GraphXica: a Language for Graphical Animation of Models

T. Prosvirnova, M. Batteux, A. Maarouf & A. Rauzy
LIX
Ecole Polytechnique, Palaiseau, France

ABSTRACT: The objective of this article is to present GraphXica – a Domain Specific Language (DSL) for graphical animation of models. GraphXica enables to describe graphical representations of models and their animations. Given a graphical representation and animation description of the model, different kinds of Graphical User Interfaces (GUIs) can be generated to simulate it, for example a web based interface, a Java interface, etc.
This work is a part of AltaRica 3.0 project, which aims to propose a set of authoring, simulation and assessment tools to perform Model-Based Safety Analyses. The new version of AltaRica modeling language is in the heart of the project. It is a textual language but graphical representations can be easily associated to textual models. GraphXica can be used to define graphical representations and animations of AltaRica 3.0 models. Then a GUI can be generated to animate these models. Coupled with a stepwise simulator, it enables to perform virtual experiments on systems, via models.
GraphXica is a generic DSL and can be used to describe graphical representations and animations of any kind of models and data.

1 INTRODUCTION

The Model-Based approach for safety and reliability analysis is gradually wining the trust of engineers but is still an active domain of research. Safety engineers master "traditional" risk modeling formalisms, such as "Failure Mode, Effects and Criticality Analysis" (FMECA), Fault Trees (FT), Event Trees (ET), Markov Processes. Efficient algorithms and tools are available. However, despite of their qualities, these formalisms share a major drawback: models are far from the specifications of the systems under study. As a consequence, models are hard to design and to maintain throughout the life cycle of systems. A small change in the specifications may require revisiting completely safety models, which is both resource consuming and error prone.

The high-level modeling language AltaRica Data-Flow (Rauzy 2002, Boiteau et al. 2006) has been created to tackle this problem. AltaRica Data-Flow models are made of hierarchies of reusable components. Graphical representations are associated with components, making models visually very close to Process and Instrumentation Diagrams. It is in the core of several Safety Analysis workshops and several successful industrial experiments have been held using AltaRica Data-Flow (Bernard et al. 2007, Bieber et al. 2008).

However, more than ten years of experience showed that both the language and the assessment tools can be improved. AltaRica 3.0 is an entirely new version of the language. Its underlying mathematical model – Guarded Transition Systems (Rauzy 2008, Prosvirnova and Rauzy 2012) – makes it possible to design acausal components and to handle looped systems. The development of a complete set of freeware authoring, simulation and assessment tools is planed, so to make them available to a wide audience.

The success AltaRica, partially, comes from the fact that graphical representations can be easily associated to textual models. Thus, models can be graphically animated. The incident or accident scenarios can be visualized and discussed. In a word, virtual experiments on systems can be performed using these models.

As a part of AltaRica 3.0 project our team works on the graphical representation and simulation of AltaRica 3.0 models. The goal of this communication is to present a Model-Based approach for 2D-3D event driven simulation. This approach consists in the definition of a Domain Specific Language (DSL) for graphical animation of models. This language enables to describe graphical 2D-3D representations of models and their animations. Then, it can be used to generate a graphical user interface (GUI) to animate models. The major advantage of this approach is that given

a graphical representation and an animation description of the model, different kinds of GUIs can be generated to simulate it, for example, a web interface or a java interface. The generated GUI can be coupled with a stepwise simulator to perform graphical simulations of models.

In fact, the main goal of such a DSL is to visualize the (dynamical) behavior of physical models. This visualization is particularly helpful during the design phases of complex systems. Until now, no common language dedicated to the graphical animation of models (independent of the application domain) has been designed. Our objective is to propose a DSL for graphical animation of models.

The remainder of this article is organized as follows. Section 2 gives an overview of the AltaRica 3.0 project. Section 3 presents the motivations of this work. Section 4 introduces GraphXica – a language for graphical animation of models. Section 5 summarizes the related works. Finally Section 6 concludes the article and outlines directions for future works.

2 ALTARICA 3.0 PROJECT

The objective of the AltaRica 3.0 project is to propose a set of authoring, simulation and assessment tools to perform Model-Based Safety Analyses. The overview of the project is presented Figure 1.

The new version of AltaRica modeling language, AltaRica 3.0, is in the heart of this project. It supports modeling of looped systems and bidirectional flows. This new version significantly increases the expressive power of the previous one without decreasing the efficiency of the assessment algorithms. AltaRica 3.0 models are compiled into a low level formalism: Guarded Transition Systems (Rauzy 2008, Prosvirnova and Rauzy 2012). Guarded Transition Systems is a states/transitions formalism that generalizes classical safety formalisms, such as Reliability Block Diagrams, Petri Nets and Markov chains. It is a pivot formalism for Safety Analyses: other safety models, not only AltaRica 3.0, can be compiled into Guarded Transition Systems to take benefits from the assessment tools. The assessment tools for Guarded Transition Systems already include prototypes of a compiler to Fault Trees, a compiler to Markov chains, a stochastic and a stepwise simulators. Prototypes of a model-checker and a reliability allocation module are planed to be developed. Distributed under a free license, the assessment tools enable users to perform virtual experiments on systems, via models, to compute different kinds of reliability indicators and, also, to perform cross check calculations.

3 MOTIVATIONS

As a part of AltaRica 3.0 Project our team works on the graphical simulation of AltaRica models. Graphical simulation of models has its own interest. First of all, it enables to better understand the system behavior. Second, virtual experiments can be performed on systems, via models, for example, it is possible to play calculated failure scenarii. Finally, it helps to debug and to validate the model.

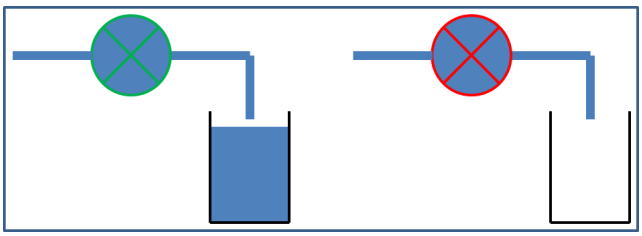

Figure 2: Graphical representation and animation of a water supply system

Consider a simple water supply system, composed of a pump and a tank. A pump can be in two states: WORKING or FAILED. If the pump is in state WORKING, then the tank is full, otherwise, it is empty. This system is represented in AltaRica as follows:

```
domain PumpState {WORKING, FAILED}
class Pump
  PumpState state (init = WORKING);
  Real input (reset = 0.0), output (reset = 0.0);
  event failure (delay = exponential(0.0005));
  event repair (delay = exponential(0.02));
  transition
    failure: state==WORKING -> state := FAILED;
    repair: state==FAILED -> state := WORKING;
  assertion
    output := if (state==WORKING) then input
              else 0.0;
end
class Tank
  Real input (reset = 0.0);
end
block WaterSupplySystem
  Pump pump;
  Tank tank;
  Real input (reset = 1.0);
  observer Boolean tank.isEmpty =
                          (tank.input == 0.0);
  assertion
    pump.input := input;
    tank.input := pump.output;
end
```

Our goal is to perform graphical simulation of this model. For that we need to define a graphical representation of the model and its animation, i.e. how the representation changes according to the values of system variables (see for example Figure 2). Basically, we would like to define the following animations:

1. If the pump is failed (*pump.state == FAILED*), then change the outline color to red.
2. If the tank is empty (*tank.input == 0.0*), then hide the blue rectangle.

To be able to perform graphical animations of models we will

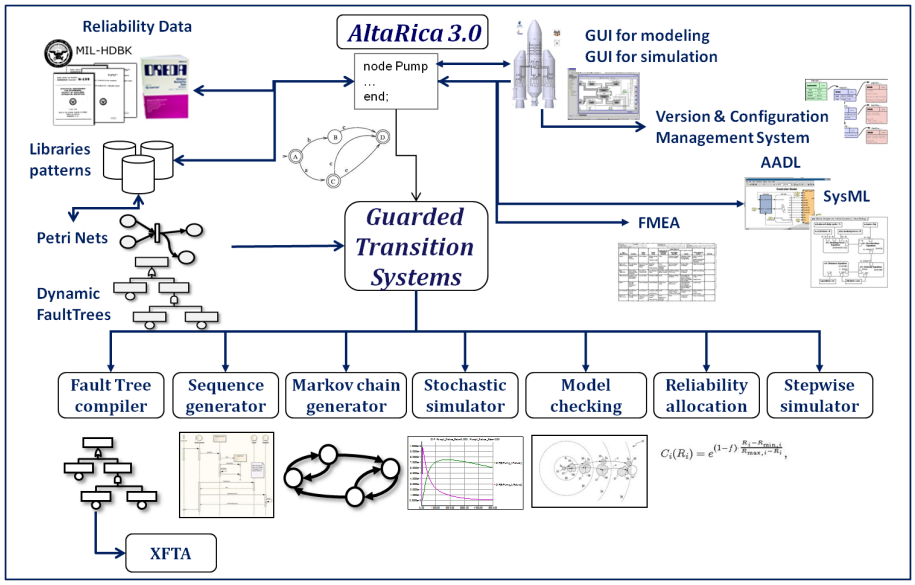

Figure 1: Overview of the AltaRica 3.0 project

- First, define a Domain Specific Language (DSL) to describe graphical representations and animations of models.
- Then, use this DSL to generate Graphical User Interfaces (GUIs) for event-driven simulation of models.
- Finally, couple the generated GUI with a stepwise simulator of AltaRica.

3.1 *DSL for graphical animation of models*

DSL for graphical animation should provide specific primitives to represent graphical objects (i.e. figures) and to describe their animations.

Graphical primitives Graphical objects give the static representation of the model (i.e. its 2D or 3D representation). The language should include at least the following graphical objects:

- Basic geometric figures: Rectangle, Ellipsoid, Line, etc.
- Links: Line, Point (to represent connections between graphical objects).
- Text (to display textual annotations of models).
- Bitmap (to use predefined pictures).

All graphical primitives should have some shared attributes, such as size, color, hidden, opacity, etc.

Composition It should be possible to design libraries of reusable graphical components (i.e. figures and their animations) and to assemble them in order to create graphical representations and animations of systems. In the example given Figure 2, one should be able to create graphical representations of a pump and a tank, their animations and to use them to create the graphical representation of the system, composed of the pump and the tank connected together.

Animation The ability to describe the animations of the model, i.e. how changes the graphical representation of the model in time, is the most important part of the language. Graphical animations of models should be done in two ways:

- Depending on external variables.
- According to user actions.

In the first case, the animation may depend on some external variables. When these variables change their values, the graphical representation is updated according to the defined animation rules. The values of these variables may be obtained from a model simulator, such as, for example, the stepwise simulator of AltaRica 3.0, from a file or a database, etc. The second way of animation is done through a Graphical User Interface (GUI) that gives the user the ability to interact with the model.

Animations are related to previously defined graphical primitives (figures). We should consider at least the following animations for them:

- Move.
- Hide or Show.
- Modify attribute values (e.g. color, size, etc.).

3.2 *Graphical user interfaces for simulation*

The generated GUI for simulation can be implemented in different programming languages, for example:

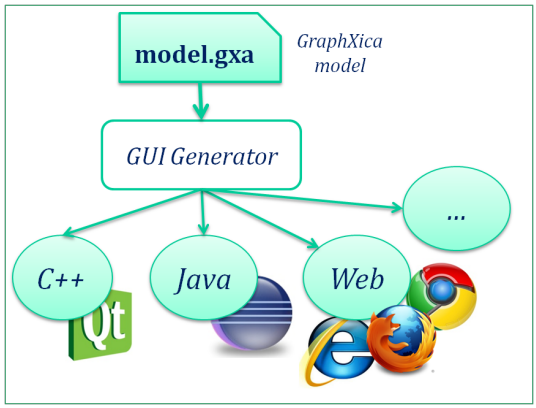

Figure 3: DSL for graphical animation of models

- All programming languages with Graphical libraries: C++ with Qt, C with SDL, Java with its standard graphics library, etc.
- Html and CSS to use with a web browser.
- Visual basic to use with the presentation program PowerPoint.

The GUI generator takes a graphical representation and animation of an AltaRica 3.0 model, generates a GUI that displays graphical animations of the model, as illustrated Figure 3.

3.3 *Graphical animation of AltaRica 3.0 models*

The stepwise simulator of AltaRica can be coupled with a graphical simulator in order to perform graphical simulation of models (e.g. see Perrot et al. (2010)). The link between the stepwise simulator and simulation GUI is done by a communication protocol.

Given an AltaRica model, it will be possible to define its graphical representation and animation using the DSL. Then a simulation GUI will be generated; it will be linked with a stepwise simulator by a communication protocol in order to perform graphical simulation of AltaRica models. The procedure is illustrated Figure 4.

4 GRAPHXICA

In this section we present GraphXica – a Domain Specific Language (DSL) for graphical animation of models. As shown figure 4, we use GraphXica with the stepwise simulator of the AltaRica 3.0. The generated simulation GUI, so called GraphXica Displayer, is implemented in Java. It takes a GraphXica model, corresponding to the graphical representation of the AltaRica 3.0 model. According to fired transitions, the stepwise simulator emits a vector of variables values. The GraphXica Displayer receives this vector and performs animations of figures. These animations are defined by the animation rules and depend on the values of variables received from the stepwise simulator.

GraphXica is, like AltaRica 3.0, a prototype oriented modeling language, see e.g. Noble et al. (1999) for a discussion on objects versus prototypes. Prototype orientation makes it possible to separate the knowledge into two distinct spaces: the stabilized knowledge, incorporated into libraries of on-the-shelf modeling components; the sandbox in which the system under study is modeled. In the sandbox, many components are unique and some others are instances of reusable components. With prototype-orientation, models can be reused in two ways: at component level by instantiating on-the-shelf components; at system level by cloning and modifying a model designed for a previous project. Classes represent the stabilized knowledge: they can be instantiated and extended like in object-oriented languages. Blocks model unique components, that cannot be instantiated. The system is always represented by a block.

A GraphXica model is made up of declarations. It is possible to declare global variables, variable domains (i.e. enumerated types) and components (i.e. classes or blocks) to describe figures and their animation rules. In the following, we present the grammar in extended BNF.

Model ::= (*Declaration*)* ;
Declaration ::=
  DomainDeclaration
| *ExternVariableDeclaration*
| *VariableDeclaration*
| *ClassDeclaration*
| *BlockDeclaration*
;

4.1 *Domains*

Domains are named sets of symbolic constants. They are defined in the following way:

DomainDeclaration ::=
  'domain' *Identifier* '{' *Identifier* (',' *Identifier*)* '}' ;

An identifier is a letter followed (not necessary) by a sequence of letters or numbers. The special character '_' is also included to define an identifier. The rule *Identifier* is the following:

Identifier ::=
  (*Letter* | '_') (*Letter* | *Number* | '_')* ;
Letter ::= 'a' | ... | 'z' | 'A' | ... | 'Z' ;
Number ::= '0' |... | '9' ;

Declared domains can be used as a type for variables anywhere in the model. In the example given Figure 2, domains expressing the state of the pump and the tank are declared as follows:

```
domain PumpState {WORKING, FAILED}
domain TankLevel {FULL, EMPTY}
```

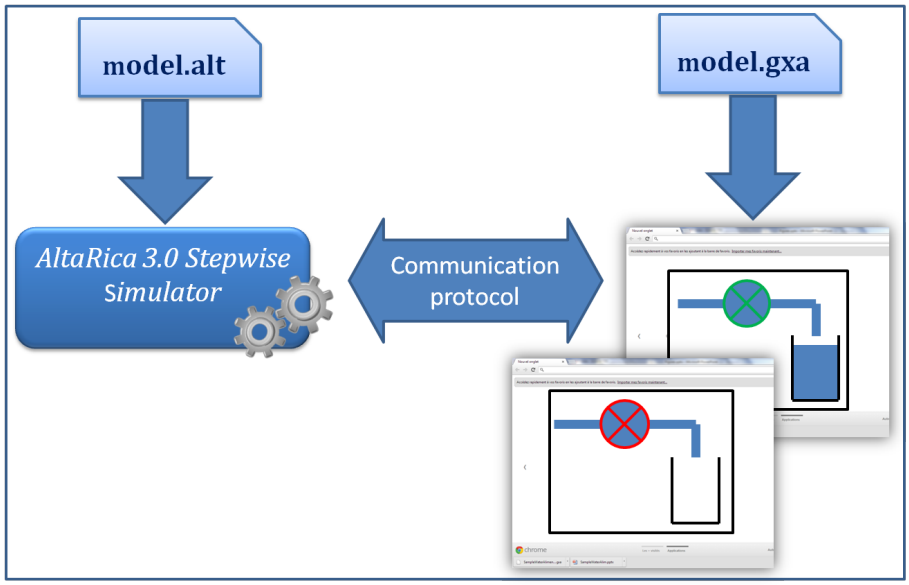

Figure 4: Graphical simulation of models

4.2 *Global variables*

Variables are declared in the following way:

ExternVariableDeclaration ::=
  'extern' *VariableDeclaration* ;
VariableDeclaration ::=
  Type Variable (',' *Variable*)* ';' ;
Type ::=
    NumericalType'
  | *FigureType*
  | *Identifier*
  ;
NumericalType ::=
    'Boolean'
  | 'Integer'
  | 'Real'
  ;
Variable ::=
    Identifier ['(' *Attributes* ')'] ;
Attributes ::=
    Attribute (',' *Attribute*)* ;
Attribute ::=
    Identifier '=' *Expression* ;

The rule *Expression* defines formulas built over variables and parameters using common arithmetic, comparison and logical operators. We will not detail it here. Global variables can only have numerical or user defined type. Thus, the rule *FigureType* will be defined later.

Global variables have the same sense like in the programming languages C, C++ or Java. A global variable can be used anywhere in the model. The keyword `extern` can be added to the declaration of a variable. It expresses the fact that the variable will be linked to a variable coming from the vector of values (from an assessment tool linked to the GraphXica Displayer). When a variable is declared, a set of attributes (e.g. its initial value) can also be declared. In the example given Figure 2, we will declare two external variables `ext_pumpState` and `ext_tankInput`, corresponding to variables from AltaRica 3.0 model:

```
extern PumpState ext_pumpState (init = WORKING);
extern Boolean ext_tankInput (reset = false);
```

4.3 *Classes and Blocks*

Classes are used to create libraries of reusable graphical representations and animations. Blocks are used to design graphical representations of systems using libraries of reusable components. They represent clearly the separation between stabilized knowledge, incorporated into libraries of on-the-shelf modeling components, and the sandbox in which the system under study is modeled. Classes are declared as follows:

ClassDeclaration ::=
    'class' *Identifier*
      (*ComponentDeclaration*)*
      [*Animations*]
    'end' ;

Blocks are declared as follows:

BlockDeclaration ::=
    'block' *Identifier*
      (*ComponentDeclaration*)*
      [*Animations*]
    'end' ;

Components are declared as follows:

ComponentDeclaration ::=
    VariableDeclaration
  | *ParameterDeclaration*
  ;

Classes may embed instances of other classes so to get hierarchical representations of systems under study. Blocks may be composed of other blocks and instantiated classes. Blocks and classes may contain variables, parameters, figures and animation rules. Here, types of variables can be figures type (e.g.: rectangle, oval, line, etc.) or user declared classes.

The following classes describe the graphical representations of the pump and the tank from the system given Figure 2.

```
class PumpRep
  parameter PumpState state  = WORKING;
  Oval cir (x = 0, y = 0, width = 5, height = 5,
       color = blue, lineColor = black,
       thickness = 2);
  Line delta1 (x = 1.5, y = 8.5, width = 7,
       height = -7, color = black, thickness = 2);
  Line delta2 (x = 1.5, y = 1.5, width = 7,
       height = 7, color = black, thickness = 2);
  animation
    state == FAILED ->
       {
         delta1.color := red;
         delta2.color := red;
         cir.lineColor := red;
       }
    state == WORKING ->
       {
         delta1.color := green;
         delta2.color := green;
         cir.lineColor := green;
       }
end
```

```
class TankRep
  parameter TankLevel level = FULL;
  Line horLeft (x = 0 ,y = 0, width = 14,
       height = 0, color = black,
       thickness = 2);
  Line horRight (x = 10 ,y = 0, width = 14,
       height = 0, color = black,
       thickness = 2);
  Line base (x = 0 ,y = 14, width = 0,
       height = 10, color = black,
       thickness = 2 );
  Rectangle rectFull (x = 0, y = 2,
       width = 10, height = 12,
       color = blue, thickness = 0,
       visible = true);
  animation
    level == EMPTY -> rectFull.visible := false;
    level == FULL -> rectFull.visible := true;
end
```

Graphical primitives GraphXica contains specific primitives to represent figures and to define their animations. Figures are declared as variables: they have a type, a name and a list of attributes. Different types of figures are included, such as, for example, `Line`, `Rectangle`, `Oval`, `Bitmap`, `Text`, etc. Each type of figure has its own list of attributes. However, there are some common attributes, such as the color of the figure, its position (with two dimensions), its size (with two dimensions), its visibility and its opacity. The idea is to consider a figure inscribed inside an "imaginary" rectangle (it is not a figure of the language) and positions are according to the left-top corner.

In the example given above a pump is represented by a circle (an oval) *cir* and two lines *delta1* and *delta2* and a tank is described by three lines *horLeft, horRight, base* and a rectangle *rectFull*. A line is defined by the coordinates of its source and its target, by its color and its thickness. A rectangle is defined by the coordinates of its left-top corner, its size, its color and the thickness of its border. Note, that users don't have to fill in all the attributes, GraphXica Displayer gives a default value for all unspecified attributes.

Parameters Parameters are introduced by a keyword **parameter**, followed by the type, the name and the value (i.e. an expression depending on variables and other parameters). They are declared in the following way:

ParameterDeclaration ::=
'parameter' *NumericalType Identifier* '=' *Expression* ';' ;

In the example given earlier we declare two parameters: *level* in the class *TankRep* and *state* in the class *PumpRep*. The values of this parameters can be changed when the classes are instantiated. Parameters can be used in different manners: they can be linked to external variables or they can define constant values, such as circle radius, rectangle height, etc.

Variables Local variables within a class or a block can be declared in the same way as global variables.

Animations Animations are defined by a set of rules. Each rule is represented by a condition followed by a set of instructions. Animations are declared in the following way:

Animations ::=
'animation' (*Animation*)* ;
Animation ::=
Condition '→' *Instruction* ;
Condition ::=
LogicalExpression ;
Instruction ::=
Assignment
| *Block*
;
Assignment ::= *Identifier* ':=' *Expression* ';' ;
Block ::= '{' *Instruction*+ '}' ;

Conditions are Boolean expressions, built over variables and parameters of the model (e.g. `state == FAILED`) or user actions (e.g. click on a figure). Boolean expressions are evaluated according to the vectors of values, received from a linked assessment

tool. Instructions enable to modify the values of attributes of figures (e.g. change the color or visibility of a figure, move or enlarge a figure, etc.). Of course, the modifications of attribute values are defined according to the considered figures. For example, it is possible to change the color of a **Rectangle**, but it is not possible to modify the color of a **Bitmap**.

In the example given above, the class *Pump* defines the following animation rules: if the value of the parameter *state* is WORKING, then the color of lines is green, otherwise it is red. In the class *Tank* if the value of the parameter *level* is EMPTY then the rectangle *rectFull* is hidden.

Composition To create a GraphXica model of the water supply system, given Figure 2, we use the previously defined classes *PumpRep* and *TankRep*. The block *WaterSupplySystemRep* is composed of an instance of class *PumpRep* and an instance of class *TankRep*. The parameter *state* of the *pumpRep* is set to the external variable `ext_pumpState` (this variable comes from AltaRica model of the system). The parameter *level* of the *tankRep* is calculated according to the external variable `ext_tankInput` coming from the AltaRica model.

```
block WaterSupplySystemRep
  TankRep tankRep (posX = 1, posY = 8,
      level = if ext_tankInput then FULL
                  else EMPTY);
  PumpRep pumpRep (x = 35, y = 2,
       state = ext_pumpState);
  Line hLineLeft (x = 0, y = 5, width = 10,
       height = 0, color = blue, thickness = 6 );
  Line vLineLeft (x = 30, y = 5, width = 0,
       height = 7, color = blue, thickness = 6 );
  Line hLineRight (x = 20, y = 5, width = 10,
       height = 0, color = blue, thickness = 6 );
end
```

The animation rules of the block are executed in the following way: first, animation rules of all instantiated classes and nested blocks are executed in the same order as they are declared in the block, then animation rules of the main block are performed.

5 RELATED WORKS

In this section we introduce some existing languages for graphical representation and animation of models.

Modelica (Fritzson 2004) is an object-oriented language based on equations for modeling continuous or discrete event behavior of physical systems. The grammar of Modelica, amongst others, defines annotations used to store additional information about the model, such as its graphical representation. They are used to graphically represent a model and its components by means of graphical objects (rectangles, circles, etc.), component icons and connection lines. Although Modelica's annotations define all the necessary properties and primitives to represent a model in a graphical way. These representations are purely static. There is no animation of models and also no interaction between the user and a model.

Some existing languages are specifically dedicated to animation of algorithms: JAWAA (Rodger 2002), XAAL (Karavirta 2005), AnimalScript (Rößling and Freisleben 2001), JSamba (Stasko 1998), JHAVÉ (Naps et al. 2000). They are scripting languages for creating animation of algorithms. They contain primitives to represent graphical objects such as circles, rectangles, lines, etc.; and to animate them. The main principle is to write, or automatically generate, a script from an algorithm. This script corresponds to a translation from the algorithm to its graphical representation and animation. The script is then used by a "displayer", e.g. a web browser for JAWAA. Nevertheless, no communication is possible between the "displayer" and the user or another tool. For example, if one want to change the value of a variable, he has to rewrite the script.

SVG, for Scalable Vector Graphics (Eisenberg 2002), is a language for describing two-dimensional graphics in XML. SVG allows to create graphical forms (e.g.: circles, polygon, etc.), images and texts. SVG representations (i.e. drawing) are interactive. They can react to user's actions such as pressed button by the mouse.

The study of existing languages containing a part dedicated to graphical animation and the limited domain of use of these languages give us the reason to create a new language of graphical animation of models (GraphXica).

6 CONCLUSION AND PERSPECTIVES

In this article, we introduced GraphXica – a high-level modeling language for graphical animation of models. GraphXica enables to describe graphical representations of models and their animations. Animations are defined according to values of external variables and user actions. GraphXica models can be used to generate different types of Graphical User Interfaces (e.g. a Java interface, a web based interface, etc.). The generated GUI, so called GraphXica displayer, can be coupled with simulators in order to perform graphical simulations of models. GraphXica is a prototyped based modeling language. Thus, it is possible to create libraries of reusable graphical components and to use them to design graphical representations and animations of complex systems.

This work is done as a part of AltaRica 3.0 Project which aims to propose a complete set of authoring, simulation and assessment tools to perform Model-Based Safety Analyses. The idea is to couple GraphXica displayer with AltaRica 3.0 stepwise simulator in order to perform graphical simulation of models.

GraphXica displayers, implemented in Java and in C++ Qt, are currently under development. We also

plan to implement a web based GraphXica displayer using HTML 5 and CSS. Forthcoming papers will completely present the grammar of GraphXica and its semantic. Furthermore, our future works will focus on the definition and on the implementation of the communication protocol between GraphXica displayers and stepwise simulator. The redaction of pedagogical materials for GraphXica, including a primer, a best practices guide and a book of exercises is also an important part of the project. These materials will help interested persons to quickly understand and learn how to use GraphXica. Another interesting tools will be a generator of GraphXica models from AltaRica 3.0 models and an authoring tool for GraphXica.

REFERENCES

Bernard, R., J.-J. Aubert, P. Bieber, C. Merlini, & S. Metge (2007). Experiments in model-based safety analysis: flight controls. In *Proceedings of IFAC workshop on Dependable Control of Discrete Systems, Cachan*.

Bieber, P., J.-P. Blanquart, G. Durrieu, D. Lesens, J. Lucotte, F. Tardy, M. Turin, C. Seguin, & E. Conquet (2008, January). Integration of formal fault analysis in assert: Case studies and lessons learnt. In *Proceedings of 4th European Congress Embedded Real Time Software, ERTS 2008*, Toulouse (France).

Boiteau, M., Y. Dutuit, A. Rauzy, & J.-P. Signoret (2006). The altarica data-flow language in use: Assessment of production availability of a multistates system. *Reliability Engineering and System Safety 91*, 747–755.

Eisenberg, J. (2002, February). *SVG Essentials*. O'Reilly Media.

Fritzson, P. (2004). *Principles of Object-Oriented Modeling and Simulation with Modelica 2.1*. John Wiley & Sons Inc.

Karavirta, V. (2005). Xaal - extensible algorithm animation language. Master's thesis, Helsinki University of Technology.

Naps, T., J. Eagan, & L. Norton (2000). JHAVÉ: An environment to actively engage students in web-based algorithm visualizations. *31st ACM SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2000), Austin, Texas*, 109–113.

Noble, J., A. Taivalsaari, & I. Moore (1999). *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag.

Perrot, B., T. Prosvirnova, A. Rauzy, J.-P. S. d'Izarn, & R. Schoening (2010, October). Expériences de couplages de modèles AltaRica avec des interfaces métiers. In E. Fadier (Ed.), *Actes du congrès LambdaMu'17 (actes électroniques)*. IMdR.

Prosvirnova, T. & A. Rauzy (2012, Octobre). Guarded transition systems: Pivot modelling formalism for safety analysis. In J. Barbet (Ed.), *Actes du Congrès LambdaMu 18*.

Rauzy, A. (2002). Modes automata and their compilation into fault trees. *Reliability Engineering and System Safety 78*, 1–12.

Rauzy, A. (2008). Guarded transition systems: a new states/events formalism for reliability studies. *Journal of Risk and Reliability 222*(4), 495–505.

Rodger, S. (2002, June). Using hands-on visualizations to teach computer science from beginning curses to advanced courses. In *Proceeding of the Second Program Visualization Workshop*.

Rößling, G. & B. Freisleben (2001). Animalscript: An extensible scripting language for algorithm animation.

Stasko, J. (1998). Smooth continuous animation for portraying algorithms and processes. In *Software Visualization*, pp. 103–118. MIT Press.