AniMAL
A user interface prototyper and animator for *MAL* interactor models

Nuno Guerreiro
pg10965@mail.uminho.pt

Sandrine Mendes
pg10968@mail.uminho.pt

Vítor Pinheiro
pg11965@mail.uminho.pt

José Creissac Campos
jose.campos@di.uminho.pt

Departamento de Informática/CCTC, Universidade do Minho
Braga, Portugal

Resumo

Engineering correct software is one of the grand challenges of computer science. Practical design and verification methodologies to ensure correct software can have a substantial impact on how programs are built by the industry. As human-machine systems become more functional, they also become more complex. Consequently, the interactions between the machine and its users becomes less predictable and more difficult to analyse. Using Model Checking it is possible to automatically analyse the behaviour of a modelled system. Hence, different authors have investigated the applicability of model checking to the analysis of human-machine interactions.
The IVY workbench is a tool that supports system design and verification, by providing a model checking based integrated modelling and analysis environment. The tool is based around a plugin architecture, and although it features a verification results' analyser, it thus far lacked the ability to visually expose the sequence of events that lead to a system failure on a system's prototype. We propose the AniMAL plugin as an extension to the IVY workbench, providing automatic user interface prototyping and verification results' animation, while allowing thorough customisation.

Palavras-Chave

dialogue analysis tool, user interface specifications, visual representations, prototyping

1 Introduction

Interactive systems design is a very challenging undertaking, considering the multiplicity of areas that need to be taken into account during the design and verification processes. End-users and context are crucial factors that must be taken into account, in order to create a successful system. There are several disciplines involved, each one with different inputs, from psychology to system analysis, from design to ergonomics, to name a few.

Developing interactive systems will always be a complex endeavour. As the development process advances, it is increasingly difficult and expensive to introduce changes, in order to correct errors or comply with new requirements. Therefore, system analysis should be performed as early as possible, providing a broader view of the problem, while enabling a more effective design.

Creating abstract models is a common engineering practice. It enables reasoning over a system's design, by breaking the problem into smaller, understandable parts, while eliminating irrelevant aspects. Models are also easier to handle than the complete system, which makes it possible to perform a more thorough analysis of the modelled aspects. Models can be expressed using different modelling languages, by capturing and defining relevant aspects of a system and creating a comprehensible representation.

Formal specification of interactive systems provides a way to analyse the consequences of systems design and thereby reduces the risks of interface design. Concerning systems behaviour, model checking has gained particular relevance. This is also the case for interactive systems. For example, Paternò [9] uses model checking for the analysis of dialogue properties of device models specified as interactors. Campos and Harrison use system models specified as MAL interactors [4] for mode confusion analysis, using SMV [1]. See [3, 7, 10] for some examples of recent work.

An interactor is a particular type of modelling artifact that focuses on human interface components of the system and can be used in a verification process. The IVY workbench uses the Interactor concept as developed in [4]. In tis light, an interactor consists of an object capable of rendering

(part of) its state into some presentation medium. MAL interactors support the notion of action, and have their behaviour expressed in Modal Action Logic [1]. MAL interactor models can have their behaviour verified through model checking in the IVY workbench.

Model checking, although an automated verification technique, can be a complex task, as it involves developing apropriate models, expressing relevant properties, and interpreting the result of the checking process. There are tools that support these steps, easing the verification process. The IVY (*Interactors VerifYer*) project aims to provide one of such tool for the specific case of interactive systems' analysis: the IVY workbench. It is based on the Java Plugin Framework and provides a pluggable and extensible architecture.

This paper describes the design and development of a tool that aims to complete that application, by providing user interface prototyping and user interface prototype animation.

1.1 Context

Computers have become almost essential for most daily functions. The popularity of the computer, increased by easier access to the Internet, has won a huge number of distinct types of users, who use it as an essential tool, on a daily basis. We need to create computer systems that justify the trust that society increasingly places on them. Developing software is such a difficult task that often unpredictable failures of systems arise. It is hard to determine requirements, expect interactions and consider new functionality, while maintaining previous behaviour.

Due to the wide recognition of software verification, tools have been created to prove software correctness and to understand how software works, in order to achieve sustained reliability. As software verification becomes a challenge, tools are needed to automatically guarantee that programs meet given specifications [6].

Model checking is a systematic approach for verifying system property correctness. This technique is based on models that represent the target system. Such models describe the set of properties to be proven, for a given system module. A property defines expected behaviour throughout time; the system meets the property if each execution's result matches the expected behaviour. Verification tools either inform that a model satisfies a property's formula or show why the formula fails to prevail, on the given model. These counterexamples are particularly helpful in identifying errors in the model or the system. To reach a completely automatic approach, it may be necessary for the model checking algorithm to traverse all reachable states of the system. This is only possible if state space is finite [8].

1.2 Motivation and Objectives

Despite providing functionality to create models and properties, and to simplify their verification and failure detection, the IVY workbench still lacks the possibility to generate a user interface prototype in order to expose detected interaction failures.

Currently, the tool application already enables failure explanation. Its Traces Analyzer plugin is able to display the sequence of events of a failure, by animating a state machine and showing state tables. However, this approach does not provide a clear explanation of what happens during a system failure, including state transitions or state values. There is no way to visualize this changes on a user interface. As interactors are intrinsically related with user interface components, it is important to provide a preview of what the system would look like. Errors analysis on the user interface could give more real perception.

Our goal is to create a tool that is able to create user interface prototypes automatically, by using the interactor models already handled by the Interactor Editor. Furthermore, the tool needs to allow for customisation, in order to create a more realistically-looking user interfaces. Finally, this tool must be able to animate system failures, on the created user interface prototype.

1.3 Article structure

The paper is organised as follows. Section 2 describes the IVY project. In Section 3 we discuss the AniMAL plugin: an addition to the existing IVY workbench application that enables user interface prototyping and animation for MAL interactor models. Section 4 presents an example, illustrating AniMAL's use. Finally, Section 5 draws some conclusions and outlines future work.

2 The IVY workbench

The IVY workbench (see figure 1) is an output of the IVY project, financed by FCT and FEDER, under contract POSC/EIA/56646/2004. This project addressed the development of models of the interactive devices, and their verification through model checking against properties that encode assumptions about usability of the device. The models are developed using the MAL Interactors language, and the properties expressed in CTL (Computacional Tree Logic). Verification is performed resorting to the NuSMV model checker.

The tool was developed around a plugin framework (the IPF – Interactors Plugin Framework). The framework stores and makes available information about the models and the verification process. Appropriate plugins can then be connected to the IPF to provide the modelling and verification services to the users of the tool. Figure 1 illustrates the typical workflow of the tool. The diagram includes the AniMAL plugin, which is the subject of this paper, and not part of the original set of plugins.

The models can be created using IVY workbench's Model Editor plugin (see figure 2). This editor allows the user to input MAL models textually or to create a class-like diagram and generate the MAL code automatically.

The Property Editor plugin enables the user to express properties that must be respected by the model's behaviour. The plugin includes a set of patterns that can be instanti-

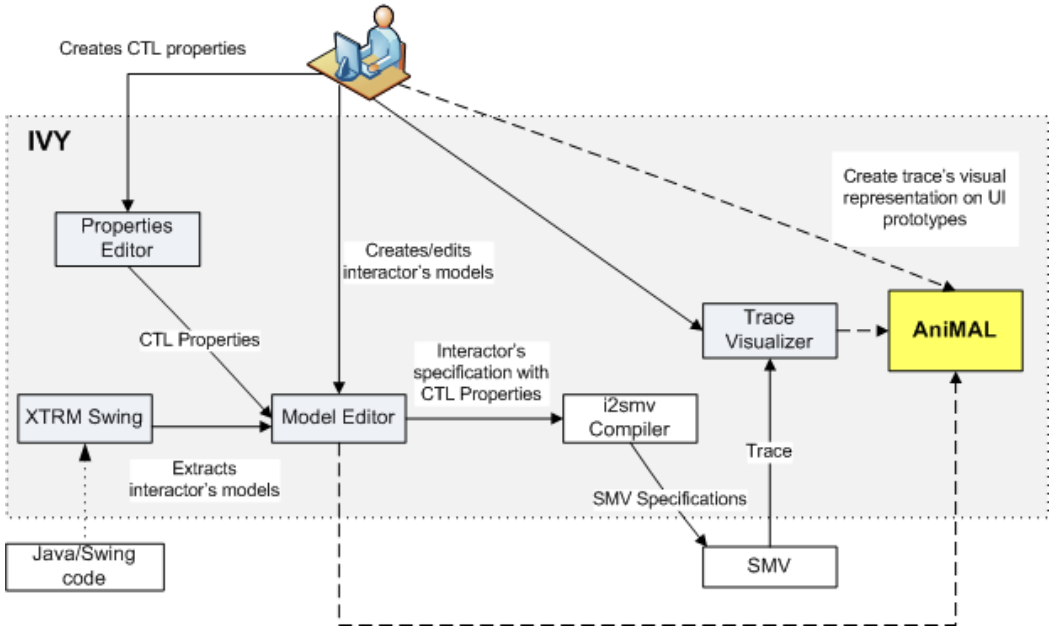


Figure 1. IVY workbench workflow

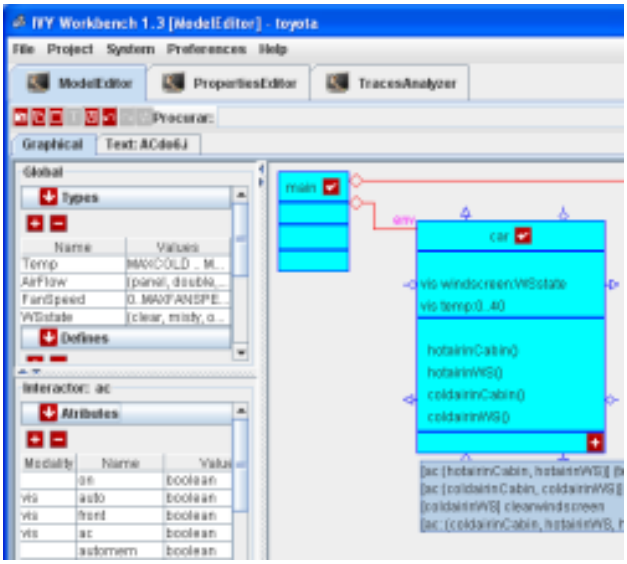


Figure 2. IVY workbench's editor (detail)

ated to create these properties for verification. Detailing the patterns is out of the scope of the paper, see [2] for an overview.

Once a set of properties is defined, the integrated SMV model checker can be used in order to validate them. If the verification fails, the verification process typically produces a trace, a sequence of states, illustrating a behaviour of the model that falsifies the property. Depending on the model complexity, the traces can become very large, and their analysis time consuming and complex.

A Trace Visualiser plugin is provided to support the analysis of the traces. The plugin eases this analysis, by expressing the fail traces in a number of different representations. These include a tabular representation, where rows represent interactor attributes and columns the different states in the trace; a state transition-like representation, showing the different states of each interactor and the actions causing the transitions between them; and an activity diagram representations, where the main focus is on the actions of the trace. Figure 3 illustrates the different representations.

Although the Traces Visualizer plugin provides basic animation functionalities, they work at the level of the trace (for example, sequencially highlighting the different states of a trace), and not of the interface. That is to say that there is no support in the plugin to analyse the problem by means of directly inspecting (a prototype of) the user interface.

3 Our Proposal

The AniMAL plugin is an addition to the existing IVY workbench suite (see appendix A for a simplified class diagram). It enables user interface prototyping, by fetching interactor data from the suite and generating a default, yet customisable, user interface prototype. That prototype can be animated, in order to present the sequence of events recorded by a fail trace.

In this section we discuss the implementation and features of the plugin.

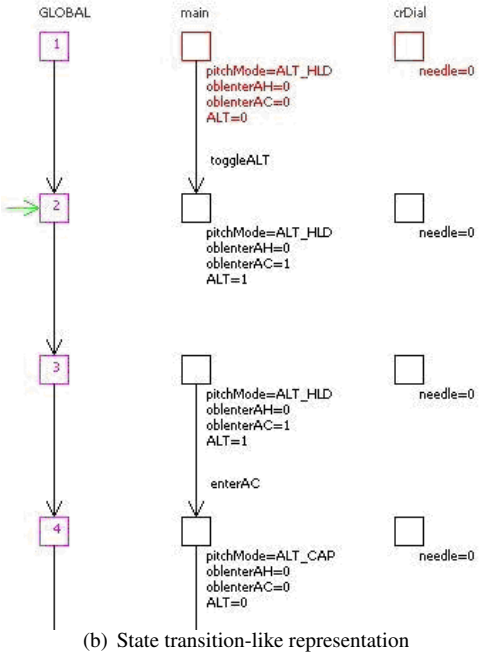
3.1 Data integration

The IVY workbench's plugin framework (IPF) has its own data repository. It acts as a broker, providing all plugins with the ability to store and query interactor information. Currently, we are using interactor models published by the Model Editor, as well as traces provided by the Traces Analyser, as can be seen in Figure 4.

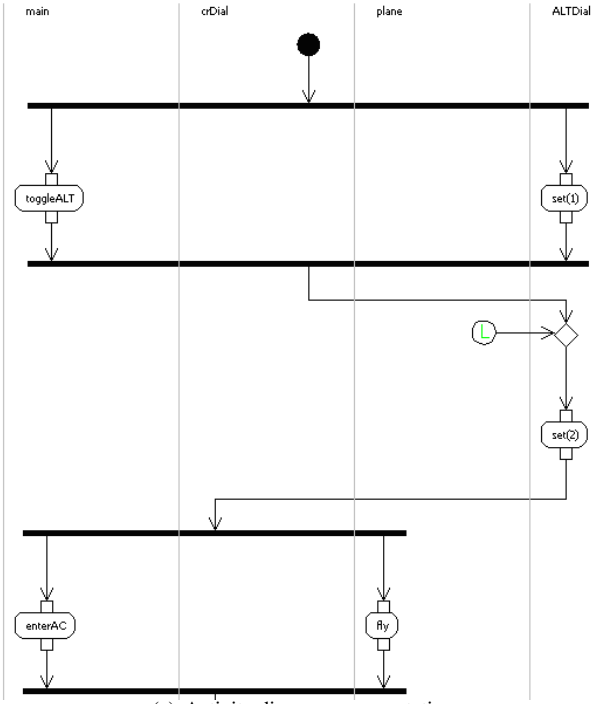
Whenever AniMAL detects that new information is available, a new data model is generated, containing the loaded

	1	2	3
main.ac	0	0	0
acmem	0	0	0
action	airintakekey	autokey	fanspeedup
airflow	wsclear	floorws	floorws
airflowmem	wsclear	wsclear	wsclear
airintake	0	0	0
airintakemem	0	0	0
auto	0	1	0
automem	0	1	1
fanspeed	0	10	10
front	1	0	0
on	0	1	1
settemp	15	15	15

(a) Tabular representation



(b) State transition-like representation



(c) Activity diagram representation

Figure 3. Different trace representations

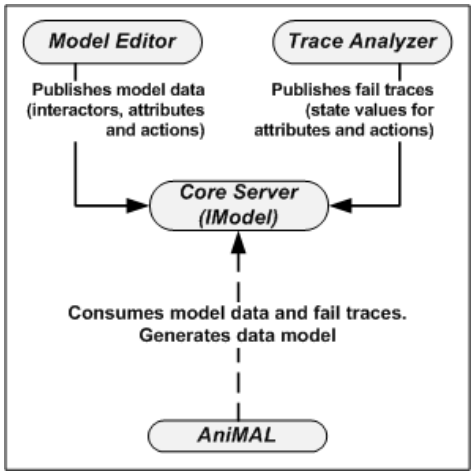


Figure 4. Data integration

interactor and trace data.

Interactor data includes the list of interactors, their hierarchical relationships, its visible attributes and their possible values, as well as visible actions. Using the lists of values, AniMAL infers data types, in order to determine how to represent each attribute.

Trace data is queried directly on IPF's repository and contains the attributes' and actions' values, for each of the fail trace's states.

3.2 Interactor to User Interface Mapping

Due to the hierarchical nature of interactors, AniMAL represents them as panel components, grouping their own attributes' and actions' representations.

Rendering attributes requires increased flexibility. Firstly, an attribute may need to be rendered with one of several components, in order to have its values represented more realistically or conveniently. For example, we might wish to include a label next to an input field. Secondly, there are several data types for storing values. For instance, boolean attributes require different components than integer attributes. Finally, rendering attributes requires the user interface prototype to communicate with the data model, to synchronise values. Our approach requires each widget to implement a set of interfaces, making it possible to write new widgets and use them to represent attributes, without requiring changes to the plugin itself (appendix B shows a sample widget's code).

For simplicity's sake, actions are rendered as toggle buttons. An action can currently be seen as a boolean attribute as well, whose value indicates whether or not the action is being executed. This allows us, when animating a trace, to signal which action has just been executed, at each step in the animation.

3.3 User Interface Generation

From the extracted data model, AniMAL creates a tree, containing all interactors and their attributes and actions. It is possible to drag elements from that tree to the view port,

in order to create a visual representation of the interactor, attribute or action. Furthermore, an automatic prototype generation can be requested, rendering all elements using the default components (see Figure 5).

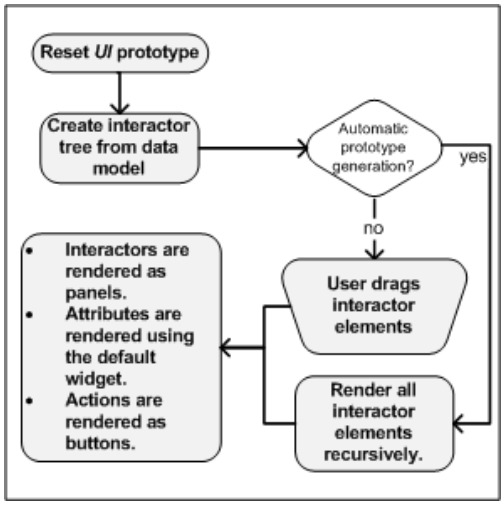


Figure 5. User Interface Mapping and Generation

3.4 User Interface Architecture

Once rendered, elements are placed hierarchically, using the main interactor's panel as the root, on the view port. The prototype is built around the known Hierarchical Model-View-Controller (or HMVC) pattern, using the *Tikeswing* framework [11]. Each interactor is represented by a Model-View-Controller (or MVC) set. The *View* aggregates graphical elements, like child interactors' views, as well as attributes' and actions' widgets. The *Controller* handles view events, like mouse drags and keyboard events, as well as View-Model synchronisation. Finally, we use a centralised *Model*, to store attributes' and actions' values (see Figure 6).

Using the HMVC pattern, values are transparently synchronised between model and view, eliminating the need for complex event handling and user interface component dependent value updates. Therefore, a change on an attribute's value on the model is automatically reflected on the view, on the corresponding widget.

Both the user interface and data elements are centrally managed by the InteractorManager. This is the entry-point for adding new data elements to the model (during data extraction), to generate a user interface prototype, saving and loading, as well as resetting the environment, prior to loading a new model.

Whenever new data elements are added to the model, the InteractorManager updates the hierarchical structure, and its tree-like graphical representation, while maintaining a centralised registry. This entity provides the methods to render interactors, attributes and actions using the default components, as well as to replace the widget being used for rendering a given attribute. Components are created via reflection, using the classes defined on the configuration file, allowing the plugin to be extended with other widgets without the need of recompilation.

3.5 Customisation

In order to adapt the prototypes to different situations, AniMAL allows the user to select different customisations, creating a more realistic user interface. In this section we cover the three levels of customisation that are possible: changing the layout management algorithm; changing the widget associated to a given attribute; and changing the rendering properties of a given widget.

3.5.1 Layout Management

Layout management can be performed manually or automatically. In manual mode, graphical elements can be manually disposed, using drag-and-drop options, for every panel or widget. Dimension can also be changed for every element, after boundary validation.

As a convenience, and in order to ease prototype creation, AniMAL enables the use of automatic layout managers. These layout managers are responsible for resizing and changing positions for every element, according to their own algorithms. Currently, AniMAL provides two simple layout managers that spread components on a grid: one that places all elements side-by-side, and another that render interactors first, then attributes, then actions, from top to bottom. These are simply proofs of concept, and additional layout managers can be created.

If new layout managers need to be created, and in order to shield the plugin's code from thorough modification, AniMAL relies on a factory (a well know pattern [5]), which makes layout managers available for the user to choose, while decoupling the plugin's user interface from specific layout manager implementation. This way, a new layout manager can be implemented, using a specific interface, and only the factory needs to be slightly changed.

3.5.2 Widget interchangeability

Attribute rendering widgets are required to implement a set of interfaces, in order to make them interchangeable and maintain View-Model synchronisation. This allows the plugin to be widget-agnostic, as all widgets are treated alike. Most of the methods included on those interfaces are already implemented by Java Swing and Tikeswing components, so creating new widgets is fairly simple.

When a widget is replaced, the data model is unchanged and therefore the new widget represents the same value as the old one.

3.5.3 Properties

A user interface editor would not be complete without the ability to change graphical elements' properties, like colours, captions, tooltips, and many other parameters. AniMAL uses the L2FProd Common Components library, which makes it possible to change most visual aspects of a Java Swing component.

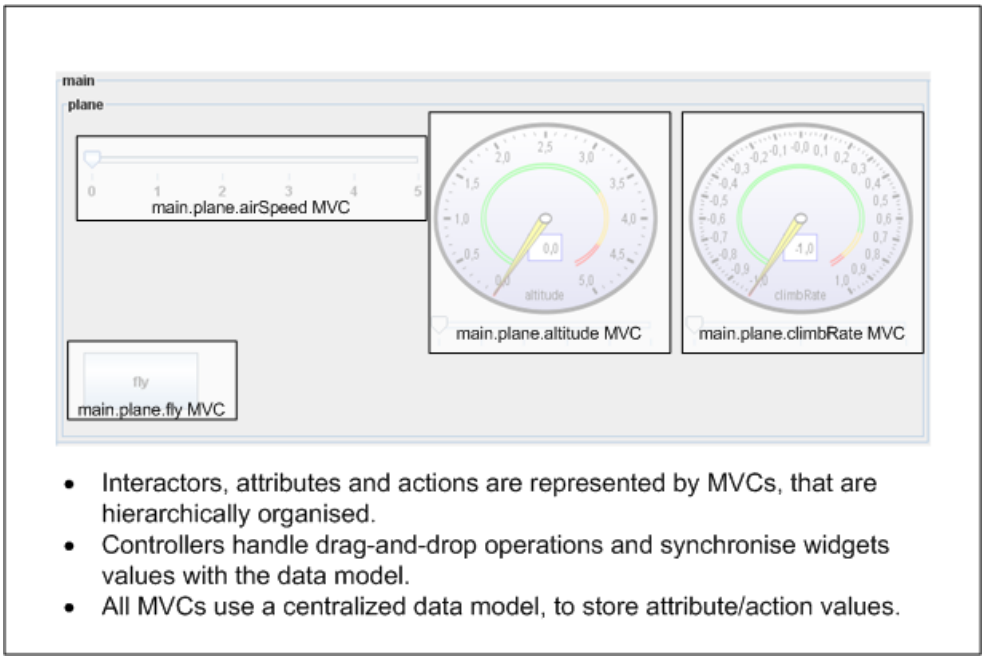


Figure 6. Sample User Interface prototype, highlighting some HMVC aspects

3.6 Fail trace animation

Each fail trace includes a set of values for each attribute and action, for each state. Animating a fail trace is done by changing the data model to sequentially match each state in the trace. By using the HMVC design pattern, only the data model needs to be changed, while graphical elements are automatically updated, due to View-Model synchronisation, provided by the Tikeswing framework.

3.7 Miscellaneous

An additional set of features can currently be found on the AniMAL plugin, as described below.

3.7.1 Configuration

AniMAL uses a set of configuration files to allow the user to change the behaviour and visual aspects of the plugin. It is possible to change several parameters, including drag-and-drop highlight colour, default panel size, default layout parameters, as well as fail trace animator's time parameters.

The list of widgets that can be used for each data type is also configurable. As has already been discussed, new widgets can be added, without requiring plugin's code compilation. This is achieved by including the widget's class name on the list of possible widgets for appropriate data types. Appendix C shows an example of a configuration file.

3.7.2 Persistence

AniMAL provides user interface prototype persistence, by allowing the user to save prototypes to XML files. Those files can be loaded afterwards, recreating the user interface.

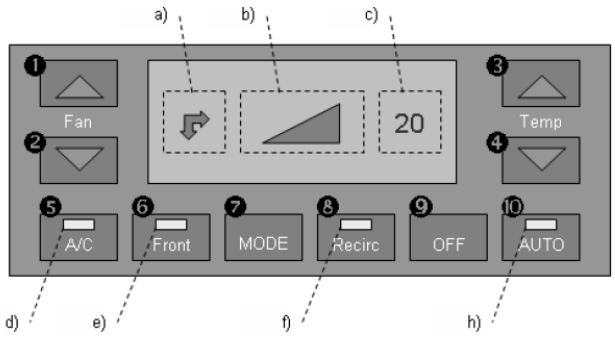


Figure 7. The air-conditioning control panel

3.7.3 Internationalisation

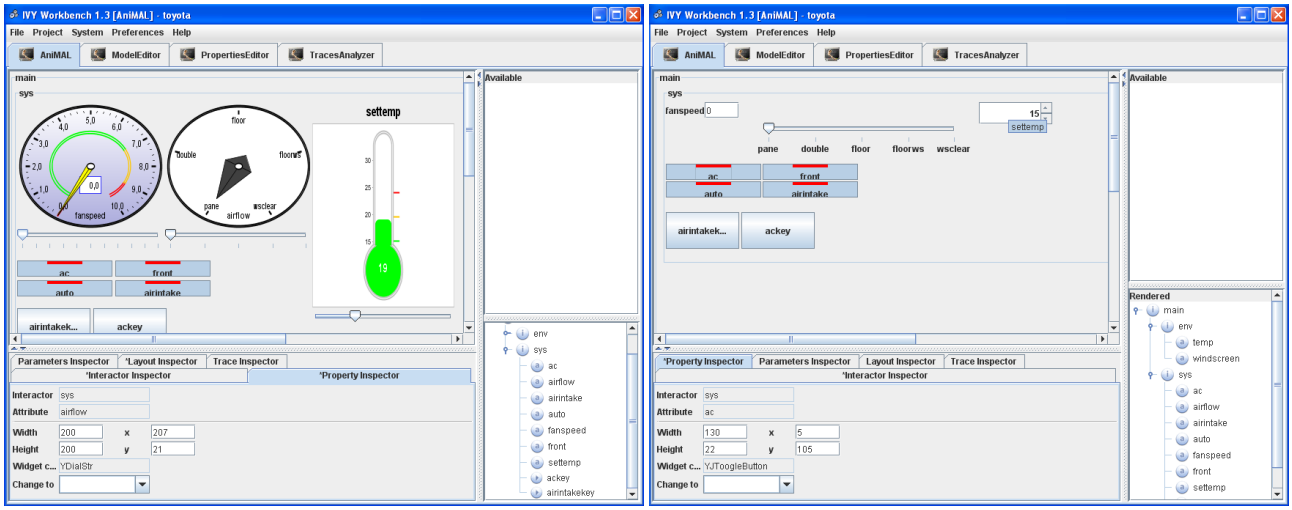
All labels and messages used in the plugin can be translated onto different languages, without the need for code recompilation. AniMAL uses a resource manager that loads labels and messages from resource bundles, according to the user's locale.

4 An illustrative example

In this section we present an example, describing how AniMAL can be used to create and animate a prototype of a MAL interactors model.

The example is adapted from [3]. It consists of a car's automatic air-conditioning (A/C) panel user interface. Figure 7 shows a representation of the actual user interface of the panel. The A/C panel has a total of 10 buttons and 7 display items of interest. The buttons are the following (numbers are used to identify the buttons in the figure, names in parenthesis will be used in the model):

1. increase fan speed button (fanspeedup)



(a) Prototype 1 (b) Prototype 2

Figure 8. Two customised user interface prototypes

2. decrease fan speed button (fanspeeddown)
3. increasing target temperature button (tempup)
4. decreasing target temperature button (tempdown)
5. air conditioning mode selection button (ackey)
6. windscreen (front) flow mode selection button (frontkey)
7. flow mode selection button (modekey)
8. air intake mode selection button (airintakekey)
9. off button (off)
10. automatic mode button (autokey)

4.1 The model

The following partial MAL Interactor describes the set of attributes and actions that model the A/C panel:

```
types
 Temp = MAXCOLD .. MAXHOT
 AirFlow = {pane, double, floor, floorws, wsclear}
 FanSpeed = 0..MAXFANSPEED
defines
 MAXCOLD = 15
 MAXHOT = 30
 MAXFANSPEED = 10
interactor sys
 attributes
        on : Boolean
  [vis] auto, airintakefresh, front, ac : Boolean
  [vis] settemp : Temp
  [vis] airflow : AirFlow
  [vis] fanspeed : FanSpeed
 actions
        autokey off modekey fanspeedup
        fanspeeddown tempup tempdown frontkey
  [vis] ackey airintakekey
```

The *sys* interactor aggregates attributes and actions that define the visible parameters and available actions. Attributes include the temperature to set (*settemp*), the fan speed (*fanspeed*), the active air flow ducts (*airflow*), as well as others, like whether or not fresh air should be introduced into the cockpit (*airintakefresh*) and whether or not the air conditioning is on (*ac*). Attributes' values' domains are set using the user-defined *Temp*, *AirFlow* and *AirSpeed* data types. Available actions include changing the air speed, temperature, as well as activating the air conditioning, for instance.

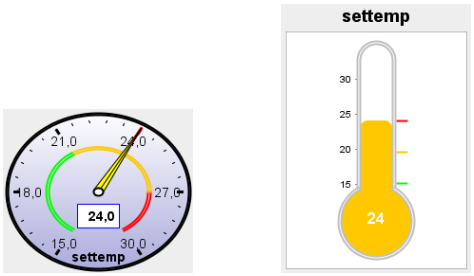
We will not delve into extensive detail of the language or the model, as our goal is not to describe the interactor, but its representation. In particular, the axioms of the model are not presented since they are relevant to the discussion that follows.

Figure 8 presents AniMAL showing two customised prototypes, for the previous model. The prototypes were automatically generated and then some custom widgets were chosen, in order to adapt it to this specific case study. However, no adaptation would be required, as long as the configuration stated that the chosen widgets were the default ones. The *sys* interactor is rendered as a panel on the viewport, inside the *main* interactor's panel. Each attribute is rendered using a widget. Besides usual widgets like toggle buttons (for boolean attributes), some custom widgets were used, like a thermometer for displaying the *settemp* attribute and a dial, for representing the *airflow* attribute, while actions are rendered as toggle buttons.

There are endless possibilites to creating a user interface prototype. Figure 9 shows some of the widgets that have been implemented, and can be used to render attributes like *settemp*.

4.2 Animation

Using traces imported from the Traces Analyzer plugin, AniMAL is able to animate sequences of events, making it easier to see the states transitions and value changes that violate the property under scrutiny. Figure 10(a) shows



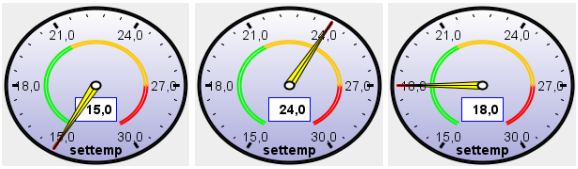
(a) YDial (b) Thermometer

(c) YJSpinner

Figure 9. An example of different attribute representations

	1	2	3
sys.settemp	15	24	18
sys.airflow	pane	pane	pane
sys.fanspeed	2	4	2

(a) Tabular representation (traces)



(b) state 1 (c) state 2 (d) state 3

Figure 10. An example of prototype animation based on traces

one of Traces Analyzer's trace visualisation methods (tabular) representing how the *settemp*, *airflow* and *fanspeed* attributes change in the particular trace. Figure 10(b), Figure 10(c) and Figure 10(d) show the *settemp* attribute changing its value, based on the fail trace. As the value changes, so does the widget's rendering. This is the principle behind AniMAL's trace animation: attributes and actions change their values throughout time and the user interface representation changes accordingly.

5 Conclusions and Future Work

System design and development is a complex undertaking that involves knowledge of multiple disciplines. As such, it is often difficult to predict system behaviour and guarantee correctness. Model checking approaches help unveiling potential interaction failures, allowing errors to be found before becoming excessively expensive to deal with. This requires automated tools, to help in understanding the system and detecting its potential design flaws.

The IVY workbench supports system design and verification, by providing a model and properties editor as well as a fail trace analyser. The *AniMAL* plugin extends this application, by providing user interface prototyping and fail trace animation capabilities. *AniMAL* enables the creation and customisation of a user interface prototype based on a model, in order to visually expose the sequence of events that lead to a system failure. Additionally, the plugin allows for automatic user interface prototype generation, freeing the user from unnecessary work.

At this point, there are still much work that can be done to improve the plugin. We list some of the possible improvements and areas for future work:

- The widget library could be extended, in order to provide a wider set of user interface representations, thus supporting the creation of more realistic user interfaces.
- Additional layout managers could be developed, in order to automatically generate different types of layouts.
- Undo management is still to be developed and could improve the user's experience.

Finally, providing a fully functional user interface, one capable of accepting user input and behaving according to the axioms in the model, would also be an important addition. The plugin could ultimately generate application skeletons, from MAL interactor models, in order to go from a prototype into a full-blown user interface. It would be also interesting to create user interfaces for different visualisation platforms (e.g. web sites), by using the generated XML file (exported using the save operation).

Acknowledgments

The authors would like to thank Nuno Sousa, for developing the data integration mechanisms that allow *AniMAL* to query the *InteractorEditor*'s data model. Also, Nuno Guerreiro (*Alert Life Sciences Computing*), Sandrine Mendes (*Alert Life Sciences Computing*) and Vítor Pinheiro (*Enabler, a Wipro Company*) would like to thank their employers for sponsoring this work.

References

[1] J. C. Campos and M. D. Harrison. Model checking interactor specifications. *Automated Software Engineering*, 8(3/4):275–310, August 2001.

[2] J. C. Campos and M. D. Harrison. Systematic analysis of control panel interfaces using formal tools. In *XVth International Workshop on the Design, Verification and Specification of Interactive Systems (DSV-IS 2008)*, number 5136 in Lecture Notes in Computer Science, pages 72–85. Springer-Verlag, July 2008.

[3] J.C. Campos and M.D. Harrison. Considering context and users in interactive systems analysis. In *Engineering Interactive Systems 2007*, Lecture Notes in Computer Science. Springer-Verlag, 2007. to appear.

[4] D. J. Duke and M. D. Harrison. Abstract interaction objects. *Computer Graphics Forum*, 12(3):25–26, 1993.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[6] Cliff B. Jones, Peter W. O'Hearn, and Jim Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[7] K. Loer and M.D. Harrison. Analysing user confusion in context aware mobile applications. In M. Constabile and F. Paternò, editors, *Interact 2005*, volume 3585 of *Lecture Notes in Computer Science*, pages 184–197. Springer, 2005.

[8] Karsten Loer. *Model-based Automated Analysis for Dependable Interactive Systems*. PhD thesis, The University of York, August 2003.

[9] F. D. Paternò. *A Method for Formal Specification and Verification of Interactive Systems*. PhD thesis, Department of Computer Science, University of York, 1995.

[10] M. ten Beek, M. Massink, and D. Latella. Towards model checking stochastic aspects of the thinkteam user interface. In M. Harrison and S. Gilroy, editors, *Proceedings 12th International Workshop on the Design, Specification and Verification of Interactive Systems*, volume 3941 of *Lecture Notes in Computer Science*, pages 39–50. Springer, 2006.

[11] Tomi Tuomainen. *Tikeswing - User's Guide*. Ministry of Agriculture and Forestry of Finland, 2007.

A Class diagram

Figure 11 shows a class diagram with the most relevant classes developed. Due to the extension of such diagram, only the manager classes are fully described. These are the classes responsible for adding new elements to the *UI* prototype.

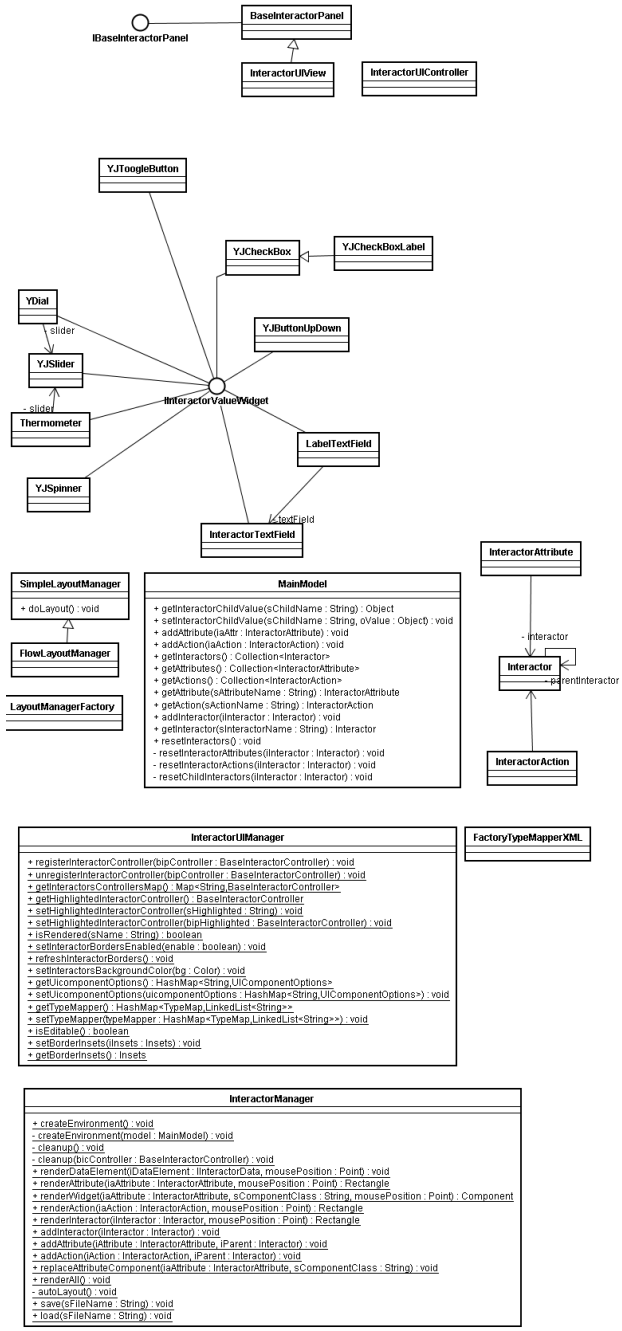


Figure 11. Class diagram, showing the most relevant classes

B A sample widget

Creating a widget for use in *AniMAL* is fairly simple. The following (Listing 1) is an example of a graphical element that was adapted to become a widget. In order to be a valid widget, a graphical element's class must implement *IInteractorValueWidget* interface. This includes methods for getting and setting the widgets value (*setModelValue* and *getModelValue*), for registering the controller (*addViewListener*), for setting the widget's name (*setName*) and setting the list of values to be represented (*setValuesList*). This particular widget does not need which values will be represented, as it will only be representing boolean values.

Listing 1. A checkbox widget

```
/**
* This class shows how a checkbox can be adapted to act
* as an AniMAL's widget.
*
* @author Vitor Pinheiro
* @since 2008/02/08
* @version 1.0
*/
public class YJCheckBox extends YCheckBox
    implements IInteractorValueWidget {

  private YProperty myProperty = new YProperty();

  /** Value of CheckBox */
  private Object value;
```

```
/** Name */
private String name;

/** Constructor */
public YJCheckBox() {
  super();
  name = "";
}

/**
* Returns the widget's property map, used by
* the tikeswing framework
* @return widget's property map
*/
public YProperty getYProperty() {
  return myProperty;
}

/**
  * Method called by the tikeswing framework, in
  * order to assign a controller to this widget.
  * @param controller Controller for this widget
  */
public void addViewListener
              (final YController controller) {
  /**
  * We create an action listener that calls
  * the controller's synchronisation method
  */
  this.addActionListener(new ActionListener() {
    public void actionPerformed (ActionEvent e) {
      controller.
        updateModelAndController(YJCheckBox.this);
    }
  });
}

/**
 * Method called by the tikeswing framework,
 * to get the widget's value.
 * @return widget's model value
 */
public Object getModelValue() {
  return this.isSelected();
}

/**
* Method called by the tikeswing framework,
* to set the widget's value, using the data model's.
*/
public void setModelValue(Object obj) {
  this.setSelected((Boolean) obj);
}

/**
* Sets the Model-View-Controller name.
* This name should correspond to an attribute/action
* in the model, with the same name
*/
public void setMvcName(String mvcName) {
  getYProperty().put(YIComponent.MVC_NAME, mvcName);
}

/**
* This method is called when the widget is created
* and before being rendered.
* Some widgets need to know which values will
* need to represent (a slider, a dial, etc).
* @param list of values that the widget must represent
*/
public void setValuesList(List listOfValues) {}

/**
* Method called when the widget is created,
* to set its name
* @param widget's name
*/
public void setName(String name) {
  super.setName(name);
  this.name = name;
  setToolTipText(name);
}
}
```

C Using the widget

To start using the widget, it is necessary to configure the *TypeMapper.xml* file, setting the widget to be used for boolean values:

Listing 2. TypeMapper.xml file

```
<BOOLEAN type="java.lang.Boolean">
  <Class id="0">
    pt.uminho.di.msc.AniMAL.ui.widgets.YJToogleButton
  </Class>
  <Class id="1">
    pt.uminho.di.msc.AniMAL.ui.widgets.YJCheckBox
  </Class>
  <Class id="2">
    pt.uminho.di.msc.AniMAL.ui.widgets.YJCheckBoxLabel
  </Class>
  <Class id="3">
    pt.uminho.di.msc.AniMAL.ui.widgets.YJSlider
  </Class>
</BOOLEAN>
```