = v
eval (Plus x y)= (eval x) + (eval y)
eval (Mult x y)= (eval x) * (eval y)
```

The function lifts the three constructors into the three functions ($g = id,\ h = (+);\ i = (\times)$). So, *eval* is a particular case of the more generic function f defined by:

```
f g h i (Value v ) = g v
f g h i (Plus x y) = h (f g h i x) (f g h i y)
f g h i (Mult x y) = i (f g h i x) (f g h i y)

eval = f id (+) (*)
```

Figure 7 presents a graphical representation for these equations. Now, choosing other particular values for (g,h,i) allows the definition of most functions on expressions. For instance, the code generation for a stack machine can be defined by:

```
generate = f (\v->"PUSH "+v)
         (\x y->x+y+" ADD")
         (\x y->x+y+" MULT")

generate e == "PUSH 1 PUSH 2 PUSH 3 MULT ADD"
```

Thus, the functional model for arithmetic expressions is defined in a compact and formal manner by four elements: *value/plus/mult* to build expressions, and f to transform expressions ; to do this, f is parameterized by three functions (g, h, i) and all functions on expressions can be defined by choosing specific values for these parameters.

Despite their interest, researches using functional paradigm in Model Driven Engineering for systems are not numerous. Among the latter, Mathaikutty (2005) proposes a set of (meta)models for the modeling and simulation of hybrid systems and Uustalu and Vene (2006) describe a framework for the modeling and analysis of continuous systems. The proposed functional metamodeling paradigm, based on the previous concepts, tries to be more general by describing how functions can capture modeling languages, metamodels, models and model transformations.

3. Functional metamodeling

3.1. *Presentation*

Functional metamodeling is based on a generalization of the approach used to specify arithmetic expressions. More precisely, modeling languages will be captured by sets of functions m_i to construct models ; in the MDE context these sets will be considered to be functional metamodels. The relations, or transformations, between metamodels will be defined in a way similar to the function f for the arithmetic expressions: i.e. a generic model of transformation will correspond to another set of functions f_i plus a set of equations (one for each function m_i).

Theses sets of functions can be represented on a graph: types correspond to the nodes, and the functions to the edges. For instance, Figure 7 proposes another representation for arithmetic expressions ; the dotted parts represent the model $Exp = value, plus, mult$, and the grey parts represent the transformation $f = g, h, i$. The equations of the transformation represent the equalities between the paths on these graphs (i.e. $f \circ value = g$ and $f \circ plus = h \circ (f \times f)$, that also corresponds to the equations of the preceding section $f(value(v)) = g(v)$ and $f(plus(x, y)) = h(f(x), f(y))$).

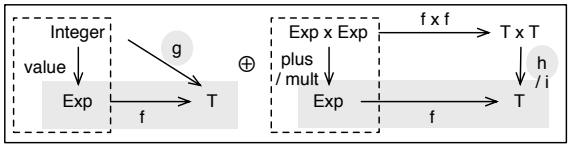

Fig. 7. Graphical representation for expressions.

Remark. From a more formal point of view, the elements presented can be explained in the context of category theory, Walter (1992). However, an understanding of the theory is not necessary to apprehend the elements proposed. The only thing to remember is that diagrams correspond to equations and, with the proposition, to models, metamodels and transformations. The language used to describe equations is defined by elementary functions $f, g : T \rightarrow T'$, applications $f(g)$, compositions $f \circ g$, and products $f \times g$.

As an illustration, a UML object diagram can be built using three functions: to create a new empty diagram, to add an object to a diagram, and to add a link between two objects. The functions $New, AddObject, AddLink$ will be defined in the

functional language Haskell by a type *Diagram* and three constructors:

```
data Diagram = New
         | AddObject String Diagram
         | AddLink  String String Diagram
-- This definition is equivalent to
-- New     :: Diagram
-- AddObject:: String->(Diagram->Diagram)
-- AddLink  :: String->(String->(Diagram->Diagram))
```

Every sequence of actions to create a particular diagram can be modeled using partial evaluation (e.g. $AddObject\ o : Diagram \rightarrow Diagram$) and function composition ($\circ$). For instance, the following *model* has two objects $Producer/Consumer$ and a link between these objects ; $model'$ extends *model* with a $Consumer$ object.

```
actions = AddLink "Producer" "Consumer"
      . AddObject "Consumer"
      . AddObject "Producer"
model  = actions(New)
model' = AddObject "Consumer" model
```

Functions on diagrams, and model transformations, will be defined by following the approach presented, with three equations (one for each construction). For instance, the function *check* returns True if a sequence of actions is consistent ; i.e. the addition of a new object into a diagram requires that this object does not already exist, and the addition of a link requires that the extremities exist. The function *contains* tests if an object belongs to a diagram.

```
contains :: Diagram -> String -> Bool

check :: Diagram -> Bool
check (New)         = True
check (AddObject o d) = not (contains d o)
                  && (check d)
check (AddLink o o' d)= (contains d o)
                  && (contains d o')
-- check(model) == True
-- check(model')== False , Consumer already exist !
```

Rather than defining specific metamodels (such as the one of object diagrams), the paper will focus on two fundamental (meta)models for lists and graphs. Indeed, most modeling languages used to describe systems are based on these two concepts. For instance, behaviors are generally represented by sequences (of states), block diagrams used for modeling the structure of systems or state-transition diagrams used for modeling discrete behaviors are graphs, etc. The models for lists and graphs are presented in the following sections and will be applied in part 4 to a set of modeling languages for systems.

3.2. *Model for generic lists*

The behavior of a discrete time system can be described in various ways like, for instance, a sequence of values (x_i) representing the state x at each instant i, or a sequence of values (a_i) such as $x(n) \approx \Sigma_{i=0}^{k} a_i.n^i$ (for any instant n). Possible transformations between these models are interpolation and sampling. The previous models (that will be detailed in part 4.1) are based on a common model (a_i), also written A^*. A functional description of lists consists of two elements: $empty : \emptyset \rightarrow A^*$, to create an empty list, and $add : A \times A^* \rightarrow A^*$, to add an element to a list of As. From a MDE point of view, $\{emty, add\}$ corresponds to the metamodel for lists and a model for a particular list will be specified by a composition of these functions. For instance, $one = add\ 1\ one = add\ 1\ (add\ 1\ one) = \cdots$ will be interpreted as an infinite sequence of "1". The modeling language captured by this metamodel will be the set of terms $\{empty,\ add\ a_1\ empty,\ add\ a_2\ (add\ a_3\ empty), \cdots\}$, for any a_i in A, and will serve, for instance, to model behaviors. A graphical representation of the functional metamodel is presented on the left part of Figure 8 (with π_i the projections of the cartesian product, e.g. $\pi_1 : A \times B \rightarrow A$) ; the right part corresponds to an equivalent class diagram.

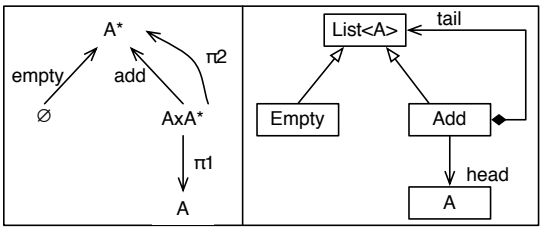

Fig. 8. Representation of generic lists A^*.

As explained, functions can be used to define generic transformations. More precisely, the function $f : A^* \rightarrow T$, defined by a couple (v, g), such as $f(empty) = v$ and $f(add(h,t)) = g(h, f(t))$, will model all common transformations on lists. Figure 9 presents a representation of this transformation (and of the previous equations). Sample transformations are: $map(l) : A^* \rightarrow B^*$ with $v = empty$ and $g(h,t) = add(l(h), t)$, and $select(p) : A^* \rightarrow A^*$ with $v = empty$ and $g(h,t) =$

$if\ p(h)\ then\ add(h,t)\ else\ t$. The function *map* applies a function l to each element of a list and *select* extracts the elements of a list that satisfy a property p.

Remark. The composition of the preceding functions, i.e. $(map(f).select(p))(xs)$, can be used to model sets such as $\{f(x) \circ x \in xs \wedge p(x)\}$. In Haskell, these expressions are encoded by the syntactic sugar [f(x) | x <− xs, p(x)] and are commonly used to express more easily some transformations on lists.

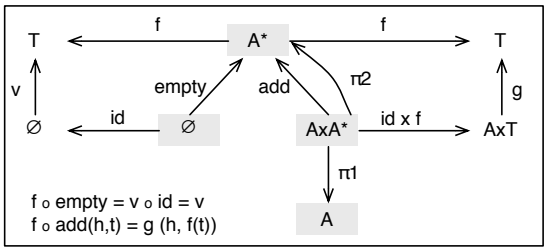

Fig. 9. Generic transformation on lists $f : A^* \rightarrow T$.

The equations used to formalize f can be used to prove properties attached to a particular transformation ; for instance, it can be proved that $map(f) \circ map(g) = map(f \circ g)$. Some interesting properties offered by f are the possibility to *lift* functions $A \rightarrow B$ into functions $A^* \rightarrow B^*$ (with *map*), or the capability to generalize binary functions $A \times A \rightarrow A$ into n-ary functions $A^* \rightarrow A$ (e.g. a Σ expression will be modeled by $sum = f\ 0\ (+)$). These capabilities will be used by the functional metamodel proposed in the next part for dataflow and block diagram metamodels.

The functional metamodel proposed for lists (and more generally for all functional metamodels presented in the paper) is: 1) compact - here, there are only three elements to know: *empty*/*add* and f, 2) formal - the equations for f can serve to prove properties of transformations, Doets and van Eijck (2004), and 3) attached to a visual representation, Figure 9. The metamodel can be encoded directly into Haskell and can serve to capitalize mathematical models based on sets or sequences [3].

The following code gives an interpretation of the preceding equations into Haskell ; the reader will note the similitude with the metamodel for arithmetic expressions.

```
data List a = Empty
        | Add a (List a)
-- or equivalently
-- Empty :: (List a)
-- Add   :: a -> (List a -> List a)

f v g  Empty    = v
f v g (Add h t) = g h (f v g t)

map   l = f Empty (\h t -> Add (l h) t)
select p = f Empty (\h t -> if (p h) then Add h t
                      else t )

model :: List Integer
model = Add 1 (Add 2 Empty)
map (+1) model == Add 2 (Add 3 Nil)
map odd  model == Add 3 Empty
```

[3] These elements are lists with extra properties

3.3. *Model for graphs*

The second metamodel used by the article, and more generally in MDE and systems modeling, is the one of graphs. Indeed, most visual models can be represented by a graph. For instance, MOF class diagrams are graphs whose nodes correspond to classes, and edges to relations between classes (i.e. inheritance, association, composition, etc.), state-transition diagrams are graphs whose nodes are states and edges are transitions, etc.

In practice, a graph can be specified by a list of nodes A^*, a list of edges B^*, and a list of relations between nodes and edges $(A \times B \times A)^*$. Figure 10 presents a graphical representation of this functional metamodel and the code below proposes a Haskell implementation.

```
data Graph a b = Graph (List a)
                       (List b)
                       (List (a,b,a))
```

An example of graph is the model on the left part of Figure 8 that can be translated to [4] :

```
type Model    = Graph Type Function
type Type     = String
type Function = String

fig1 :: Model
fig1 = Graph [ "{}"   , "A"   ,"A*" ,"AxA*" ]
         [ "empty", "add" ,"pi1","pi2" ]
         [ ("{}"  , "empty","A*")
         , ("AxA*", "add" , "A*")
         , ("AxA*", "pi1" , "A" )
         , ("AxA*", "pi2" , "A*")
         ]
```

[4] The notation $[a, b, ...]$ is a syntactic sugar for $add(a, add(b, ...))$.

As for lists, there is a generic transformation f defined by a couple of functions: $f_1 : A \rightarrow A'$ and $f_2 : B \rightarrow B'$, Figure 10. Using the function *map*, this transformation becomes:

```
f :: (a->a')->(b->b')->(Graph a b)->(Graph a' b')
f f1 f2 (Graph as bs rs) = Graph as' bs' rs'
  where as' = map f1 as
        bs' = map f2 bs
        rs' = map f3 rs
        f3 (x,y,z) = (f1 x, f2 y, f1 z)
```

An application of this transformation can be found in the domain of concurrent processes, to express specific operators such as relabeling or parallel composition, Winskel and Nielsen (1995). The second part of Figure 10 presents an application of graph transformation for the relabeling operator.

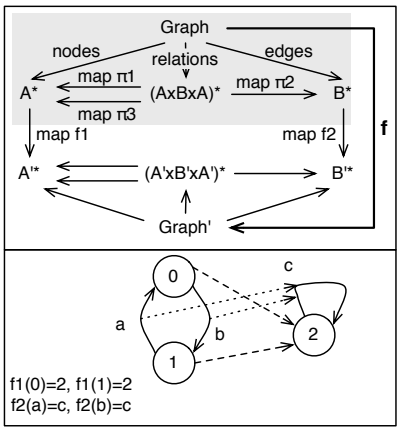

Fig. 10. Representation of graphs (gray part), graph transformations f, and an example.

The next part presents a family of *functional metamodels* dedicated to system modeling. These metamodels are similar to (and are based on) the ones presented for lists and graphs. Next, the resulting framework is used in part 5 to model a control architecture for a legged robot.

4. Functional metamodels for systems

The models of systems presented in part (2.1) can be formalized by metamodels (2.2) that can be specified by sets of functions (2.3). This proposition has lead to the concept of *functional metamodeling* detailed in part 3.

As an application of the proposition, this part presents a framework (i.e. a set of concepts and tools) for dataflow models used to represent continuous behaviors, for state-transition systems used to represent discrete behaviors, and for actions to represent interactions and concurrent executions. Continuous behaviors can be described numerically (4.1) or analytically (4.2). Discrete behaviors can be described from an implementation point of view using state-transition systems (4.3) or from a specification point of view using linear temporal logic formulae (4.4). Section 4.5 proposes an implementation model that facilitates the deployment of the previous specification models onto a target platform.

4.1. *Dataflows*

Dataflows are a standard in the context of discrete time dynamical systems, Thielemann (2004). This kind of model specifies a set of relations between sequences of elements called "flows" or "signals" ; a sequence is defined by a list (3.2). As explained in part 3, sequences can be defined by a functional metamodel (i.e. a set of functions to build and to transform sequences). Then, this metamodel can be implemented into the functional language Haskell and used to model continuous behaviors/systems. More precisely, a flow is either a sequence of elements (e.g. numerical values), or a function that computes an element depending on time, Lee and Varaiya (2003). For example, a flow representing a unit step can be described either by the sequence $step = add\ 1\ step$, or by the function $step\ (time) = 1$. To get the numerical value of this flow (or signal) at an instant t, it is possible to use a *get* operator (!!), i.e. $step\ !!\ t$ for the first representation, or the function application $step\ (t)$ for the second representation. As a remark, the first representation corresponds to an infinite flow: reducing step leads to $step = add\ 1\ (add\ 1\ ...)$. In standard Haskell, the previous function will be written $step = 1 : step$ [5] and can be generalized by a function that lifts any value into a flow, i.e. $lift0\ x = x : (lift0\ x)$. Higher-order functions, i.e. functions taking other functions as argument or as result (e.g. f or *map*), are used to lift operators and functions on values into functions on flows. The application of these higher-order functions to a particular function will be used to model basic blocks found in dataflow models. As an illustration, a model for a block that computes the absolute value (*abs*) of a flow whose first value is x and following values are given by another flow xs,

[5] For readability reasons, the operator (:) will be preferred to the binary function and constructor *add*, i.e. $x : xs \equiv add\ x\ xs$.

is $abs\ (x : xs) = |x| : abs\ xs$, what can be generalized with $abs = lift1\ (||)$ where $lift1\ f\ (x : xs) = (f\ x) : (lift1\ f\ xs)$. The function $lift1$ is similar to the function *map* defined in part 3.2 and any unary function on a value can be lifted using $lift1$. In a similar manner, a function $lift2$ is defined for binary functions or operators (with $lift2\ f\ (x : xs)\ (y : ys) = (f\ x\ y) : (lift2\ f\ xs\ ys)$), a function $lift3$ for ternary operators, etc.

In dataflow models, flows can be composed or transformed using blocks. A block is either a primitive "function" (e.g. add, integrate, multiply or gain, delay, etc.), either a "composition" of more simple blocks. Figure 11 presents an example of a dataflow model for a first-order system.

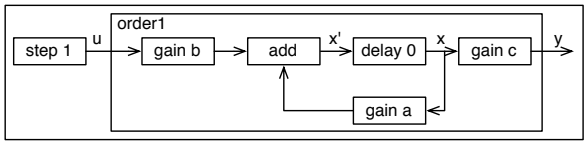

Fig. 11. Example of a dataflow model.

The previous description is fundamentally "functional" in the sense that flows are functions from instants to values, and blocs are functions from flows to flows. By considering discrete time, a flow will be simply modeled by a list of values A^*, and blocks by functions $A^* \rightarrow B^*$, $A^* \times B^* \rightarrow C^*$, etc. For instance, a step generator that returns a constant value will be modeled by $step : \mathbb{R} \rightarrow \mathbb{R}$, a delay whose initial value is $z0$ by $delay\ z0 : \mathbb{R} \rightarrow \mathbb{R}$, an adder by $add : \mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$, etc. A graphical representation of the functional metamodel is presented on Figure 12.

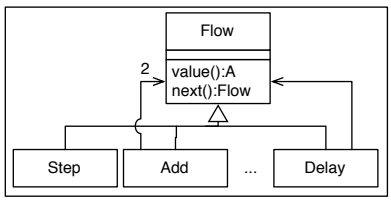

Fig. 12. Metamodel for dataflows.

More precisely, the functional metamodel proposed for dataflows is based on the one of list (3.2). The difference between lists and flows is that the latter are not necessarily of finite length (i.e. flows do not have the possibility to be empty). To make this distinction, the functional metamodel of flows is restricted to one constructor F that plays the same role as *add*. As a consequence, functions on flows (i.e. transformations on flows such as $lift_i$, and blocks) are now defined by a single rule/equation as illustrated by the functions *delay x0* or *gain k* below.

```
-- specification of Flow and Block
data Flow a = F a (Flow a)
-- i.e. F :: a -> (Flow a -> Flow a)

type Block a b = Flow a -> Flow b

-- examples of basic blocks
delay, gain :: a -> Block a a
delay x0 = F x0
gain k   = lift1 (*k)

lift1 f (F x xs) = F (f x) (lift1 f xs)

add :: Block (a,a) a
add = lift2 (+)

lift2 f (F x xs) (F y ys) = F (f x y)
                      (lift2 f xs ys)

-- example of composite block
firstOrder a b c u = y
  where x  = delay 0 x'
        x' = add (gain a x) (gain b u)
        y  = gain c x
```

Despite its simplicity, this functional metamodel allows the modeling and the simulation of most of discrete time systems. As an illustration, the following model describes how to get the first values of the response of a first order system to a unit step. These values can be saved into a file and then edited using a spreadsheet to draw the systems' behavior ; figures 14 and 19 have been obtained in this way.

```
system :: Block Float Float
system = firstOrder 0.9 0.1 1.0

u, y :: Flow Float
u = step
  where step   = lift0 1.0
        lift0 v = F v (lift0 v)
y = system u
y == [0, 0.1, 0.19, 0.27, 0.34,
```

Thus, the definition of a modeling language dedicated to discrete time systems leads to a set of functions {*step*, *gain*, *delay*, *add etc.*}, that can be composed and then evaluated to get the behaviors of these systems. All the functions are based on generic functions $lift_i$ that model transformations on flows. A change on the equation of $lift_i$ leads to other semantics and other possible metamodels. In particular, the function $lift_0 : A \rightarrow List(A)$ can be defined by $lift_0\ v = v : (lift_0\ v)$ or by $lift_0\ v = v :$

($lift_0$ 0). The first equation is used in this section for numerical models of behaviors, and the second equation is used in the next section for analytical models of behaviors.

4.2. *Analytical models of continuous behaviors*

A dataflow model is generally a graphical representation of a finite difference equation or of a differential algebraic equation (see 2.1) ; the basic blocks correspond to the mathematical operators used in this equation. The semantics used in the previous section consists in computing the successive values of a flow using other flows or using the preceding values of the considered flow. The trajectory/behavior of a system corresponds to these successive values.

Another approach to compute the behavior is to use mathematical series. This change in the interpretation of dataflow models lies essentially in the manner a value is lifted: in dataflow model, a constant value v is transformed into a constant flow $[v, v, v, ...]$; in series based models, a value v is transformed into a list of coefficients for the series, i.e. $[v, 0, 0, ...]$. More precisely, a behavior is derived from a sequence of values (v_i) ; the state at the instant n is then v_n with the dataflow interpretation, and $\sum_{i=1}^{\infty} v_i \times n^i$ with series. As for flows, series can be described using the lists' metamodel (3.2): a list ($add\ x\ xs$), also written ($x : xs$) in Haskell, will correspond to a polynomial $p(n) = v0 + n \times vs$ where vs is another series (and another polynomial). Operators on series are expressed in the same way as basic blocks in the functional metamodel for dataflows. As an illustration, the integration of a behavior (v_i) defined by $\int(v_i) = [0, v_0, v_1/2, v_2/3, ...]$ will be modeled by the following function:

```
integral (vs)    = 0:(integral' vs 1)

integral' (v0:vs) n = (v0/n):(integral' vs (n+1))
```

As an application, a first order system modeled by the differential equation ($E0$) $p' + p = 0$ with an initial condition $p(0) = 1$ can be represented by $p(n) = p(0) + \int p.dn$; and this expression can be modeled with the preceding function by : $p = add\ (value\ 0)\ (integral\ p)$. The function $value$ is then based on a new interpretation of $lift0$, i.e. value v = v:(lift0 0). The function add is based on the expression $lift2$ used for dataflows. As a result, the successive values returned by the Haskell interpreter is then $[1, -1, 1/2, -1/6, ...]$ that correspond to the analytical solution of the differential equation ($E0$) : $p(n) = \sum_i (-n)^i/i!$).

The approach followed for continuous systems with the definition of a modeling language and a functional metamodel will be considered for discrete state behaviors in the next section.

4.3. *State-Transition Systems*

Discrete behaviors are generally represented by State-Transition Systems (STS). An sts can be modeled by a graph (3.3) on which a node corresponds to a configuration (or State) of a system at a given instant, and a transition to a change of this configuration when an event occurs. Block Q on Figure 3 gives an example of sts with two States $Q = \{Up, Down\}$ and two events $E = \{min, max\}$. To be able to integrate discrete behaviors with continuous behaviors, an sts is modeled by a *block* with an input flow for events and an output flow for values generated by the sts. In the previous example, the input is y and the events are $\{min = y < 0.1,\ max = y > 0.9\}$; the output is a continuous value u such as $u = 1$ when the current state is Up, and $u = 0$ when the state is $Down$.

With the proposition, a functional metamodel for this definition of STSs will correspond mainly to a current state s and a transition function $t : State \times Event \rightarrow State \times Value$; i.e. considering the current state s and an input event e, $t(s, e)$ will return a couple (s', v) where s' is the next state and v the output value. This model has to be completed by a semantic function sem describing how the values of the input/output $flows$ and the evolution of the sts can be linked together. The functional metamodel proposed for STS is then defined in Haskell by the following code and a graphical representation of the functional metamodel is proposed on Figure 13.

```
data STS = STS State Transition
type Transition = (State,Event) -> (State,Value)

sem :: STS -> Flow a -> Flow b
sem (STS s t (F i is)) = F o os
  where (s',o) = t (s,i)
        os     = sem (STS s' t is)
```

As an illustration, the model below describes how the process Q of Figure 3 can be modeled with this new metamodel : t corresponds to the transition function, q to the sts with an input flow y, and u to the output flow that will be connected to the first order system presented in 4.1.

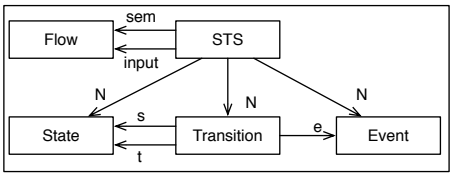

Fig. 13. Metamodel for state-transitions systems.

```
t :: Transition
t (0,y) = if y>0.9 then (1,0) else (0,1)
t (1,y) = if y<0.1 then (0,1) else (1,0)

q :: STS 0 t y
u = sem q
```

Putting together the previous model and the first order system, i.e. "u=step" is simply replaced by "sem q", will return the behavior of Figure 14.

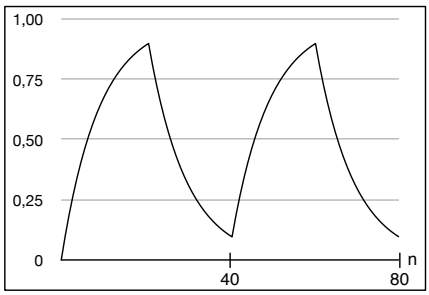

Fig. 14. Output y(n) for the example.

4.4. *Specification of temporal properties*

As a complement to the previous section, and to show the benefit brought by the concept of functional metamodeling, this part proposes a modeling language dedicated to requirements. Indeed, the development process for software is generally based on successive steps beginning with requirements (or expression of what a system must do/be), followed by design and tests (or formal proofs), and terminated by implementation and deployment on a target architecture.

Properties found in requirements can be formalized with temporal logic formulae ($Form$) - Clarke et al. (2000), and the following metamodel specifies a subset of the language used to express these formulae with: constant ($FTrue$), atomic propositions ($FProp$), classical operators (Not, Or) and temporal operators ($neXt$, $Until$). These elements are modeled by functions (e.g. $Not : Form \rightarrow Form$) that are defined in Haskell by the following code. As an example of transformation, the function $(sat\ p) : Form \rightarrow Bool$ checks if a path p satisfies a formula. A *path* models a particular behavior and is represented here by a sequence (3.2) of states where each state is labeled by a set of properties (represented here by strings). This model also describes how to build syntactic sugars (*globally* and *imply*) on the top of the previous elements.

```
type Property = String
type State = List Property
type Path  = List State
data Form  = FTrue | FProp Property
        | Not Form
        | Or  Form Form
        | X Form
        | U Form Form

sat::Path  -> Form      -> Bool
sat p       FTrue      = True
sat (ps:pss) (FProp p)   = contains p ps
sat p       (Not f)     = not (sat p f)
sat p       (Or f f')   =(sat p f)||(sat p f')
sat (ps:pss) (X f)       = sat pss f
sat p@(ps:pss) g@(U f f') = ((sat p f) &&
    (sat pss g)) || (sat p f')
sat _       _         = False

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

Figure 15 presents a graphical view of this metamodel. Form becomes a class with an operation $sat(t : Trace) : Boolean$ and $FTrue$, $Property$, Not, etc. become subclasses ; the implementation of *sat* is then described by the previous code/equations. This kind of diagrams establishes a bridge between the elements proposed and the standard MDE approach. As an application of this metamodel, the following model gives an example of test for the expression $(Push \Rightarrow X\ Move)\ U\ Stop)$.

```
path = [["Push"],["Move"],["Stop"]]
form = U (imply (FProp Push)
         (X (FProp Move)))
       (FProp Stop)

sat path form == True
```

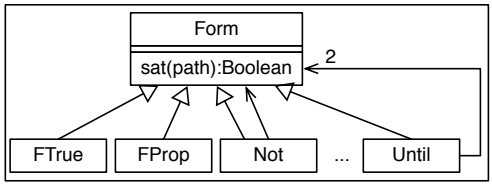

Fig. 15. Metamodel for LTL Formulae.

Thus, the functional metamodels proposed can be used to model and to simulate either continuous behaviors, or discrete behaviors, or continuous+discrete behaviors. However, these metamodels do not take into account implementation details such as the interactions with an environment or a target platform on which models will have to be deployed. To go beyond this limitation, the following section presents a functional metamodel for actions and interactions.

4.5. *Actions and concurrency*

The concept of *action*, found in imperative programming languages, can be modeled by functions transforming a *state*, i.e. the execution environment at a given instant, into another state plus a result. An operator called *bind* is then defined to compose actions sequentially and to retrieve a result ; *bind* is similar to the operator ($\circ$) used for function composition. Basic (and concrete) actions are defined: *get* the value of a variable from the execution context, and *put* of a value into an execution context. These elements are represented by the following functional metamodel where an execution context corresponds to a state defined as a list of couples ($Variable$, $Value$) and an action is a function returning a value of type a and a new state.

```
type State  = List (Variable,Value)
type Action = State -> (Value,State)

get v vs   = (findIn v vs,vs)
put v e vs  = ((),insert (v,e) vs)
bind p f c  = let (r,c')=p c in f r c'
```

From now, the blocks presented on sections 4.1 and/or 4.3 can be interpreted as (concurrent) *processes* ; a process being defined by a sequence of actions. These processes use the global state and shared variables to communicate and to synchronize. Indeed, grouping together two processes, where (*put* v *chan*) is defined on the first and (*get* v) is defined on the second, can be interpreted by a sending of the value v into the communication channel *chan*. For instance, the processes P and Q shown on Figure 3 communicate by the way of two shared variables u and y. As another example, the producer-consumer model below describes a process *prod* (and a block which generates a ramp, i.e. a flow: v=1:2:3:...) that emits values on *chan*, and a process *cons* which reads these values and displays them.

```
type Process = List Action

prod,cons :: Process
prod chan = p 0
  where p v = (put chan v) : (p (v+1))
cons chan = display : cons
  where display = bind (get chan) (put output)
```

The formalization of processes by the way of actions makes it possible to define a particular execution models such as interleaving ; and a function *par* below proposes an implementation of this basic model of concurrency. The function *sequence* executes a sequence of actions (i.e. a single process) and the function *run* is a particular use of *sequence* that executes n actions of a process. The function *main* is an implementation model for the system $producer||consumer$: this function corresponds to the interleaving of two concurrent processes/blocks that exchange values by the way of channels (and shared variables) u and y. Calling "run 10 $main$" on an initial state ([ ]) returns the new state [(output,012345),(chan,5)] after executing 10 actions.

```
parallel :: Process -> Process -> Process
parallel (x:xs) ys = x:(parallel ys xs)

sequence :: Process -> Action

run :: Integer -> Process -> Action
run n p = (sequence (take n p)) []

main :: Process
main  = parallel prod cons

run 10 main == [(output,012345), (chan,5)]
```

With the various functional metamodels, a software system would be defined by a set of *processes* that are modeled at a logical level by continuous or discrete *blocks* and $flows$ (of values or events), and at an implementation level by *actions* and *processes*. Figure 16 presents a graphical representation of the functional metamodel usable to describe implementation models (i.e. models that facilitate the deployment of the elements of parts 4.1 and 4.3 into a target architecture).

At this point, the article has proposed the concept of functional metamodeling that is detailed in part 3 and that corresponds to the right part of Figure 1. The concept has been applied in this part to specify and to integrate modeling languages used in systems and software ; this corresponds to the middle part of Figure 1. The next part will use these functional

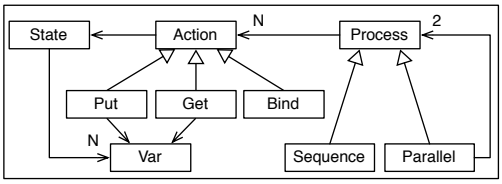

Fig. 16. Functional metamodel for actions/processes.

metamodels in a more complete example and will corresponds to the left part of Figure 1.

5. Application to a legged robot

5.1. *Description of the system*

The functional metamodels are used to describe the control of the locomotion of a legged robot (figure 17), Thirion and Thiry (2002). The main blocks used to control this system are given by Figure 18. The displacement of each leg i (represented by dx_i) is computed using the geometric and kinetic models f_i and the global speed vectors of the platform $(\vec{v}, \vec{\varpi})$. This displacement is used by the leg controller a_i, which provides the position x_i of a leg. The position depends on the operating mode: a leg can be on ground and push the platform, a leg can be up and move, or a leg can wait to move. Indeed, a control signal s_i is used to synchronize the legs and to keep the platform stable. More precisely, a leg i can move if, and only if, the legs $(i \pm 1)$ *mod* 6 are not moving.

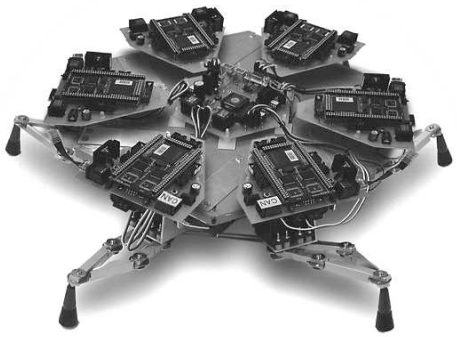

Fig. 17. Legged robot.

5.2. *Functional model*

A leg is defined by a continuous state corresponding to the position $P = \mathbb{R}^3$, and by a discrete state $E = \{Push,\ Wait,\ Move\}$. The synchronization between the legs is done using a signal S defined by two values : if leg i emits the event Can (resp. $Cant$) then leg $(i+1)$ *mod* 6 can do the transition $Wait \rightarrow Move$ (resp. can not do the transition). The state-transition model describing the discrete behavior is represented on the upper part of Figure 18. The initial state is $Push$ and the transitions have three elements $(in1, in2)/out$ where $in2$ is the signal coming from leg $i-1$, and out is the signal emitted to leg $i+1$. $In1$ is used to control the position x and can take three values: min when a leg can not push anymore (i.e. $min = reachMin\ x$), max when a leg can not move anymore (i.e. $max = reachMax\ x$), and $\_ = not\ (min\ or\ max)$. Using the functional metamodel for STS (4.3), the discrete behavior of a leg is represented by an sts composed by the initial state $Push$ and the transition function t (see Haskell model below). Let's keep in mind that a transition function takes two parameters (:the current state and the input event - here $(x, Can/Cant)$), and returns two elements: the new state and the output values - here (Can/Cant,k_x). The values k_x are used to control the trajectory of a leg (i.e. to compute new value of x for each leg i) ; they are detailed with the dataflow model in the next section.

```
-- States
type P = (Float,Float,Float) -- Continuous
data E = Push | Wait | Move -- Discrete

-- Events
data S = Can | Cant
reachMin, reachMax :: P -> Bool

-- State-transition model
t :: Transition
t Push (x,_)   = if reachMax x then (Wait,(Can,k1))
              else (Push,(Can,k2))
t Wait (x,Can ) = (Move ,(Cant,k3)
t Wait (x,Cant) = (Wait,(Can ,k1)
t Move  (x,_  )= if reachMin x then (Push,(Can,k2))
              else (Move,(Cant,k3))

sts = STS Push t
```

At this point, the function sem (4.3) is used to transform STSs into blocks (4.1) and to compute the synchronization signals s_i (see below). The only thing that remains to be done is to precise how continuous state x_i is computed and what the use of the parameters k_x is. The trajectory of a leg can be split into three equations $x_{i+1} = f_k(x_i)$, i.e. one equation for each discrete state, and these equations can be merged with $f(x) = k_p.f_{Push}(x) + k_w.f_{Wait}(x) + k_m.f_{Move}(x)$ with $k_x \in \{0, 1\}$. Thus,

the values k_x computed by each *sts* correspond to $k_1 = (k_p, k_w, k_m) = (0, 1, 0)$, $k_2 = (1, 0, 0)$ and $k_3 = (0, 0, 1)$. Using the function *map* on lists (3.2), it is possible to extract these values from the flow $k_i = (k_p, k_w, k_m)^*$ returned by each sts, e.g. $k_p^* = map\ \pi_1\ k_i$ with $\pi_1 : (A, B, C) \rightarrow A$. Finally, the computation of the flow x is realized by a dataflow model composed with *Gain*, *Add* and *Delay*. The functions $fPush$ and $fMove$ correspond to geometric transformations and are not detailed here ; the function $fWait$ shows that when a leg is waiting, its position $p = (x, y, z)$ does not change.

```
-- Dataflow model
-- Transformation of STSs into blocks
(s1,k1) = sem sts (x1,s6) -- leg 1
(s2,k2) = sem sts (x2,s1) -- leg 2
(s3,k3) = sem sts (x3,s2) -- leg 3
...
(s6,k6) = sem sts (x6,s5) -- leg 6

-- Extraction of the control flows
k1p = map pi1 k1
k1w = map pi2 k1
k1m = map pi3 k1  -- same thing for k2,k3,...,k6

-- Continuous behavior and dataflow model
x1  = Delay x0 x1'
x1' = Add (Gain kp (map fPush x1))
          (Add (Gain kw (map fWait x))
               (Gain km (map fMove x)))

fPush, fWait, fMove :: P -> P
fWait p = p
```

From now, the continuous state, i.e. the position x of each leg i at the instant n, can be obtained with (xi !! n). The discrete state can be obtained in a similar manner with (ki !! n) and using the preceding rules (i.e. if the value is kp=(0,1,0) then the leg is pushing, kw then the leg is waiting, km then the leg is moving).

5.3. *Results*

Figure 19 presents the behavior of the robot and the discrete state of the legs. A low level corresponds to $Push$, a middle level to $Wait$ and a high level to $Move$. At the origin, all legs push to their posterior extreme position (x_{max} on the Haskell model). Reaching x_{max}, the legs 1-3-5 go into $Wait$ mode while the legs 2-4-6 go into $Move$ mode. Legs 2-4-6 move to their anterior extreme position (x_{min} on the Haskell model). Reaching x_{min}, the legs 2-4-6 go into $Push$ mode, while the legs 1-3-5 go into $Move$ mode. At this point, the system has a characteristic behavior (called tripod walking) where each group of legs (i.e. 1-3-5 and 2-4-6) alternates between $Push$ and $Move$.

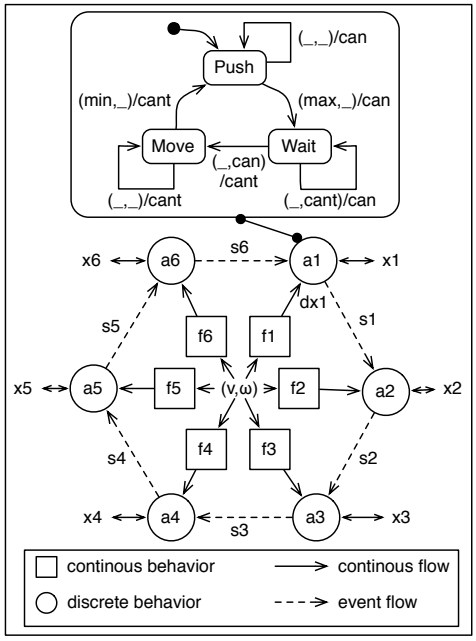

Fig. 18. Model of the control architecture.

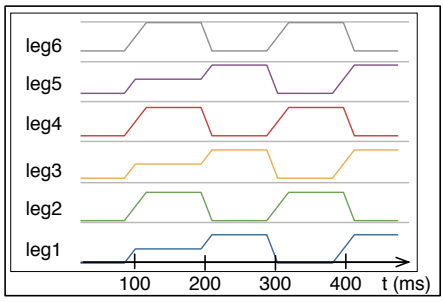

Fig. 19. Behavior of the legged robot.

Thus, the functional metamodels defined in part 3, and the modeling languages captured in part 4, are well adapted to model/simulate systems that can have a complex behavior or structure.

6. Conclusion

This paper has proposed the concept of "functional metamodeling" and its application to a set of modeling languages for systems and software. As a final result, these languages have been used to model a control architecture for a legged robot with continuous and discrete behaviors that are implemented

as concurrent processes. More precisely, with the concept, modeling elements are described by sets of functions to create models, and a model of generic transformation that is also a function. Thus, formalisms and their relations can be specified using a single but formal concept: the one of "function". This offers many advantages that are generally unknown. For instance, functions can be composed and passed as parameters of other functions (2.3). So, the paper has explained how these capabilities can be used to capture and to integrate modeling languages (3) also called metamodels in MDE.

In the context of systems where various points of view have to be considered, functional metamodels are proposed to allow the description and the composition of continuous behaviors (with numerical and analytical models), of discrete behaviors (with state-transition models and a model of temporal logic used for specifications) and of concurrent behaviors (with models of actions and processes). An other advantage of the concept is to be supported by functional programming languages. In particular, the elements proposed have been implemented into Haskell. To show the benefits of the resulting framework, the example of a control software system is developed in part 5. Thus, as summarized on Figure 1, functional metamodeling is used at three levels with the systems, the languages and tools used to model systems, and the means to specify these languages and tools.

Functional metamodeling offers an alternative to the standards proposed by the MDE community that is better adapted to mathematical descriptions. In particular, functional metamodeling makes equational reasoning possible and the first perspective considered now will consist in studying how this capability can be used to prove model equivalences or how properties of a model can be preserved by a transformation, for instance. As a second perspective, studying how functional metamodeling can be applied to more software-oriented elements is considered. In particular, MDE adds concrete syntaxes to models, metamodels and models of transformations. The use of functional metamodels to specify textual and/or visual views as a front end to the (meta)models presented here will give a better understanding of the advantages/limitations of the concepts.

References

Bird, R., 1998. Introduction to Functional Programming using Haskell, 2nd Edition. Prentice Hall PTR.

Breton, E., Bezivin, J., 2001. Towards an understanding of model executability. In: International Conference on Formal Ontology in Information Systems. pp. 70–80.

Clarke, E., Grumberg, O., Peled, D., 2000. Model Checking. The MIT Press.

Denckla, B., Mosterman, P., 2005. Formalizing causal block diagrams for modeling a class of hybrid dynamic systems. In: IEEE Conference on Decision and Control.

Deursen, A., Klint, P., Visser, J., 2000. Domain-specific languages: an annoted bibliography. In: SIGPLAN Notices. Vol. 35. pp. 26–36.

Doets, K., van Eijck, J., 2004. The Haskell Road to Logic, Maths and Programming. College Publications.

Dubey, A., 2005. Metamodel based language and computation platform for algorithmic analysis of hybrid systems. Ph.D. thesis, Faculty of the Graduate School of Vanderbilt University.

Favre, J.-M., 2004. Towards a basic theory to model driven engineering. In: Workshop on Software Model Engineering, WISME 2004, joint event with UML2004.

Guttag, J. V., 2002. Abstract data types and the development of data structures. Software pioneers: contributions to software engineering, 453–479.

Harisson, W., 2006. The essence of multitasking. In: Springer (Ed.), Lecture Notes on Computer Science. Vol. 4019. pp. 158–172.

Henzinger, T. A., Sifakis, J., 2007. The discipline of embedded systems design. Computer 40 (10), 32–40.

Hudak, P., Courtney, A., Nilsson, H., Peterson, J., 2003. Arrows, robots and functional reactive programming. In: Springer (Ed.), Lecture Notes on Computer Science. Vol. 2638. pp. 158–172.

Hudak, P., Hughes, J., S., P.-J., P.A., W., 2007. History of haskell: Being lazy with class. In: 3rd ACM Sigplan History of Programming Language. pp. 1–55.

Jantsch, A., Sander, I., 2005. Model of computation and languages for embedded systems design. In: Computer and Digital Techniques. Vol. 152. pp. 114–129.

Lee, E., Varaiya, P., 2003. Structure and Interpre-

tation of Signals and Systems. Addison Wesley.
Maler, O., 1998. A unified approach for studying discrete and continuous dynamical systems. In: 37th IEEE Conference on Decision and Control. Vol. 2. pp. 2083–2088.
Mathaikutty, D., 2005. Functional programming and metamodeling frameworks for system design. Ph.D. thesis, Faculty of Virginia Polytechnic Institute and State University.
Mc Ilroy, M., 1998. Functional pearls: Power series, power serious. Journal of Functional Programming 1 (1), 1–13.
Mellor, S., Clark, A., Futagami, T., 2003. Guest editors introduction: Model-driven development. In: IEEE Software. Vol. 20. pp. 14–18.
Thielemann, H., 2004. Audio processing using haskell. In: 7th International Conference on Digital Audio Effects.
Thirion, B., Thiry, L., 2002. Concurrent programming for the control of hexapod walking. ACM Ada Letters 22 (1), 17–28.
Thiry, L., Thirion, B., 2008. Functional (meta)models for the development of control software. In: International Federation of Automatic Control, IFAC'08, Seoul.
Uustalu, T., Vene, V., 2006. The essence of dataflow programming. In: Springer (Ed.), Lecture Notes on Computer Science. Vol. 4164. pp. 135–167.
Vangheluwe, H., de Lara, J., 2003. Foundations of multi-paradigm modeling and simulation: computer automated multi-paradigm modelling. In: 35th conference on Winter simulation: driving innovation. pp. 593–603.
Varro, D., 2002. Towards symbolic analysis of visual modelling languages. In: International Workshop on Graph Transformation and Visual Modelling Techniques. pp. 57–70.
Varro, D., Pataricza, A., 2003. Vpm: Mathematics of metamodeling is metamodeling mathematic. Journal of Software and Systems Modelling, 1–24.
Wadler, P., 1996. The essence of functional programming. In: 19th Symposium on Principles of Programming Languages.
Walter, R., 1992. Categories and Computer Science. Cambridge University Press.
Winskel, G., Nielsen, M., 1995. Models for concurrency. In: Press, O. U. (Ed.), Handbook of Logic in Computer Science. pp. 1–148.

Laurent Thiry is an assistant professor at the Université de Haute Alsace (Mulhouse, France). He passed his PhD in software engineering, entitled "Models, metamodels and behavioral objects for complex dynamical systems", at the Université de Haute Alsace in 2002. His research interests include software engineering, model driven engineering, and formal methods applied to control.

Bernard Thirion is a professor of software engineering at the Université de Haute Alsace (Mulhouse, France). He passed his Ph.D. (1980) in electronics and instrumentation at the Université de Haute Alsace, France. In 1993 he received a French Habilitation for supervising research activities in software engineering at the Université de Haute Alsace, France. His current research interests include software engineering and software architectures, UML and object-oriented modeling and metamodeling, design patterns, design of concurrent and real-time systems.