Functional metamodels for Discrete Event Systems

Laurent Thiry
ENSISA, 12 rue des frères Lumière - 68093 Mulhouse, France
E-mail: laurent.thiry@uha.fr

Abstract: The paper presents a framework dedicated to the analysis and design of Discrete Event Systems. More precisely, it proposes a set of generic functions that makes it possible the integration of various modeling languages and tools. As an illustration, the paper explains how to specify behaviors by using the Eclipse Modeling Framework, validate these behaviors by using the model checker LTSA, run/implement the models into the general functional programming language Haskell. The framework is based on the concept of “functional metamodeling” on which modeling elements are specified by constructor functions, and are related by using higher-order functions. As an advantage of this formalization, one can prove that models that pass from one tool to another stay equivalent, i.e. essential information is not lost.

Keywords: Discrete Event Systems, Metamodeling, Functional Programming

1. INTRODUCTION

The development of Discrete Events Systems (DES) is generally based on many formalisms, methodologies and tools. For example, the specification of system properties can be expressed using temporal logic, the design can be based on automata or process algebra, and the deployment on a target architecture can require specific programming languages. Model Driven Engineering (MDE), a branch of software engineering, proposes concepts and tools to simplify development processes. In particular, MDE proposes to specify model types with for example the Meta Object Facility (MOF), languages such as Query-View-Transform (QVT) to relate models, languages to add constraints or semantics (e.g. OCL), and languages to add visual or textual representations to a model, etc.

As an alternative to all these standards, and to relate more easily MDE and DES, the paper proposes to use a single concept: the one of “function”. More precisely, it shows how higher-order functions, i.e. functions having other functions as parameters or as a result, can represent metamodels for logic, automata or IO, and their transformations. These elements are thus expressed in a compact and formal manner, and they can be translated and used in any functional programming language (e.g. Haskell).

The paper is divided into three parts. The first part gives an overview of DES and MDE. The second part presents the concept of “functional metamodeling” and its application to DES. The last part summarizes the main points discussed in the article.

2. DES & MDE

2.1 Discrete Event Systems

A system $S$ is generally represented by a state space $X$, an initial state $x_0 \in X$, a set of events $\Sigma$ and a transition function $f : X \times \Sigma \rightarrow X$. The behavior of the system is then a sequence of states $x$ defined by $x(n + 1) = f(x(n), \sigma(n))$ with $x(0) = x_0$, $n$ is the discrete time, and $\sigma(n)$ a sequence of events, Cassandras and Lafortune (2006). The function $f$ is most often represented by a graph whose nodes are the elements of X, and the arcs are possible transitions (e.g. Figure 1).

Thus, $S = (X, x_0, f)$, Lee and Varaiya (2003).

A system is generally composed with many subsystems that have to evolve concurrently and to synchronize, i.e. $S = \Pi_i S_i = \Pi_i (X_i, \Sigma_i, f_i)$. The state of the system is then defined by the product of the states of each component, e.g. $X = X_1 \times X_2$ with two subsystems. Compound systems have generally a great number of states and, consequently, they require the use of dedicated tools to analyze/simulate them, with for example the Labeled Transition System Analyzer (LTSA), Magee and Kramer (2006). To use this tool, one has to learn its grammar. For example, a simplified grammar for LTSA is given below. A system is defined by "SYS = " followed by a list of subsystems identifiers id separated by parallel composition (\(|\)). A subsystem is defined by an identifier (DEF) and a state (ST). A state can have zero outgoing transition (Stop), one transition (evt \(\rightarrow\) ST), or many transitions separated by (\(|\).

SYS = "SYS = (" SUB ")."
SUB = id SUB'
SUB' = "|" SUB | e
DEF = id "=" ST DEF'
DEF' = "," DEF | "." 
ST = "Stop" | id "->" ST | id | "(" ST ST' ")"
ST' = "|" ST | e

As an illustration, a system composed with : 1) a producer that produces values and writes/puts them into a buffer, 2) a consumer that reads/takes values from a buffer and
transforms them, and 3) a buffer, can be modeled in LTSA by the following code. The (not obvious) graph for the whole system SYS, and also the interpretation of the following expression, is then represented in Figure 1.

\[
\begin{align*}
\text{Producer} & \rightarrow \text{produce} \rightarrow \text{put} \rightarrow \text{Producer}. \\
\text{Buffer} & \rightarrow \text{put} \rightarrow \text{take} \rightarrow \text{Buffer}. \\
\text{Consumer} & \rightarrow \text{take} \rightarrow \text{transform} \rightarrow \text{Consumer}. \\
\text{SYS} & \rightarrow (\text{Producer} \parallel \text{Buffer} \parallel \text{Consumer}).
\end{align*}
\]

![Diagram of the producer-buffer-consumer system](image)

Fig. 1. Model of producer-buffer-consumer system.

The states of a system can be attached to a set of properties \( P \), i.e. by considering a function \( ps : X \rightarrow P(P) \) with \( P \) the powerset. Thus, a system \( S \) satisfies a property \( p \in P \) at a given instant \( n \) if \( p \in ps(x(n)) \). In general, properties depend on time and particular operators have to be introduced. For example, Linear Temporal Logic, see Clarke et al. (2000), uses the operator next \( N \) whose semantic is \( N(p) \equiv p \in ps(x(n+1)) \), or the operator globally \( G \) whose semantic is \( G(p) \equiv \forall n \cdot p \land N(G(p)) \).

Thus, by defining a trace as a sequence of states \( x : \mathbb{N} \rightarrow X \), a system \( S \) satisfies a property \( p \) if all its possible traces satisfy \( p \). Proofs can be done by using a model checker such as LTSA previously mentioned. If a system does not satisfy a property then this tool returns a trace \( x_1 \) usable to improve this system (by modifying \( f \)). As an illustration, a example of property satisfied by the producer-buffer-consumer system is: \( G(N(\text{produce})) \) which says that values are produced infinitely.

An alternative to model checking is model testing or simulation. Generally, a test is realized by: 1) computing possible events \( pe \) and fireable transitions from initial/current state, i.e. computing the elements \( e \) such as \( (x_0, e) \in \text{dom}(f) \); 2) choosing one event \( e \in pe \) and replacing current state by \( f(x_0, e) : 3 \) going back to step 1. A test can serve, for instance, to correct a system going to a deadlock defined by \( pe = \emptyset \).

Now, a real system interacts with its environment using Input and Output (IO), i.e. reading/testing some variables and setting other ones. IO are responsible of state changes and can be integrated into the previous models by considering \( X = \Sigma_I \cup \Sigma_O \). This leads to two kinds of events with events received \( e ! \) and events emitted \( e ! \); the notation \( (?, !) \) comes from Communicating Sequential Processes (CSP) , Hoare (1978). With this change, a (sub)system can be implemented using ifThenElse constructs found in any programming language. For example, transitions like \( (x_1, a^?) \rightarrow x_2 \) and \( (x_2, b!) \rightarrow x_3 \) could be translated to the following code. This translation is not unique and other (more sophisticated) codes are possible.

\[
x = x_0;
\]

while (true)
  if (x==x1) then {
    if (read(a)) then { write(b); x=x3; }
  } else ... // other transitions from x1
else ... // other states

Thus, as for using LTSA, the implementation of a DES model requires the knowledge of 1) another language and 2) mapping rules to pass from a transition function \( f \) to a sequence of statements. More generally, Discrete Event Systems (DES) are based on a family of interconnected languages to specify their properties (e.g. with LTL), to represent them (e.g. with a graph representing \( f \) or LTSA statements), to implement them (e.g. instructions in a particular programming language), etc. The next section presents Model Driven Engineering (MDE) and the means proposed to pass more easily from one modeling language to an other, and thus to reduce efforts met in the development of DES.

### 2.2 Model Driven Engineering

A model is an abstract representation of a system (or a process), Favre (2004). Generally, this representation corresponds to either a graph or an expression in a particular language. Graphical models specify entities \( E \) that compose the system, and relations \( R \) between these entities. The types of entities \( TE \) and the types of relations \( TR \) can be represented by a graph too, and this latter is then called a metamodel. Metamodels are themselves specified using a specific language with, in particular, the Meta Object Facility (MOF) . For example, Figure 2 proposes a possible metamodel for DES and LTL. Each box corresponds to a modeling concept (e.g. State) and each arrow corresponds to a relation (e.g. src : Transition \( \rightarrow \) State). The particular arrow \( > \) is used for classifications, e.g. a basic property \( Prop \) is an LTL element. It is important to notice that a metamodel corresponds to the abstract syntax of a language: the concrete syntax, e.g. what the shape used to represent a state is, must be specified with another language.

![Metamodel for DES and LTL](image)

From this, MDE proposes generic tools to be configured with metamodels, and that can be used to build models. An example of generic tool is the Eclipse platform with the Eclipse Modeling Framework (EMF) plug-in . The persistence and the exchange of (meta)models is then based on the eXtensible Markup Language (XML) standard .

1. www.omg.org/mof/
2. www.eclipse.org/modeling/emf/
3. www.w3.org/standards/xml/
In particular, the metamodel for DES and the model of the producer correspond to the following code (which is a simplified EMF model). This code is yet another way to express a transition function.

```
<-- metamodel for DES (conformed to EMF) -->
<Package name="DES">  
  <Class name="State"/>  
  <Class name="Event"/>  
  <Class name="Transition">  
    <Reference name="src" Type="State"/>  
    <Reference name="dst" Type="State"/>  
    <Reference name="evt" Type="Event"/>  
  </Class>
</Package>

<-- model for Producer (conformed to DES) -->
<DES>  
  <State>E0</State>  
  <State>E1</State>  
  <Event>produce</Event>  
  <Class name="Event"/>  
  <Class name="State"/>  
  <Reference name="evt" Type="Event"/>  
  <Reference name="src" Type="State"/>  
  <Reference name="dst" Type="State"/>  
</DES>
```

A model transformation is a set of mapping rules between two metamodels, i.e. $T : MM \rightarrow MM'$, such as if $m$ is an instance/value of $MM$ then $T(m)$ is an instance of $MM'$. Czarnecki and Helsen (2003). As for metamodels, there are dedicated languages to describe transformations with in particular the Query-View-Transform (QVT) standard to relate MOF metamodels. Transformations can be classified into two categories depending of the type of models considered. In particular, the main kinds of transformations are: 1) Text $\rightarrow$ Graph or inversely Graph $\rightarrow$ Text, and 2) Graph $\rightarrow$ Graph. The firsts are studied by language theories and the lasts are detailed by graph grammars, Krewski et al. (2006).

Now, models, metamodels and transformations can be specify using other formalisms. In particular, the next section show how to restrict the modeling conpects to the one of "function" that is found in any functional (programming) languages, and what are its benefits in the field of Discrete Event Systems.

3. FUNCTIONS AND APPLICATION

3.1 Proposition

A function $f$ is a relation between two sets $A,B$, i.e. $f \subset A \times B$. Its main property is: if $(x,y) \in f$ and $(x',y') \in f$ then $x = x' \Rightarrow y = y'$. The notation used for $(x,y) \in f$ is then replaced by $f(x) = y$, and $A \rightarrow B$ (or $B^A$) corresponds to the set of all functions from $A$ to $B$. Some particular functions are: identity functions $1_A : A \rightarrow A$ with $1_A(x) = x$, constant functions also called values $c : 1 \rightarrow A$ where $1$ is a singleton (e.g. $1 = \{\emptyset\}$), projections $\pi_i : A_1 \times A_2 \times ... \rightarrow A_i$ with $(\times)$ the cartesian product, or again injections $i_j : A_j \rightarrow A_1 + A_2 + ...$ with $(+)$ the disjoint union. Functions can be passed as parameter or as result to other functions. These functions are then called "higher-order" functions. Well known examples are given by the operator $\text{derivate} : (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow (\mathbb{R} \rightarrow \mathbb{R})$, and function composition $(\circ) : (B \rightarrow C) \times (A \rightarrow B) \rightarrow (A \rightarrow C)$ with $(f \circ g)(x) = f(g(x))$, but other examples are possible, Backus (1978).

The concept of "functional metamodeling" has been introduced in Thiry and Thirion (2008). It consists in describing metamodels and their transformations in a compact and formal manner by using functions. As an illustration, a functional metamodel for sequences/lists $A^N$ (or $L(A)$) can be specified with two functions: $\text{nil} : 1 \rightarrow A^N$ that corresponds to the empty list, and $\text{cons} : A \times A^N \rightarrow A^N$ that adds an element at the head of a list. Now, every function $A^N \rightarrow B$ could be defined by using the higher-order function $f(n,c)$ defined by: $f(n,c) \circ \text{nil} = n$ and $f(n,c) \circ \text{cons} = c \circ 1_A \times f(n,c)$ with $n : 1 \rightarrow B$ and $c : A \times B \rightarrow B$. More simply, this function transforms recursively $\text{nil}$ into $n$, and each $\text{cons}$ into $c$. The function $f$ can then be used to express most of the functions on $A^N$. For example, $\text{map}(h) = f(\text{nil},\text{cons} \circ h \times 1)$ applies $h$ to each element of a list, the concatenation of two lists $(as,bs)$ is $\text{cat} = f(bs,\text{cons})(as)$, and the concatenation of a list of lists is $\text{concat} = f(\text{nil},\text{cat})$. Thus, the use of functions leads to compact and formal description of metamodels and of transformations. A more complete description of functional metamodeling can be found in Thiry and Thirion (2009), and Thiry (2010).

An other advantage brings by the concept is that the previous equations can be interpreted in any functional programming language. For instance, the following code gives the translation of lists metamodel into Haskell, Hutton (2007). The identity function corresponds to id and function composition $(\circ)$ to $(\cdot)$. The metamodel corresponds to a (data)type definition (data $L a$) and the functions usable to construct values/models. Thus, the sequence $s = [1,2]$ will corresponds to $\text{Cons}(1,\text{Cons}(2,Nil))$, and the equations above make possible the proof that $\text{cat}(s,s) = [1,2,1,2]$.4

```
data L a = Nil | Cons (a,L a)

f(n,c)(Nil) = n
f(n,c)(Cons(a,b)) = c(a,f(n,c)(b))
map(h) = f(\text{nil},\text{Cons} . \text{fmult} (h,\text{id}))
where \text{fmult}(f,g)(x,y) = (f(x),g(y))
\text{cat}(as,bs) = f(bs,\text{Cons})(as)
```

As an other example, a (partial) functional metamodel for LTL is defined below. The code gives too the higher-order function $\text{ltl}(p,u,n) : \text{LTL} \rightarrow B$ and an application with the function $\text{ser}$ that serializes a model, e.g. $\text{ser}($Until($\text{prop}("on"),\text{prop}("broken")))$ will return "on U broken", and the function $\text{sat(isfy)}$ usable to check that a trace $t$ satisfies an LTL property. A trace is here a sequence of (basic) property lists but more complex data structures such as derivation trees can be considered. The function $\text{elem}$ tests if an element is into a list, $\text{hd}$ returns the head of a list, $\text{tl}$ returns its tail, and the operators (\&\&) and (\|) correspond to logical conjunction and disjunction. Thus, functional metamodels offer the mean to express into a single language metamodels (e.g. LTL), their generic transformations (e.g. $\text{ltl}$), and their semantics (e.g. $\text{sat}$).

```
data LTL = Property (String) | Next (LTL)
```

4 www.omg.org/spec/QVT/1.0/
\[
\text{lgl}(p,n,u)(\text{property}(x)) = p(x) \\
lgl(p,n,u)(\text{next}(x)) = n(lgl(p,n,u)(x)) \\
lgl(p,n,u)(\text{until}(x,y)) = \ldots \text{ if } (\text{isletter}(\text{hd}(s))) \text{ then } [(\text{hd}(s),\text{tl}(s))] \text{ else } [] \\
\text{char}(c) = \begin{cases} x \leftarrow \text{letter} ; & \text{if } (x == c) \text{ then return } c \text{ else } [] \\
\end{cases}
\]

From a more theoretical point of view, the elements presented can be detailed by introducing the concept of "functors" found in Category Theory, Barr and Wells (1990). Functors are fundamentally parametrized datatypes (e.g. \(L(A)\)) associated to a particular function (see \(\text{map}(h)\)) having some properties (e.g. \(\text{map}(h \circ h') = \text{map}(h) \circ \text{map}(h')\)). With this definition, \(LTL\) do not define a functor: to become a functor, \(LTL\) has to be parametrized by using for instance \(A\) as the type of properties, and by replacing \(\text{String}\) by \(A\) in the constructors. Thus, the previously defined metamodel is simply \(LTL(\text{String})\), but other metamodels are now possible, e.g. \(LTL(\text{String}^A)\) that can be more interesting to manage sets of properties into \(LTL\) formula. This last expression shows too that functors can be composed. More generally, functions can have more than one parameter with in particular \((A \times B)\) that can be associated to the higher-order function \((f \times g) : (A \times B) \rightarrow (A' \times B')\); see function \(\text{fmult}\) above.

As a sample use, the metamodel for \(\text{DES}\) (Figure 2) can now be represented by \(\text{DES}(S,E) = L(S \times S \times E)\) with \((S,E)\) the type of States and Events. With \(s : S \rightarrow S'\) and \(e : E \rightarrow E'\), \(\text{map}(s \times s \times e)\) will be a transformation \(\text{DES}(E,T) \rightarrow \text{DES}(S',E')\). As an application, a relabeling of \(h : E \rightarrow E'\), usable to synchronize a \(\text{DES}\) with another one (see Part 2.1), will correspond simply to \(\text{DES}(id, h)\).

Functors \(F(A)\) can be extended with two functions called "return" or \(\eta : A \rightarrow F(A)\), and "join" or \(\mu : F(F(A)) \rightarrow F(A)\) (see concat). \(F, \eta, \mu\) corresponds to a monad and makes possible the use of two particular notations called "comprehension" and "do-notation", Wadler (1990). These notations can then be used to increase readability or to facilitate the implementation of the models. For instance, \(\text{map}(h)(xs)\) could be written either as a comprehension \([h(x)]x \in xs\), or as a statement do \{xs<-xs; return h(x)\}. As an illustration of list comprehensions, by considering the projection functions \(\pi_1(s,d,e) = s\) and \(\pi_2(s,d,e) = d\), then the states of a \(\text{DES}\) will be obtained with \(\text{states}(\text{des}) = \text{cat}(\pi_1(\text{des}), \pi_2(\text{des}))\). The add of \(\tau\) transitions for each state (Figure 3-b) will be obtained with \(\text{addtau}(\text{des}) = \text{cat}((s,s',\tau) | s \in \text{states}(\text{des}))\), and the cartesian product of two \(\text{DES}\) with \(\text{mult}(\text{des}, \text{des}) = [(s,d'), (d',e')] | (s,d,e) \in \text{des}, (s',d',e') \in \text{des}'\).

As an illustration of the "do" notation, the producer model defined in Part 2.1 can be simulated by using the following code. Actions in Haskell are modeled with the datatype \(\text{IO}\) and they can be composed with the "do" notation.

\[
\text{producer} = \text{s0} \\
\text{s0} = \text{do} \{ \text{c} \leftarrow \text{getChar} ; \\
\text{if} (\text{c} == \text{p}) \text{ then s1} \\
\text{else quit } \} \\
\text{s1} = \text{do} \{ \text{c} \leftarrow \text{getChar} ; \\
\text{if} (\text{c} == \text{t}) \text{ then s0} \\
\text{else quit } \} \\
\text{quit} = \text{return } () \\
\]

3.2 Application: EMF ↔ Haskell ↔ LTSA

Part 2.1 has shown that the use of tools requires the knowledge of their grammar. A functional metamodel, and a functor, to deal with grammars is \(I(A) = \text{String} \rightarrow (A \times \text{String})\). In this expression, a text represented by a list of characters (or \(\text{String}\)) is transformed into a value of type \(A\) extracted from a text together with the rest of the text to interpret. As an illustration, a letter could be obtained and represented by the following code. Next, combinators can be defined for: sequential compositions with \(\text{bind}\), alternatives/choices with the infix operator (\(|\) ), and repetitions \(\text{rep} : I(A) \rightarrow I(A^0)\). These higher order functions, and the monad operators for \(I\), are given below. As an example, \(\text{rep}('t')\) ("ab21") will be equal to "\(\text{"ab"","21"}\)".

\[
\text{letter}(s) = \text{if} (\text{isString}(s) \text{ then } [\text{hd}(s), \text{tl}(s)]) \text{ else } [] \\
\text{char}(c) = \text{do} \{ \text{x} \leftarrow \text{letter} ; \\
\text{if} (\text{x} == \text{c}) \text{ then return } c \text{ else } [] \} \\
\]

Fig. 3. Some operations on \(\text{DES}\).

In this example, the function \(\text{getChar}\) is an IO that reads a character from the keyboard and \(\text{putChar}\) writes a character to the screen. Characters used are \(\text{p, t}\) for produce, and \(\text{t}\) for transfer/put; other characters will force the system to stop/quit. Now, this implementation can be extended by using, for instance, communication channels, called \(\text{MVar}\) in Haskell, to get a concurrent and distributed implementation; this proposition will be discussed in the last part of the paper.
return(r)(s) = [(r,s)]
join (r)(s) = [z |(x,y)<-r(s), z<-x(y)]
bind(r)(f) = join(imap(f)(r))
where imap(f)(r)(s) = ... other a network (i.e. they can be executed on two different computers and channel is then a network address).

With the previous functions/metamodel, an interpreter for (a subset of) LTSA whose grammar has been presented in Part 2.1 is defined as follows:

ltsa = do { i<-rep(letter); char('='); d<-def; return (i,d)}
def = do { i<-rep(letter); (x,y)<-r(s), z<-x(y) return (x:xs) } | return []

An interpreter for XML/EMF models is defined in a similar manner. XML documents are basically tree/hierarchical structures composed with leaves representing texts and nodes having a name (eventually a set of properties) and children that are leaves or other nodes. This definition corresponds to a functional metamodel \( T(A, B) \) and two functions: \( \text{leaf} : A \rightarrow T(A, B) \) and \( \text{node} : B \times T(A, B)^{\text{N}} \rightarrow T(A, B) \).

For example, the EMF metamodel for DES presented in part 2.2 is partially represented by \( n \) in the following code. \( h(f,g) \) corresponds to the higher-order function \( T(A, B) \rightarrow C \) with a sample use given by \( ser \) that returns the code XML for a model; \( \text{e.g. ser(n)} \) is then the XML code presented in Part 2.2. The function \( xnl \) reads an XML model from a text and is the inverse of the previous function, i.e. \( xnl \circ \text{ser} \cong \text{id} \). Thus, \( (xml, ser) \) makes possible the exchange of models between the EMF tool and Haskell!

As a complement, transformations can be used to rearrange information inside a model (this, to optimize for instance either memory space required to store a model, or number of computations for the transformation functions). In particular, DES is equivalent to \( \text{DES}'(S, E) = (S \times (E \times S))^{\text{N}} \), i.e. a list of states \( S \) together with outgoing transitions. The transformation is defined by \( t'(des) = [(e, out(e, des)) | e \in \text{states}(des)] \) with \( \text{out}(e, des) = [(s,d,a) | (s,d,a) \in des, s = e] \). The transformation is invertible and this is why \( \text{DES} \) and \( \text{DES}' \) are said to be equivalent (i.e. no information are lost by passing from one representation to the other). However, the advantage of \( \text{DES}' \) is to be nearest of the prefix notation used by LTSA. More precisely, the following function returns the code of a DES model useable by this tool. As an illustration, the application of the function to the XML/EMF model of part 2.2, i.e. by using \( \text{ltsa} \circ t' \circ t \circ xnl \), is given by \( (E0, E1) \) below. In a similar manner, the functions presented are used to transform LTSA models into EMF models (with \( \text{ltsa and ser} \)), thus LTSA models can be used by the other tools that are already plugged at the top of EMF.

As a final extension to the framework proposed, DES models defined in Haskell (and coming from either LTSA or EMF) are extended with IO as mentioned in part 3.1; this to get the control software specified by the models. Two approaches are possible and consist in either generating executable code (see “do-notation”), or simulating/running a DES as explained in part 2.1. In this case, by defining two functions \( \text{run} : DES \rightarrow E \rightarrow DES \) that updates the current state of a DES when receiving an (input) event, and \( \text{out} : DES \rightarrow E \) that returns an (output) event attached to the current state, the execution of a DES is obtained using the following function.

\[
\text{execute}(\text{des}, i, o) = \begin{cases} 
\text{e} & \text{getMChar(i)}; \\
\text{putMChar(out(des), o);} & \text{execute(execute(des, i, o))}
\end{cases}
\]

In the preceding code, \( (i, o) \) correspond to communication channels (see MVar in Haskell) with for example (keyboard/stdin, screen/stdout) addresses. As an application, the following code presents a possible implementation of the producer-consumer model (where \( \text{forkIO} \) puts a processes into parallel of the current execution flow). Thus, the producer and the consumer are executed concurrently, and they can be easily distributed other a network (i.e. they can be executed on two different computers and channel is then a network address).
main = do {
    channel <- newMVar
    forkIO(execute(producer,stdin,channel));
    execute(consumer,channel,stdout) }

Last but not least, the functions presented in the article are currently integrated into the TUCS\(^5\) project whose purpose is to propose a methodology/tool dedicated to the specification and the design of safe railway systems. In the project, the functions are the foundation of the UML2LTS tool that makes possible: 1) the specification of behaviors as UML statecharts (similar to the EMF model used for DES); 2) the checking of the models properties with LTSA; 3) the simulation of the models with Haskell (as explained above) with 3D animations as illustrated in Figure 4.

Fig. 4. Example of simulated system.

4. CONCLUSION

The paper has proposed a functional interpretation of MDE concepts applied to the specification, the validation and the implementation of Discrete Event Systems. More precisely, with the proposition, a metamodel is represented by an expression such as $\text{DES}(S,E) = (S \times S \times E)^n$ associated to an higher-order function usable to define model transformations, and to obtain various representation of a model for various tools (EMF, LTSA, Haskell, etc.).

The interest of "functional metamodeling" is that it leads to compact and formal representation of modeling elements. In particular, functions are used to establish (meta)models or transformations equivalence. Equivalence means that two elements correspond to a same value/function, i.e. information is not lost. The second interest of the concept is that it is not only theoretical. In particular, the examples presented lead to a tool realized using the functional programming language Haskell that transforms EMF models for DES to models usable by the model checker LTSA, then to an implementation into Haskell.

REFERENCES


\(^5\) https://gslt.inrets.fr/projet/tucs/