ViBBA: A Toolbox for Automatic Model Driven Animation

Angelo Gargantini [*]
University of Bergamo - Italy

Elvinia Riccobene [†]
University of Milan - Italy

Abstract

In this paper, we present ViBBA (Visual Bean Based Animator), a toolbox supporting the automatic model driven animation [7] consisting in automatically deriving scenarios exposing critical system behaviors from requirements specifications, and animating those scenarios through a graphical interface. ViBBA allows visual construction of graphical animators and scenarios animation. It is integrated with the ATGT [5], that automatically generates scenarios (in XML format) from Abstract State Machines (ASMs) by exploiting counter example generation capability of the model checker Spin. The operation of ViBBA is shown by means of an example of formal specification where scenarios are automatically generated from the model and then animated.

1 Introduction

Validation of formal specifications is a typical human intensive activity during system and software development. It must be performed at the very early stages of the development process in order to check whether the system, as specified, meets the customer needs, to better understand models and requirements, to gain confidence that specifications capture informal requirements, and to detect faults as early as possible with limited effort. Techniques for validation include scenarios generation, development of prototypes, simulation, and also testing. Validation should precede the application of more expensive and accurate methods, like formal requirements analysis and formal verification of properties, that should be applied only when designers have enough confidence that specifications are correct. To be effective, the validation activity should be supported by tools easy to use and requiring minimum user effort, in particular, visual representations can greatly benefit the task of this activity.
In [7], the *automatic model driven animation* is presented as a novel approach to validate requirements specifications. This approach is based on *graphical animation* (hereinafter briefly called *animation*) [10, 13, 15] which basically consists of simulation, providing the

[*]Dipartimento di Matematica e Informatica -email: gargantini@dmi.unict.it
[†]Dipartimento di Tecnologie dell'Informazione -email: riccobene@dti.unimi.it

user with a graphical interface suitable to show the state of the system by means of icons, buttons, panels, slides, and so on. An animator might be a prototype ad hoc developed or a simulator endowed with a complex graphical interface.
Animation offers several advantages both for designers and for customers. Mainly, through animation designers better understand the requirements specification and can find failures and faults. For customers, looking at the real behavior shown by the animator is better than reading mathematical or logical formulas modeling the system behavior (normally customers do not like mathematical formalisms). Customers can ignore in which notation the specification has been written and they do not need to learn new (formal) languages. Moreover animation does not require skills, ingenuity and expertise as those required by *heavy* formal methodologies like theorem proving. Therefore, animation is easier to learn, easier to use, and easier to understand by a broader group of people than formal techniques. However, animation cannot prove that a specification is correct, but it can only uncover faults [16]. For this reason, it is very similar to testing.
The main problems of animation are the *selection of system behaviors* to animate, and the actual *graphical animation* of the selected scenarios. Therefore, a toolbox is required which should be endowed with a *system able to build scenarios*, possibly in an automatic way, from the specification with respect to a given critical aspect to analyze, and with a *graphical animator panel* that visually represents the real system under animation. This verisimilar environment is of great help to customers: they do not have to learn a new environment (like a new tool or IDE or interface), they do not watch variable values only by means of digits or strings (like in debuggers), they do not have to query system state by typing commands, and they do not need to guess the meaning of the values in terms of system behaviors.
In this paper, we present ViBBA (Visual Bean Based Animator), a toolbox for visually building graphical animator and animate scenarios. ViBBA is integrated with the ATGT (Automatic Tests Generator Tool) [5], that automatically generates scenarios (in XML format) from formal specifications written using the ASM notation [9] by exploiting counter example generation capability of the model checker Spin [12]. Section 2 introduces automatic model driven animation and explains how scenarios are automatically generated and represented in XML files. ViBBA toolbox is described in Section 3 where we explain how graphical animator panels are built and how scenarios are animated in ViBBA. The operation of ViBBA is shown by means of the example of the formal specification of a Safety Injection System (SIS) [4]. Related work is discussed in Section 4, and future work and conclusions are given in Section 5. Appendix reports a short description of the case study and its ASM formal specification given in terms of a signature and a set of transition rules modeling how variables are updated according to the system requirements.

2 Automatic Model Driven Animation

We distinguish three main ways for selecting scenarios to animate:

- *user driven*: users (customers or designers) “play” with the animator and check whether the specification meets customers needs or not. Users select inputs, regardless of the specification, by means of buttons, slides, and graphical elements; outputs are computed according to the specification (that acts as oracle) and shown by means of other graphical elements. Except for the graphical interface, this approach is very similar to the classical simulation.

- *random*: inputs are randomly generated taking into account only their specified constraints. Outputs are computed according to the specification and shown through the graphical animator. In this case, the user observes the system behavior and judges its correctness. This approach is proposed in [17].

- *model driven*: inputs are selected starting from requirements specifications in a systematic way, either manually or automatically. The former is similar to case 1, but the specification is used as guideline. The latter is the new approach proposed in [7].

In any of these three animation approaches, the judgment of the specification correctness is left to the human; however, the effort required substantially differs, and model driven animation is more efficient than user driven or random animation for several reasons. By animation, designers gain confidence that the specification is correct only if the model has been extensively simulated and enough scenarios have been checked. Therefore, performing a *good* selection of critical scenarios, that can uncover specification faults, is crucial. Since random animation produces a huge amount of scenarios, the careful review of all the behaviors is time consuming. Furthermore, only few generated scenarios are able to expose critical faults.
In user driven, as well as in manual model driven animation, designers have the responsibility to cover all the critical behaviors. Since selection is carried out by hand, they risk to leave out some particular cases and choose only a small subset of all the specified behaviors. The manual selection of scenarios is, therefore, expensive and error-prone, especially in user driven animation, where specifications are not used as guidelines.
Automatic model driven animation [7] has the advantage to automatically derive scenarios from specification, and to assure the animation of all the critical behaviors according to the requirements. It does not require great user skill and ingenuity and is an effective approach for model validation. Figure 1 shows the process of generation and animation of critical scenarios, which is explained in the following.

Animation Goals They are predicates over the system state or over the system transitions representing critical behaviors to animate. They are systematically derived from a formal specification in an automatic way, selecting the most critical events and behavior by analyzing the structure of the specification as explained in [7]. From ASM specifications, they can derived from rule guards. For example, for rule R5 of the SIS specification the animation goal is `Reset = on and (Pressure = TooLow or Pressure = Normal).`

Generation of Animation Sequences Scenarios achieving the animation goals are computed by exploiting the counter example generation of model checkers. We encode the formal specification in the language of a model checker. Then, for each animation goal a, we compute the animation sequence that covers a by trying to prove with the model checker

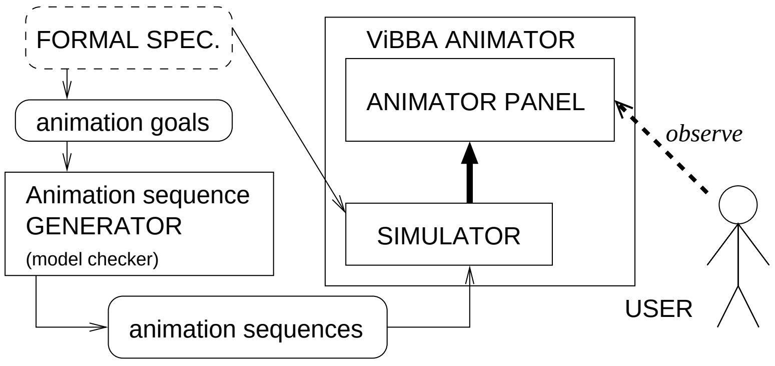


Figure 1: Automatic Model Driven Animation

the *trap property* $\neg a$. If the model checker finds a state where $\neg a$ is false, it stops and prints as counter example a state sequence leading to that state. This sequence is the animation sequence for a. Note that the generation of animation sequence is totally automatic and the model checker could fail to find the animation sequence for an animation goal. Indeed, finding a valid sequence to cover a predicate is undecidable.
We have developed a tool, called Automatic Tests Generator Tool (ATGT) [6, 8, 5], that automatically derives animation goals from ASM specifications [9], encodes ASM models in PROMELA, the language of the model checker Spin [12], and automatically generates test sequences which accomplish a desired coverage.

Animation Sequences In our framework, a scenario is a sequence of states and each state is a function mapping a set of variables into their values. The scenarios to animate, generated by means of the ATGT tool, are in XML format, whose Document Type Definition (DTD) is shown in Figure 2. The DTD defines the structure of scenarios. Figure 3 shows an example of XML scenario where at state 15 the animation goal derived from R5 becomes true.

```
<?xml version='1.0' encoding='UTF-8'?>
<!ELEMENT SCENARIO (STATE)*>
<!ELEMENT STATE (var,val)*>
<!ATTLIST STATE NUMBER CDATA #IMPLIED>
<!ELEMENT var (#PCDATA)>
<!ELEMENT val (#PCDATA)>
```

Figure 2: DTD definition of scenarios

According to the DTD, the XML file has the main element `SCENARIO` representing the scenario to animate, which contains 0 or more `STATE` elements. Each `STATE` element has

```
<?xml version="1.0"?>
<!DOCTYPE SCENARIO SYSTEM "http:// www.dmi.unict.it/garganti/vibba/scenarioDTD.dtd">
<SCENARIO>
 <STATE NUMBER="1">
 <var>Pressure</var><val>Normal</val>
...
 <STATE NUMBER="15">
  <var>Reset</var><val>off</val>
  <var>Block</var><val>off</val>
  <var>Overridden</var><val>true</val>
 </STATE>
 <STATE NUMBER="16">
  <var>Reset</var><val>on</val>
 </STATE>
 <STATE NUMBER="17">
  <var>Overridden</var><val>false</val>
 </STATE>
....
</SCENARIO>
```

Figure 3: Example of XML scenario

an attribute (denoted by `ATTLIST`) showing its number (`NUMBER`) and contains 0 or more pairs `(var,val)` that represent the update of the variable `var` to the value `val`. `Var` and `val` are strings, as denoted by `PCDATA`. In the example, the variables `Reset` and `Block` are off at state 15. At state 16, `Reset` becomes `on`, and `Overridden` becomes `false` in the next state.
Note that scenarios may be written by hand through a XML editor following the DTD for scenarios provided in the package or by another tool or program.

3 ViBBA Toolbox

ViBBA (Visual Bean Based Animator) is a tool for building and animating graphical panels (also called *animator panels*). It provides the user with a graphical panel and a toolbox for adding standard graphical elements to the graphical panel and to visually specify how they must change during the animation. To perform the animation of desired scenarios, the user has first to build the animator panel and then to animate the scenarios. ViBBA is based on the Java Bean technology developed by Sun Microsystems.

3.1 ViBBA Architecture

The tool architecture is depicted in the following Figure 4.We distinguish three main components of the animator: (*1*) the *ATGT (Automatic Tests Generator Tool)* which generates scenarios following user requests and exploiting the model checkers, and stores them in a

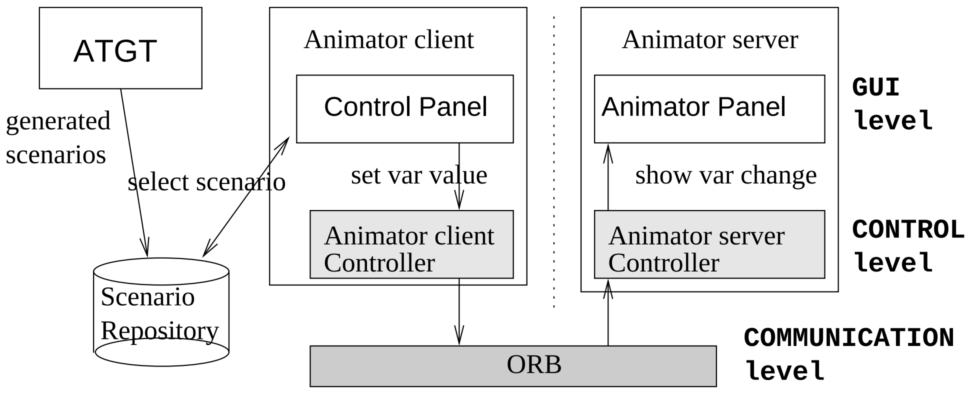


Figure 4: ViBBA Architecture

repository; (*2*) the *animator client* which retrieves generated scenarios and drives the animator server by providing the scenario to be animated (i.e. setting the value of variables); (3) the *animator server* which shows status and behavior of the system by means of the animator panel.
Animator client and server have a three layer architecture. At communication level, client and server communicate through a CORBA ORB. At control level, the animator client controller and the animator server controller manage their graphical interfaces, start the processes and connect and register themselves with the ORB. At GUI level, the animator client controls the animator through the control panel as explained in Section 3.3, while the GUI part of the animator server is the real graphical panel that shows the system state and behavior to the user.
Note that during animation we have three active components: a *Control Panel*, to control the animation; an *Animator Panel*, to show the state of the system by means of graphical elements; and an *ORB*: to connect the Control Panel with the Animator Panel.
The proposed architecture is highly modular and makes the animator easy to change adding new graphical elements and new action beans as explained in Section 3.4. The animator panel can be easily built as described in Section 3.2; this feature is relevant since each specification requires a new animator panel to be animated. We chose CORBA, a well established OMG standard, since it allows the animator to work in a distributed way and the animator server to be easly connected to other software. The animator service may run on a remote machine, for example a computer of the customer, while the animator client would be controlled by the designer.

3.2 Building Animators using ViBBA

An animator is compound of two kinds of objects: *graphical elements* and *animation action objects*. To build an animator panel, the user must add (by using drag and drop from a palette) to the ViBBA animator panel, graphical elements, which can be any Java Bean, but

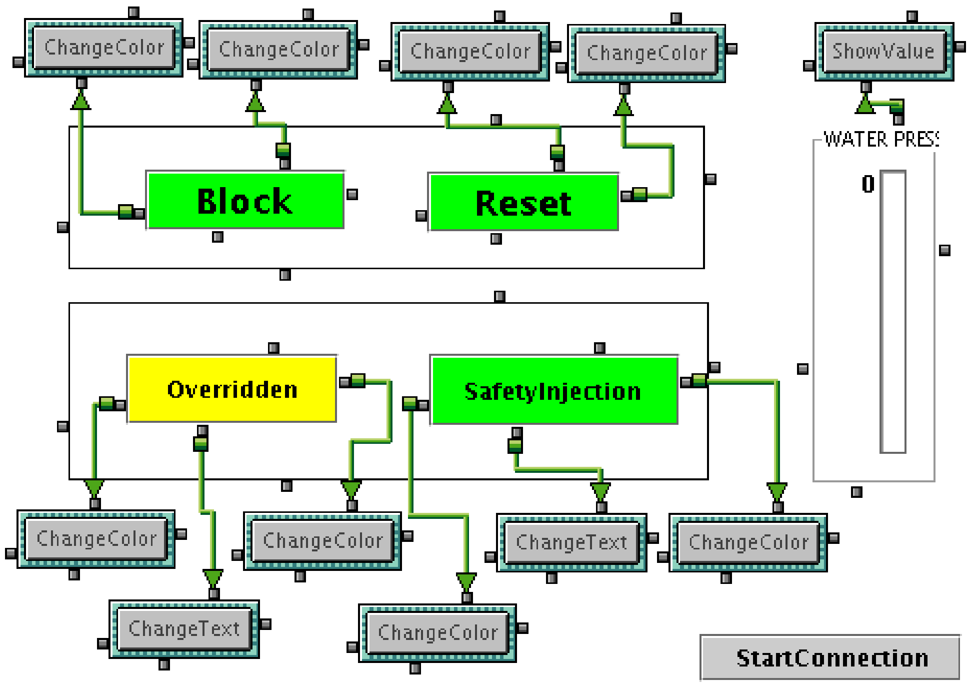
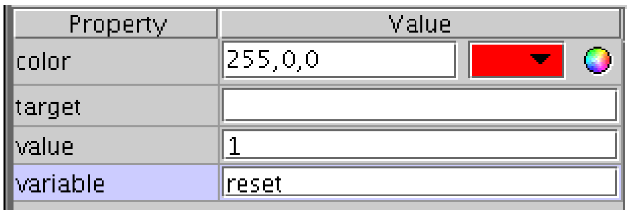


Figure 5: Building the Animator Panel for SIS

we normally use Swing components. *Animation action objects* (or *animation action beans)* specify the behavior of graphical elements. Depending on the desired behavior, the user must *a*) add (by drag and drop) to the animator panel some animation action objects, *b*) link (by connecting objects using the mouse) each animation action object to its *target* that is a graphical element, and *c*) set action properties in an appropriate way (depending on the type of animation action object). Note that a graphical element can be target of many animation action objects. ViBBA provides the following basic *Animation Action* beans:

Animation Action	Purpose	Method invoked on target
ChangeColor	change target color	setBackground
ChangeText	change target text	setText
ShowValue	show variable value	setText
UnChangeColor	set/unset target color	setBackground

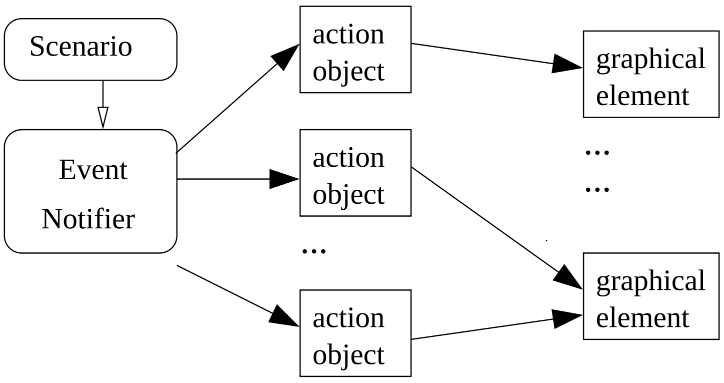
Figure 5 shows the animator panel for SIS. *Block*, *Reset*, *Overridden*, *SafetyInjection* are modeled by buttons as graphical elements, whereas *WaterPressure* is represented by a new graphical element (defined as explained in Section 3.4) that shows the value by means of a textual label and a graphical slide (the graphical element for *Pressure* is not shown



Property	Value
color	255,0,0 🟥
target	
value	1
variable	reset

Figure 6: Property of the Change Color bean

for lack of space). The figure also shows the action beans *ChangeColor*, *ChangeText*, and *ShowValue* and their connections to their targets, denoted by a green arrow.
Each animation action bean has a set of properties that the user can change using the property inspector. For example, Figure 6 shows the properties of a ChangeColor bean: *color* indicates the color that the bean will set to its target, *target* indicates its target (set by visually linking objects in the panel), *variable* and *value* indicate the variable name and value that will cause the animation action bean to act.
When an action bean is added to the panel, it is automatically registered to the *event notifier*, that is a ViBBA component in the animator panel. During animation, a variable value change is notified by the event notifier to the registered action objects; the action object changes the graphical aspect of its target, as specified by the user. The information flow is shown by the following figure.



3.3 Animation of Scenarios

After an animator for our application has been built, and a set of scenarios has been generated, the actual animation of scenarios can be performed.
To animate a scenario, first of all one must start the Object Request Broker Daemon

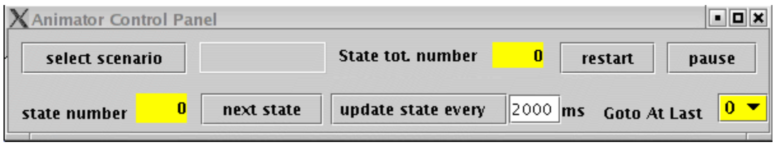


Figure 7: Control Panel

(ORBD)[1]. Then, start the server pressing the *StartConnection* button in the animator panel (see Figure 5). Finally the user can run the client, that opens the Control Panel shown in Figure 7. Through the Control Panel, the user selects the scenario to animate pressing the *select scenario* button. The first state of the selected scenario is loaded and the animation starts. The *State tot. number* label shows the total number of states composing the scenario, while *state number* text field shows step by step the number of the state under processing.

There are two different ways to animate a scenario: step by step (the user makes the animator progress by pressing the *next state* button) or automatically (setting the time interval between two subsequent states through the *millisecond* text field and pressing the *update state every* button). The *pause* button makes the animation stop; the *GoToAtLast* makes skip all the states till the selected one and the *restart* button makes the animation restart from the first state.
In Figure 8, we show the animation for SIS of three consecutive states (taken from the scenario of Figure 3): in the first state *WaterPressure* is *1200* (pressure is normal) and the system is not blocked and not reset, while in the second state *Reset* becomes *on* (and its button becomes red), and in the last state *Overridden* becomes *false* (and its button becomes blue).

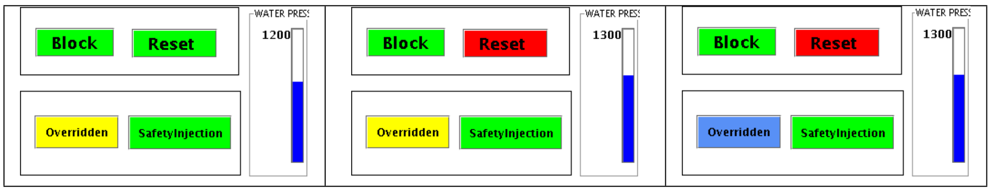


Figure 8: Animation of SIS panel: three consecutive states

3.4 How to Create New Animation Actions and New Graphical Elements

ViBBA can be extended by creating new animator actions and new graphical elements. A new animation action *NewAction* must be defined as Java class that extends the ViBBA Java

[1]We normally use Java JDK ORB, that starts by typing on the shell the command: *orbd -ORBInitialPort "nameServerPort"*, where *"nameServerPort"* specifies the port on which the bootstrap name server is running.

abstract class *AnimationAction*. The constructor of *NewAction* must take two parameters: the name of the method that is invoked by *NewAction* on the target (i.e. "setText") and the parameters that the target method requires (i.e. String.class). In this way, when the user links *NewAction* to a graphical element, ViBBA checks that the graphical element really has the method invoked by *NewAction*. Furthermore, *NewAction* must implement the method *notifyVarChange*(*String var*, *String value*) that is called by the event notifier when a variable changes its value. *NewAction* can define some properties accessed by set and get methods, that the user can change using the Java bean property inspector. *NewAction* must be compiled in a JAR file and loaded.
New graphical elements can be defined as normal Java Beans. For example for SIS, we define a new graphical element *WaterPressureLevel* that contains a label "WATER PRESSURE", a text field that shows the textual value of WaterPressure and a slide that indicates the value of WaterPressure with a blue level indicator. *WaterPressureLevel* class implements the method *setText*(*String WaterPressureValue*) that updates the text field and the blue slide. *WaterPressureLevel* can be set as target of a ShowValue animator action object, that invokes the setText method of *WaterPressureLevel.*

4 Related work

There exist several tools and methods for animating formal specifications. [1] uses the B-Toolkit for animation of B specifications. The B-Toolkit presents the user a symbolic representation of the system state and allows the invocation of specification operations. The interface is mainly text based, and the user can perform queries and run commands by typing suitable instructions in a text console. This approach is more similar to simulation, because it does not exploit any graphical element. The approach presented in [13] suffers the same limitation. [13] clearly discusses the benefits of animation in the contest of light weight approaches to formal methods, in particular Z. The user can perform a set of queries checking the initialization, verifying the preconditions of schema, and performing a simple reachability property. The model checker Spin [12] has its own simulator that provides the user with information about the system state and allows, besides verification of properties, interactive simulation, simulation driven by counter examples, and random simulation. Also Spin displays this information mainly in text format. [17] proposes a random animation for Lustre specifications. Random test inputs are generated taking into account only the constraints about the environment. Safety requirements are checked using the generated scenarios. An AsmGofer[18] simulator for UML state machines execution is presented in [3]. The user has to execute the state machine and to query function values by a textual shell.
The use of a graphical domain-specific simulator for SCR is presented in [11]. SCR simulator supports the construction of graphical front-ends, tailored to particular applications. [11] presents a front-end for a real aircraft attack specification. A pilot, instead of entering values for monitored variables and seeing the values of the controlled ones, interacts with the simulator and the results are presented in the graphically simulated cockpit.
A graphical simulator developed for the ASM specification of a light control system is

presented in [2]. It is based on AsmGofer and uses TCL/TK for the animator panel.
A complex and complete graphical animator is presented in [15, 14]. The authors develop an animator engine called Scenebeans based on Timed Automata semantics. They introduce *behavior beans* for actions and behaviors (for modeling system operations), as well as graphical components called SceneGraphs that represent the system state. A script language based on XML is introduced and used to build animations. Scenebeans is a flexible general purpose framework for animations. However, the problem of animation sequence generation in not tackled. In [14], Scenebeans is applied to an air traffic control case study (Short Term Conflict Alert), and historical data are used as animation sequences. ViBBA has functionalities similar to those provided by Scenebeans, but is much easier to use because it allows visual construction of animator panels.
The tool Possum [16, 10] can be used to animate Z specifications. Graphical interfaces using TCL/TK can be easily implemented depending on the specification to animate. [16] presents a systematic approach to plan, document, and maintain animation scenarios starting from Z formal specifications. While in our approach the derivation of animation scenarios is automatically performed, by using Possum the user must follow some guidelines to manually derive animation scenarios.

5 Conclusions and Future Work

In this paper, we have presented a tool supporting automatic model driven graphical animation, an approach to animate requirements specification. Animation is useful to better understand requirements and to gain confidence of correctness of their specification. Automatic model driven animation minimizes user effort to build those scenarios able to animate all the critical system behaviors. We provide a graphical framework to build animator panels with ease. The user chooses graphical animator elements (buttons, lights, etc.) from a palette, connect them to specification variables, and place them on a pane. Initially, the palette offers only a limited set of elements, but the user could introduce new graphical items. Thanks to CORBA, ViBBA allows the animation of scenarios in a distributed manner.
We have started to extend ViBBA for classical software visualization, animating algorithms like sorting algorithms. In this case scenarios are generated instrumenting the original code and new graphical elements for displaying array content are defined.

Acknowledgments ViBBA implementation was mainly done by our students Sonia Andronaco and Sandra Caprino Campana.

References

[1] J. Bicarregui, J. Dick, B. Matthews, and E. Woods. Making the most of formal specification through animation, testing and proof. *Science of Computer Programming*, 29(1–2):53–78, July 1997.

[2] E. Börger, E. Riccobene, and J. Schmid. Capturing requirements by abstract state machines: The light control case study. *Journal of Universal Computer Science*, 6(7):597–620, July 2000.

[3] A. Cavarra and E. Riccobene. Simulating UML statecharts. In R. Moreno-Diaz and A. Quesada-Arencibia, editors, *Formal Methods and Tools for Computer Science - Eurocast 2001*, pages 224–227, 2001.

[4] P.-J. Courtois and D. L. Parnas. Documentation for safety critical software. In *Proc. 15th Int'l Conf. on Softw. Eng. (ICSE '93)*, pages 315–323, Baltimore, MD, 1993.

[5] A. Gargantini and S. G. Formaggio. ATGT: Asm test generator tool. In B. Thalheim and W. Zimmerman, editors, *Abstract State Machine Conference, ASM 2004*, 2004.

[6] A. Gargantini and E. Riccobene. ASM-based testing: Coverage criteria and automatic test sequence generation. *Journal of Universal Computer Science*, 7(11):1050–1067, Nov. 2001.

[7] A. Gargantini and E. Riccobene. Automatic model driven animation of scr specifications. In M. Pezzé, editor, *Fundamental Approaches to Software Engineering, 6th International Conference, FASE 2003, Part of ETAPS 2003*, number 2621 in Lecture Notes in Computer Science, pages 294–309. Springer, 2003.

[8] A. Gargantini, E. Riccobene, and S. Rinzivillo. Using spin to generate tests from asm specifications. In *ASM '03*, volume 2589 of *LNCS*, pages 263–277. Springer, 2003.

[9] Y. Gurevich. Evolving algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–37. Oxford University Press, 1994.

[10] D. Hazel, P. Strooper, and O. Traynor. Requirements engineering and verification using specification animation. In *Thirteenth International Conference on Automated Software Engineering*, pages 302–305. IEEE Computer Society Press, 1998.

[11] C. Heitmeyer, J. Kirby, B. Labaw, and R. Bharadwaj. SCR: A toolset for specifying and analyzing software requirements. In *Proc. 10th International Computer Aided Verification Conference*, pages 526–531, 1998.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[13] E. Kazmierczak, M. Winikoff, and P. Dart. Verifying model oriented specifications through animation. In *Asia Pacific Software Engineering Conference*, pages 254–261. IEEE Computer Society Press, 1998.

[14] J. Magee, J. Kramer, B. Nuseibeh, D. Bush, and J. Sonander. Hybrid model visualization in requirements and design: A preliminary investigation. In *Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD-10)*, Nov. 2000.

[15] J. Magee, N. Pryce, D. Giannakopoulou, and J. Kramer. Graphical animation of behavior models. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 499–508. ACM Press, June 2000.

[16] T. Miller and P. Strooper. Animation can show only the presence of errors, never their absence. In *Proc. of the 2001 Australian Software Engineering Conference (ASWEC 2001)*, pages 76–85. IEEE Computer Society, 2001.

[17] I. Parissis. A formal approach to testing lustre specifications. In *1st International IEEE Conference on Formal Engineering Methods, Hiroshima*, pages 91–100, 1997.

[18] J. Schimd. Executing ASM specifications with AsmGofer. http://www.tydo.de/AsmGofer.

SIS ASM Specification

SIS is a simplified version of a control system for safety injection in a nuclear plant [4], which monitors water pressure and injects coolant into the reactor core when the pressure falls below some threshold (`Low`). The system operator may override safety injection by pressing a “Block” switch and may reset the system after blockage by a “Reset". To specify the requirements of the control system, we use the monitored variables (updated by the environment) `WaterPressure`, `Block` and `Reset`, and the output controlled variable (updated by the rules) `SafetyInjection` to denote the controlled quantity which can be `on`, `off`. To specify the modes the system can be according to the `WaterPressure` values, we introduce the internal controlled variable `Pressure`, which can take values `TooLow`, `Normal`, `High`. A drop in water pressure below a constant `Low` causes the `Pressure` to become `TooLow` and the activation of the safety injection if the system is not overridden. The controlled variable `Overridden` is *true* if safety injection is blocked, *false* otherwise. The controller policy is simple: `SafetyInjection` is `on` when `Pressure` is `TooLow` and `Overridden` is *false*; it is `off` otherwise.

```
MONITORED VARIABLES
Block : {off,on}; Reset : {off,on};
WaterPressure: 0..2000;

CONTROLLED VARIABLES
Overridden : boolean;
SafetyInjection : {on,off};
Pressure : {TooLow, Normal, High};

STATIC VARIABLES
    Low = 900;    Permit = 1500;

RULES
R1: if WaterPressure >= Low and
       Pressure = TooLow
    then Pressure := Normal
R2: if WaterPressure >= Permit and
       Pressure = Normal
    then Pressure := High;
         Overridden := false
R3: if WaterPressure < Low and
       Pressure= Normal
     then Pressure := TooLow
R4: if WaterPressure < Permit and
       Pressure = High
    then Pressure := Normal;
         Overridden := false
R5: if Reset = on and
    (Pressure = TooLow or
     Pressure = Normal)
    then Overridden := false
R6: if Block = on and
      Reset = off and Pressure = TooLow
    then Overridden := true
R7: if Pressure = TooLow
    then if Overridden
         then SafetyInjection := off
         else SafetyInjection := on
R8: if Pressure != TooLow
    then SafetyInjection := off
```