

# **IVY**

## **A model-based usability analysis environment (POSC/EIA/56646/2004)**

### **Um Visualizador de Traços de Comportamento para a Ferramenta IVY**

IVY-TR-5-03

Outubro, 2006

Nuno Miguel Eira de Sousa  
José Creissac Campos



União Europeia  
FEDER







RELATÓRIO DE ESTÁGIO DA  
LICENCIATURA EM  
ENGENHARIA DE SISTEMAS E INFORMÁTICA  
-  
UM VISUALIZADOR DE TRAÇOS DE  
COMPORTAMENTO PARA A FERRAMENTA IVY

Nuno Miguel Eira de Sousa  
Nº 27643

Departamento de Informática da Universidade do Minho

Supervisores:

José Creissac Campos  
António Ramires Fernandes

Universidade do Minho, 2006

## Acknowledgements

I would like to thank my supervisor, José Creissac Campos, for all the support that he has offered me, and for the constant suggestions he gave to improve the tool, IVY Trace Visualiser, which is the subject of this report. Without the weekly reunions we had, the tool would not have achieved the desired performance and functionality. In these reunions, we did small usability analysis of the functionality of the tool to try to find problems and to discover more useful functionalities to add.

I would like to thank my adviser, José Ramires Fernandes, for suggesting the creation of *markers*. It was a great idea to provide a more useful analysis of the visual representations generated by the tool.

I also would like to thank Alexander de Ridder and Francisco Martinez Posadas for the work developed in early versions of the tool (see [Ridder05] for more detail on their work).

The development of the IVY Trace Visualiser tool is supported by Fundação para a Ciência e a Tecnologia (FCT, Portugal) and by the European Regional Development Fund (FEDER) under contract POSC/EIA/56646/2004. So, I would like to thank them for funding the traineeship.

# Contents

Acknowledgements	i
Contents	ii
Figure Index	iv
1. Introduction	1
1.1. Contextualisation	1
1.2. Objectives	3
1.3. Structure of report	4
Part I – Theory and Early Trace Visualiser	6
1. Theoretical Beddings of the Work	7
1.1. IVY	7
1.2. CTL (Computational Tree Logic)	8
1.3. Analysis of MCP example	9
2. Early Trace Visualiser	11
2.1. Architecture	11
2.1.1 Parser	12
2.1.2 Structure	12
2.1.3 Graphics	13
2.1.4 Shared	16
2.2. Graphical Interface	16
2.3. Diagram Representation	17
2.4. Tabular Representation	18
2.5. Tree Representation	19
2.6. User Input in Filters	20
2.7. Bridge between early and final Trace Visualiser	21
2.7.1 Implications in the Code	21
2.7.2 Implications in the Graphical Interface	21
2.7.3 Changes in the Visual Representations	22
2.7.4 New Functionalities	23
Part II – User Manual of the Final Trace Visualiser	24
1. Introduction	25
2. Graphical Interface	26
2.1. Visual Representations	32

2.1.1 Tree	32
2.1.2 Tabular	34
2.1.3 Physical States	35
2.1.4 Logical States	38
2.1.5 Activity Diagram	39
2.2. MCP Example Analysis	40
2.2.1 Markers	41
2.2.2 Filters	44
Part III – Technical Manual for the Final Trace Visualiser	46
1. Introduction	47
2. High Level View	48
2.1. Class Design	49
2.1.1 Visualiser	49
2.1.2 Parser	49
2.1.3 Structure	50
2.1.4 Graphics.shared	51
2.1.5 Graphics.visualisation	54
2.1.6 Graphics.elements	56
2.1.7 Util	56
2.2. How to make a new visual representation?	57
Part IV – Conclusions and Future Work	64
1. Conclusions and Future Work	65
References	67
Acronyms	69

## Figure Index

Figure 1 – IVY architecture with description of each component task.	2
Figure 2 – Trace.	3
Figure 3 – The MCP (adapted from Honeywell Inc., 1988).	9
Figure 4 – Extract of trace counterexample.	10
Figure 5 – Package View.	11
Figure 6 – CommonGraphics Diagram.	14
Figure 7 – FiniteStateCommonGraphics Diagram.	15
Figure 8 – GUI of Early Trace Visualiser.	16
Figure 9 – Diagram representation.	17
Figure 10 – Diagram representation with labels.	17
Figure 11 – Filter “Get values of state 5”.	18
Figure 12 – Tabular representation.	18
Figure 13 – Filter “States where main.pitchmode=IAS or plane.altitude=1”.	19
Figure 14 – Tree representation.	19
Figure 15 – Filter “States where ALTDial.needle changed to 2”.	19
Figure 16 – First and second panels.	20
Figure 17 – Input with state number.	20
Figure 18 – GUI of Final Trace Visualiser	22
Figure 19 – Main frame.	26
Figure 20 – Visual Representations ComboBox.	27
Figure 21 – File Menu.	28
Figure 22 – View Menu.	28
Figure 23 – Markers menu.	28
Figure 24 – Filters menu.	28
Figure 25 – About message.	29
Figure 26 – Markers conditions.	30
Figure 27 – Marker ColorChooser.	30
Figure 28 – Filter conditions.	32
Figure 29 – Tree representation with filter and markers.	33
Figure 30 – Collapsing states.	34
Figure 31 – Showing marker condition.	34
Figure 32 – Tabular representation with filter and markers.	35

Figure 33 – Tabular animation.	35
Figure 34 – Physical States representation with filter and markers.	36
Figure 35 – Physical States animation.	37
Figure 36 – Showing marker information with popups option.	37
Figure 37 – Logical States representation with filter and markers.	38
Figure 38 – Activity Diagram.	39
Figure 39 – Activity Diagram animation.	40
Figure 40 – Verification of property in Physical States representation.	41
Figure 41 – Verification of the property using Physical States.	42
Figure 42 – Verification of the property using Activity Diagram.	43
Figure 43 – Verification of the property conditions using Logical States.	44
Figure 44 – Verification of the property using Logical States.	45
Figure 45 – Package View.	47
Figure 46 – Data structure.	50
Figure 47 – Representation class diagram.	52
Figure 48 – Markers class diagram.	52
Figure 49 – Filters class diagram.	53
Figure 50 – Visualisation class diagram.	54



# 1. Introduction

This report describes the development of a component (Trace Visualiser) for a modular tool named IVY (Interactors VerYfier). The context in which the IVY tool appeared is explained in section 1.1, with text adapted from [Campos04]. Section 1.2 describes the objectives to be achieved in the current work. Finally section 1.3 describes the structure of the current report.

## 1.1. Contextualisation

In the development of interactive systems, the areas of Human-Computer Interaction (HCI) and Software Engineering join. Studies have shown that the success of those systems depend largely on usability. The ISO DIS 9241-11 standard identifies the relevant factors to the usability of a system by defining it as the effectiveness, efficiency and satisfaction with which specified users achieve specified goals in specified environments. Effectiveness has to do with the possibility (or not) that the user can achieve his goals using the system on a given context. Efficiency has to do with the amount of effort that the user has to spend to achieve a goal. Satisfaction is a subjective measure of degree of agreeability in the utilization of the system.

The analysis of the quality of an interactive system, with respect to usability, must take into consideration the users of the system and the context in which it is used. The Software Engineering Book of Knowledge [SWEBOK01] considers the design of user interfaces as an area related to but distinct from software engineering, not mentioning the area of HCI. In practice, the areas of software engineering and HCI have been living relatively apart. Therefore it is necessary to establish contact bridges between the two areas.

In the context of the IVY project (A model based development environment - POSC/EIA/56646/2004) techniques and tools are being developed to facilitate the incorporation of usability concerns into the development of software, promoting the communications between the two communities. The approach is model based and aims at giving a greater autonomy to software engineers when aspects of usability, related with the behaviour of the system, need to be taken into consideration. It also aims at helping software engineers in identifying the points at which help of HCI experts is needed.

The main goal of the project is the development of a modelling and analysis tool, for the detection of potential usability problems early in development of any interactive system. The tool will enable the automated inspection of interactive systems models. The models are obtained by

a modelling process based on an editor or by reverse engineering of user interface code [Silva06]. Analysis is performed using the SMV model checker [McMillan93].

The architecture of the tool (see figure 1) consists of the following components:

- Model editor;
- Properties editor;
- i2smv compiler;
- Reverse engineering component (XTRMSwing);
- Trace Visualiser.

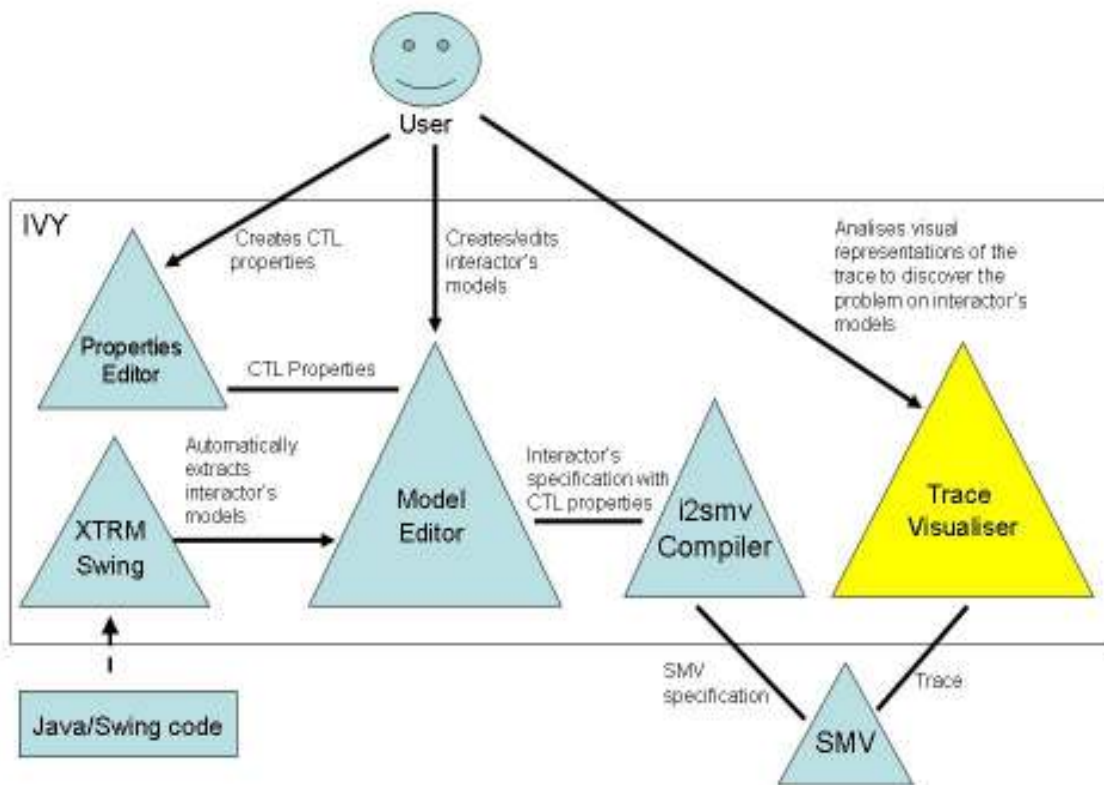


Figure 1 – IVY architecture with description of each component task.

Given an interactive system's model and a property about its behaviour expressed in CTL (for example, some state of the user interface can be reached), the i2smv compiler will produce a specification suitable for analysis by the SMV model checker.

When a property does not hold, the SMV model checker tool usually produces a trace showing behaviour that falsifies the property in question. A SMV trace consists of a sequence of states of the model (see figure 2). In terms of the analysis of interactive systems, this trace can be seen as a usage scenario that falsifies the property under consideration. These scenarios must then be analysed in terms of their cognitive plausibility, and impact on the design.

The Trace Visualiser component is responsible for presenting the results of the verification process to the IVY user in a manner that facilitates understanding the meaning of the trace.



The main objective of the work was the development of a Trace Visualiser component to integrate in the IVY tool. The specific objectives were the following:

- to analyse the architecture of the early Trace Visualiser, in order to see if alterations on the code structure were needed;
- to improve the visual representations that were implemented in the early version of the component;
- to implement additional trace analysis mechanisms;
- to improve the functionality of filters;
- to improve the animation in all the visual representations;
- to improve the graphical user interface;
- to study the possibility of implementing sub-filtering;
- add more standard visual representations.

To fulfil the objectives the means used were:

- Borland JBuilder 2006 Enterprise Evaluation Version to make the implementation in the Java language;
- Visual Paradigm for UML 5.3 Community Edition to create UML diagrams that describe the architecture of the visualiser component.

### **1.3. Structure of report**

The report is divided into four parts: Theory and Early Trace Visualiser, Technical Manual of Final Trace Visualiser, User Manual of Final Trace Visualiser and Conclusions and Future Work.

Part I has the following chapters:

- Theoretical Beddings of the Work – describes some concepts needed to easily understand the rest of the report.
- Early Trace Visualiser - describes an early trace visualiser (developed by me in a previous work) that was the starting point for the current final visualiser.

Part II has the following chapter:

- User Manual of the final Trace Visualiser – intends to be a manual for users of the Trace Visualiser. This manual will explain to users which functionalities they have at their disposal to easily analyse the meaning of a trace.

Part III has the following chapter:

- Technical Manual of the final Trace Visualiser – intends to be a manual for programmers that in the future may want to modify the implementation or add more functionality to the Trace Visualiser. It is expected that it will be sufficient clear to

help programmers understand the implementation and easily maintain the application.

Part IV has the following chapter:

- Conclusion and Future Work – describes the conclusions and future work that can be done to improve the Trace Visualiser.

# **Part I – Theory and Early Trace Visualiser**

# 1. Theoretical Beddings of the Work

## 1.1. IVY

The IVY tool supports an approach based in the utilization of models and their verification with model checking. The models are developed using the MAL Interactors language [Campos01, Campos04]. For a better understanding of what will be presented its necessary to have in mind the notion of *interactor*. An *interactor* can be seen as an object capable of presenting (part of) its state through some presentation medium. An *interactor* is defined by:

- a set of typed attributes that define its state;
- a set of actions that can change the *interactor's* state;
- a set of axioms written in MAL (Model-Action Logic) [Ryan91] that defines the semantics of the actions in terms of its effect in the *interactor's* state.

A model is constructed composing *interactors* hierarchically. This way, a model can always be represented by a state machine in which the states are defined by the values of its attributes and the transitions labelled by the actions that cause changes in the attributes. The stage of verification consists in testing properties of this state machine's behaviour.

The properties are written in CTL (*Computational Tree Logic*) [Clarke86] (see section 2.2). For details of the application of this logic in the present context see [Campos01]. To the discussion that follows the important point is that when one given property does not hold, the SMV tries to supply a trace behaviour that demonstrates the falseness of the property in question (see figure 2 for an example). That trace represents one sequence of states of the machine that violates the property.

To allow for their verification, the MAL interactor models are compiled to the SMV language. Because the expressivity of SMV language is limited, when compared with the MAL language of *interactors*, the process of compilation to SMV introduces a series of auxiliary variables. It deserves particular mention the introduction of the attribute 'action', used to model the actions because the SMV does not use that concept.

To the discussion that will follow, another important aspect has to do with the execution models. At the level of MAL *interactors* the actions of the different interactors can happen in an asynchronous way. So, one *interactor* can execute one action while the others remain inactive. At the level of SMV, however, the state transitions occur in a synchronous way. To model

asynchronous state transitions, it is necessary to introduce a special action *nil* that at the level of MAL interactors (logic level) corresponds to nothing happening, but at the level of SMV (physical level) represents a state transition (to a state with the same attribute values). In this way, the SMV module corresponding to an interactor can suffer one state transition associated with a given action, while the others execute the transition associated to *nil* (that is, they stay in the same logical state).

The traces produced by the verification process do, as we can expect, make reference to the variables and states existing at the level of SMV code. Thus, it is necessary to revert the process so that the entities referred to by the visualiser are the ones that exist at the level of abstraction of the original model.

These traces, however, can achieve sizes in the order of tens or hundreds of states, depending on the model's complexity. The visualiser component of the IVY tool, through visual representations and trace analysis mechanisms, will facilitate its comprehension as well as give a better idea of its relation with the model, in order to more clearly show which problem is pointed out by the trace, and possible solutions to it.

## 1.2. CTL (Computational Tree Logic)

The following description of CTL was taken from [Campos01].

CTL is used to express properties of the behaviour of the system specified in SMV. A formal description of CTL is given by [Clarke99]. An informal account of the operators is given here. Beside the usual proposition logic connectives CTL allows for operators over the computation paths that emanate from a state:

- A - for all paths (universal quantifier over paths);
- E - for some paths (existential quantifier over paths).

and over states in a computation state:

- G - used to specify that a property holds at all the states in the path (universal quantifier over states in a path);
- F - used to specify that a property holds at some state in the path (existential quantifier over states in a path);
- X - used to specify that a property holds at the next state in the path;
- U - used to specify that a property holds at all states in the path prior to a state where a second property holds.

These operators allow us to express concepts such as:

- universally:  $AG(p)$  -  $p$  is universal (for all paths, in all states,  $p$  holds);



- inevitability:  $AF(p) - p$  is inevitable (for all paths, for some state along the path,  $p$  holds);
- possibility:  $EF(p) - p$  is possible (for some path, for some state along that path,  $p$  holds).

### 1.3. Analysis of MCP example

In this section we introduce an example that will be used throughout this report.

The MCP (Mode Control Panel) of an aircraft is one element of the interface between the pilot and the aircraft autopilot.

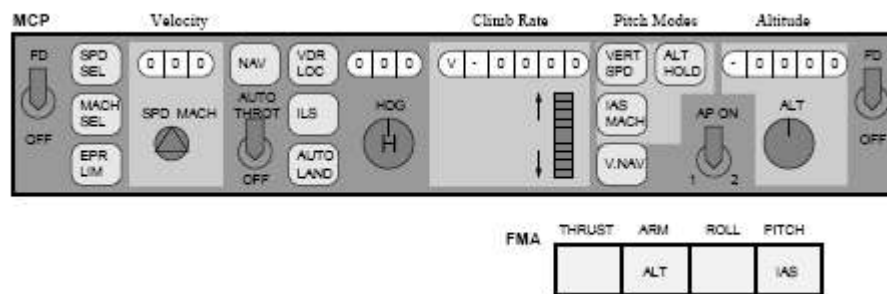


Figure 3 – The MCP (adapted from Honeywell Inc., 1988).

In [Palmer95] a case study is put forward which deals with a problem relating to altitude acquisition in a real aircraft, the MD-88. In the present case the issue is how automation and user interact during altitude acquisition. A reasonable expectation for the pilot to have of the system is that:

*Whenever the pilot sets the automation to climb up to a given altitude, the aircraft will climb until such altitude is acquired and then maintain it.*

The design of the MCP panel interface has been based on the plausible assumption that if the altitude capture (ALT) is armed the aircraft will stop at the desired altitude (selected in ALTDial). This can be expressed as the CTL formula:

$$AG((plane.altitude < ALTDial.needle \ \& \ ALT) \ \rightarrow \ AF(pitchMode=ALT\_HLD \ \& \ plane.altitude=ALTDial.needle))$$

which reads: it always happens (AG) that if the plane is below the altitude set on the MCP and the altitude capture is on then eventually (AF) the altitude will always be reached and the pitch mode be changed to altitude hold.

A model of the MCP was built using [Palmer95], and the manual of the aircraft as reference. When the SMV model checker is used to check a specification, the checker answers whether or not the test succeeds. When we check the model that was developed against the formula above a trace is returned as counterexample (see figure 4 for an extract of the trace).

```
-> State: 1.4 <-  
  action = enterAC  
  ALTDial.action = nil  
  plane.action = fly  
  plane.climbRate = 1  
  plane.airSpeed = 5  
  plane.altitude = 1  
  oblenterAC = 0  
  pitchMode = ALT_CAP  
  ALT = 0  
-> State: 1.5 <-  
  action = enterIAS  
  plane.action = nil  
  asDial.action = set_5  
  asDial.needle = 5  
  pitchMode = IAS  
-> State: 1.6 <-  
  action = toggleALT  
  ALTDial.action = set_1  
  ALTDial.needle = 1  
  plane.action = fly  
  plane.climbRate = 0  
  asDial.action = nil  
  oblenterAC = 1  
  ALT = 1
```

Figure 4 – Extract of trace counterexample.

Now, the Trace Visualiser can be used to analyse the generated trace, using trace visual representations and analysis mechanisms. In the following chapters, the manner in which we can perform that analysis will be described in detail.

The full example can be seen in [Campos01] (without trace visualiser analysis).

## 2. Early Trace Visualiser

In this section a description of the architecture of an early Trace Visualiser is given (see [Sousa06] for its full description) and the visual representations that it had are described. These representations are the following: Diagram representation, Tabular representation and Tree representation. The last section makes the bridge between the early and the final Trace Visualiser.

### 2.1. Architecture

To more easily explain the individual components, their class diagrams and a description of their responsibilities are given in the following sections. The architecture of the early component is provided in the package view in figure 5. Now each package will be described in detail.

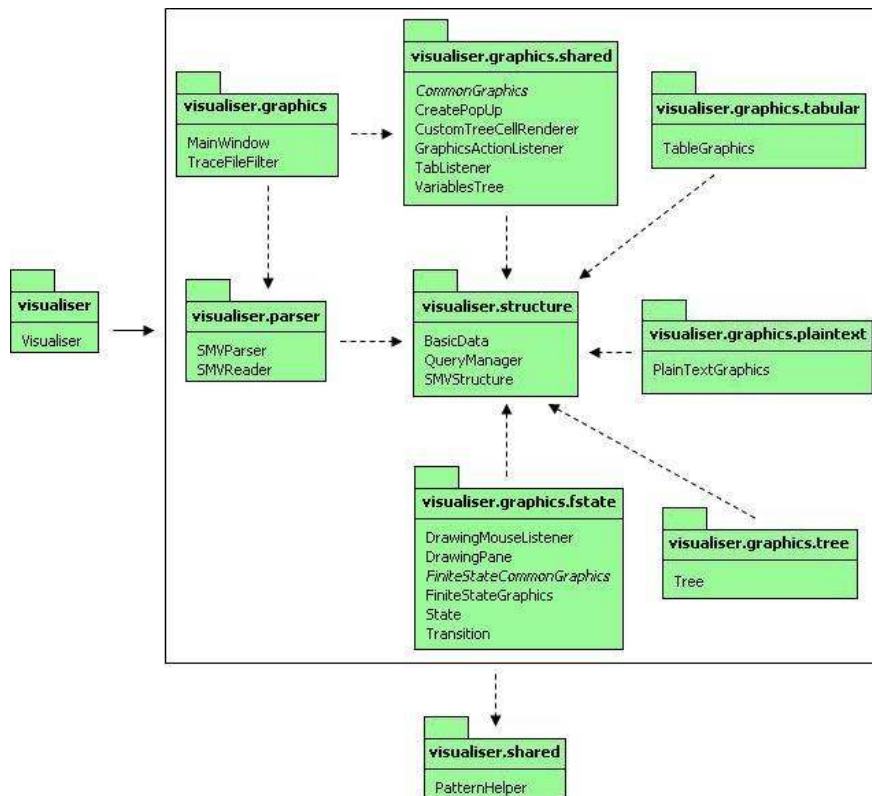


Figure 5 – Package View.

## 2.1.1 Parser

**SMVReader** reads the trace file and then calls **SMVParser** to create the **SMVStructure**. **SMVParser** is responsible for parsing the trace file and calls the methods of the **SMVStructure** to create a data set.

## 2.1.2 Structure

**SMVStructure** is responsible for the storage and manipulation of a trace's parsed data. It performs queries on datasets using a **QueryManager**.

**QueryManager** performs queries on a dataset. The following queries are supported:

- `getActionsState()` – returns all the interactor's actions in a given state.
- `getState()` – returns all the interactor's attributes on a given state.
- `getValues(); --` returns all the values for given interactor's attributes.
- `getChangeStatesValue()` – returns all states where a set of interactor's attributes changed to given values<sup>1</sup> (the final set of states result is the union of the sets of states returned by each individual interactor attribute).
- `getAllChangedStatesValue() --` returns all states where a set of interactor's attributes changed to given values (the final set of states result is the intersection of the sets of states returned by each individual interactor attribute).
- `getStatesValue() --` returns all states where a set of interactor's attributes are equal to given values (the final set of states result is the union of the sets of states returned by each individual interactor attribute).
- `getAllStatesValue() --` returns all states where a set of interactor's attributes are equal to given values (the final set of states result is the intersection of the sets of states returned by each individual interactor attribute).

The queries use the functionality provided by the **BasicData** class to access the data which is stored in a hashtable. Since such basic methods are provided, adding new queries should be easy.

**BasicData** is the most low-level part of the program. It stores the parsed data and allows for simple manipulation of the data. The data store consists of the names of the variables and the values they have. If a variable does not have a value in a state (in the trace), the value of the previous state is used. Furthermore a list of state IDs is kept. These are needed to identify

---

<sup>1</sup> By "changes to a given value" we mean that the value of the attribute in the previous state must be different from the value we are looking for. Consider a sequence of states [`<a=1,b=1>`,`<a=2,b=2>`,`<a=2,b=1>`], if we were looking for state where attribute 'a' changes to 2, then the second state would be in the result but not the third.

to which states the values in the table belong, as states can be collapsed (if two consecutive states have the same values, the second state can be removed).

### 2.1.3 Graphics

**Visualiser** contains the main routine which starts the **MainWindow**.

**MainWindow** is responsible for creating the actual visualiser and starting all the representations. It allows the user to load trace files, switch between them and to end the program.

When a file is opened, the **TraceFileFilter** class ensures that instead of all types of files, only files which end on .trace are shown.

All graphical representations have to extend the abstract **CommonGraphics** class. By casting these representations to **CommonGraphics** objects it is still possible to use the API defined by the **CommonGraphics** class. At each instant several graphical representations can be present at the interface. By keeping track of the currently activated (static) **CommonGraphics** object, filtered data can be given to the correct graphics class. For example, if the user is working with a Tabular Representation, the **currentGraphic** object is set to that representation, ensuring that when a filter is applied, it is applied to that object.

In order to correctly set the **commonGraphics** object, the abstract method **setCurrentGraphic()** must be implemented (see [Ridder05]).

The **CommonGraphics** class is also responsible for the internal frame ensuring that new graphical representations can be added and removed correctly. The class diagram is shown in Figure 6.

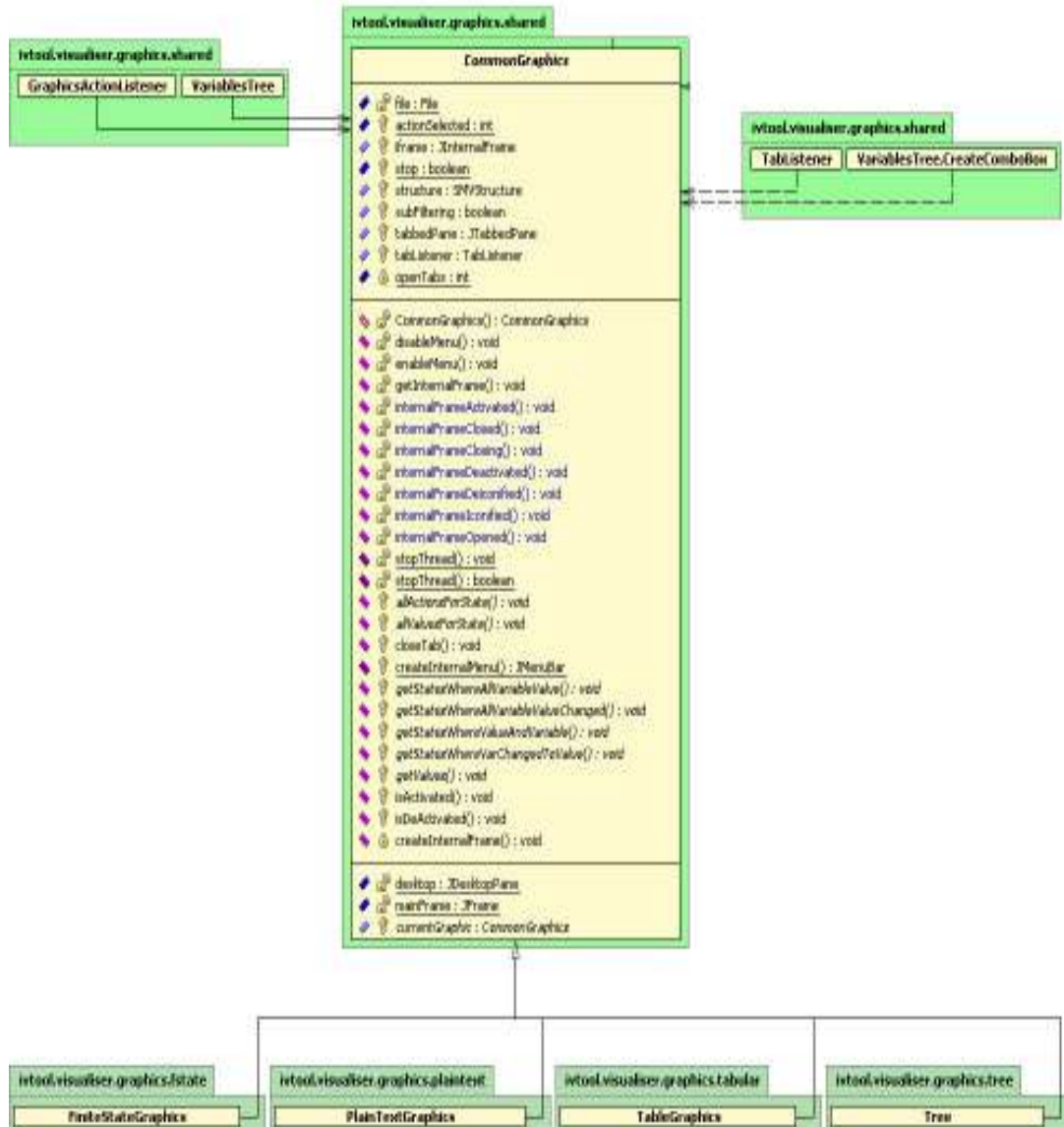


Figure 6 – CommonGraphics Diagram.

**CreatePopUp** is a class which creates simple popups for the **GraphicsActionListener** class.

**GraphicsActionListener** is used by **CommonGraphics**. It is used to determine which action to take when a filter is selected from the menu.

**TabListener** does the listening of the tabbed pane, ensuring that tabs are added and deleted correctly. It also ensures that the currentGraphic object (in **CommonGraphics**) is set properly.

**VariablesTree** is used for the selection of variables for certain filters. The sub-class **CreateComboBox** allows for dynamic creation of combo boxes that are used to interact with the user when the user selects the parameters for the filters.

**FiniteStateGraphics** is responsible for diagram representation and to do that extends the **CommonGraphics** class and implements the abstract filter routines. It also creates the **State** and **Transition** objects and calculates their placement on the screen.

**PlainTextGraphics** shows the original trace. It extends the **CommonGraphics** class, but the abstract methods concerning the filtering are kept empty.

**TableGraphics** creates the tabular representation of the trace file extending **CommonGraphics** and implementing the filters. It creates and colors the table and cells.

**Tree** creates the tree representation. It extends the **CommonGraphics** class and implements the abstract methods to allow for filtering.

**FiniteStateCommonGraphics** contains what the **State** and **Transition** class share, such as coloring and storage of displayable data for labels and mouseover events. It has the abstract method “hit”, used to define when a mouse is inside the graphical object. All classes extending the **FiniteStateCommonGraphics** must implement this method. The class diagram is shown in Figure 7.

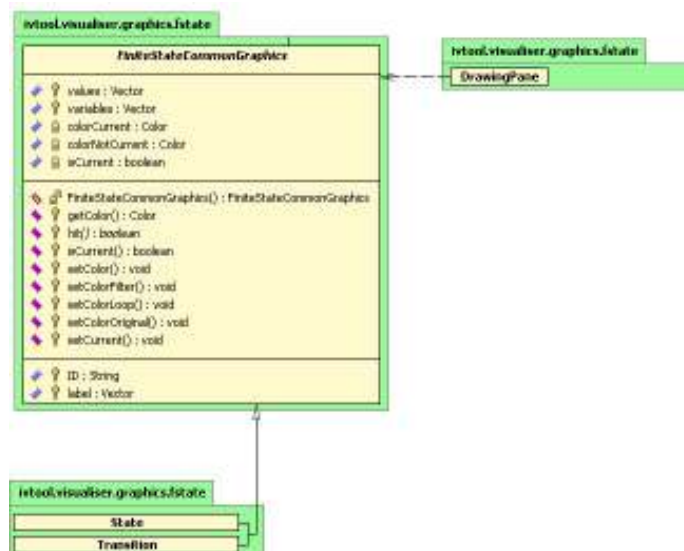


Figure 7 – FiniteStateCommonGraphics Diagram.

**State** has methods which are typical for the state object, mainly concerning the coordinates of the rectangle that represents the state in the graphics. Since it extends the **FiniteStateCommonGraphics**, it must implement the abstract “hit” method defined there.

**Transition** has methods which are typical for the transition object, mainly concerning the coordinates of the arrow that represents the transition in the graphics. Since it extends the **FiniteStateCommonGraphics**, it implements the abstract “hit” method defined there.

**DrawingPane** does the actual drawing of the diagram. It defines how arrows and rectangles are drawn and draws them in the proper color. Also is responsible for the animation of the trace when using play, step forward, step backward, etc, buttons.

**DrawingMouseListener** is responsible for the mouseover showing the data. Currently the mouseover only shows data for the states, since the data for the transitions is always shown.

## 2.1.4 Shared

**PatternHelper** contains pattern methods which are used by classes of other packages. Essentially these are methods for working with strings.

## 2.2. Graphical Interface

The early Trace Visualiser had a JDesktopPane with internal frames. The internal frames are created when a trace file is opened (one internal frame by trace file). In an internal frame it is possible to see four visual representations (clicking in the tabbedPane) and a filters menu with all the filters available.

The diagram representation has buttons to play, pause, go forward, go back and reset its animation. There exists also a clean button to clean filtering operations. The Tabular and Tree representations only have the clean button, because the animation is not implemented on them. The Trace visual representation does not have any such button, which means that animation and filtering is not possible there.

Figure 8 shows the GUI of the Early Trace Visualiser.

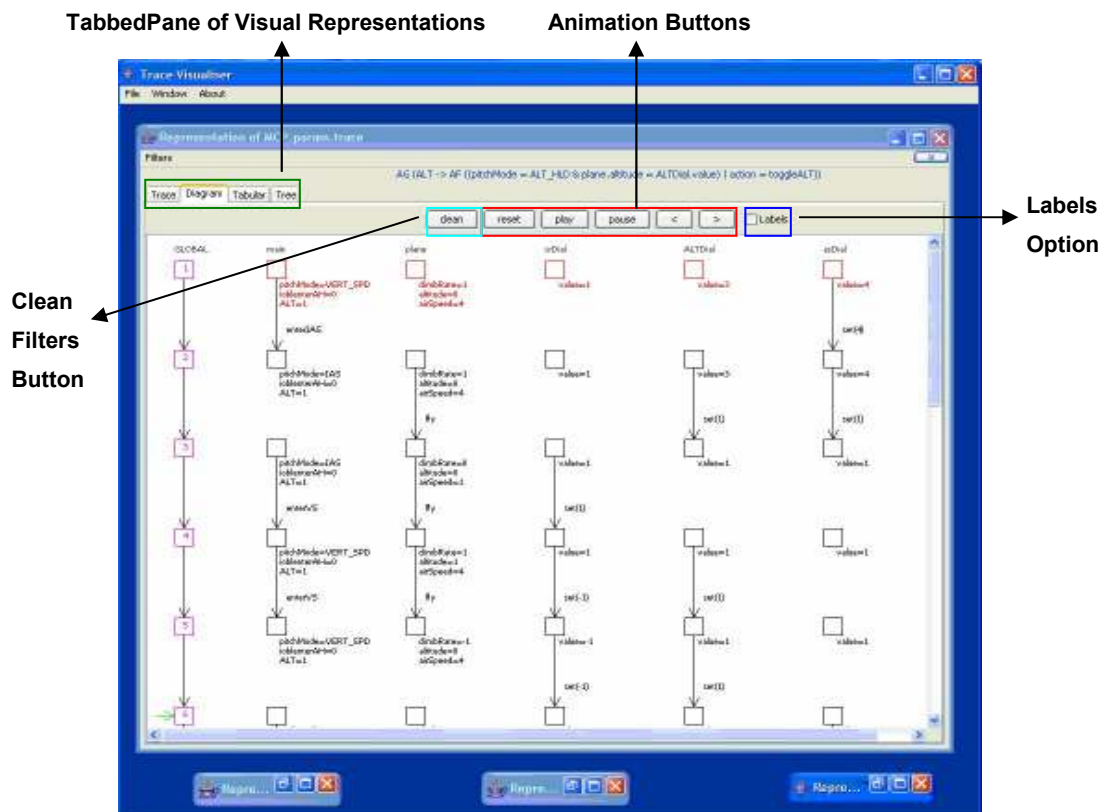


Figure 8 – GUI of Early Trace Visualiser.



## 2.3. Diagram Representation

In this representation (see figure 9), for each interactor there is a column with a state diagram-like representation showing the respective variables and their values, and also their actions. The global state (GLOBAL in figure 9) with all the variables of all interactors is also represented to act as an index. In the case of interactors, the variables and their values are shown near the rectangle. The actions are shown as labels on the arrows between consecutive states. The arrows are shown only if there exists an action between states. When no arrow is shown, the nil action is performed (see section 1.1, for a discussion on physical states). In the case of loops two more arrows are created representing the beginning and the end of the loop.

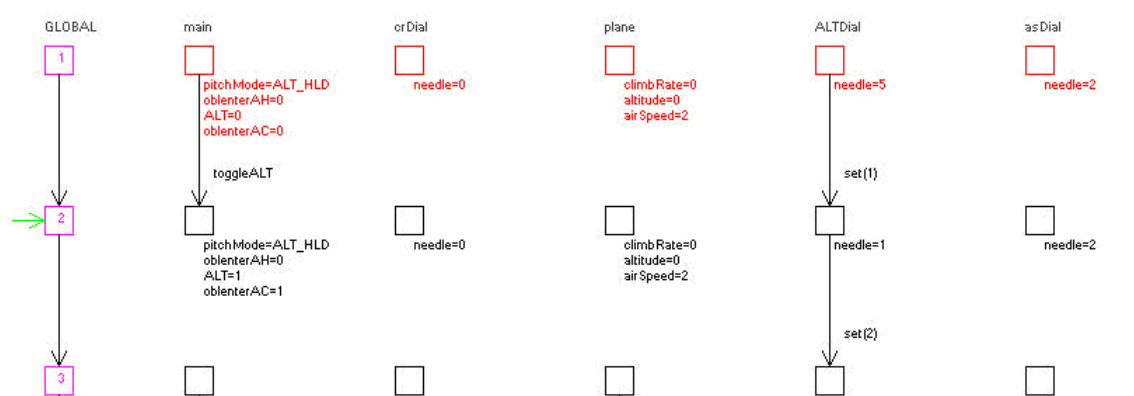


Figure 9 – Diagram representation.

If the Trace Visualiser user selects the labels option, the diagram is reduced and only the rectangles of the states and the arrows with actions are shown. The interactors and Global state variables and their values are shown as popup labels when the mouse passes over the states. In this way, the user can see more states in the screen and choose to see the variables and respective values of a specific interactor in some state. In this option of representation the arrows of the loop were not correctly shown because their coordinates are wrongly transformed when switching between boxes and labels. Figure 10 presents the diagram representation with the labels option enabled.

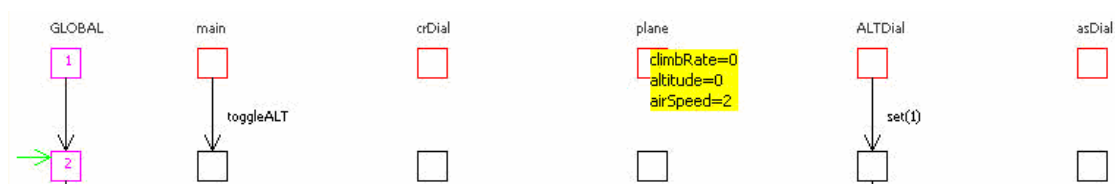


Figure 10 – Diagram representation with labels.

When using filters some of them return states as result and others return variables from interactors. In the first case the lines corresponding to states belonging to the result are highlighted in yellow (see figure 11).

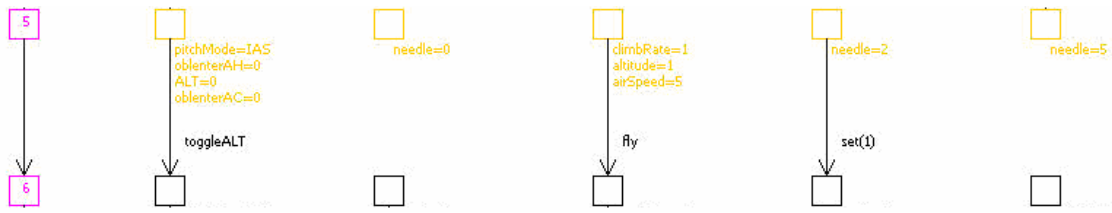


Figure 11 – Filter “Get values of state 5”.

In the second case only the columns of interactors with variables in the result are highlighted.

Having the labels option disabled is the default when showing the initial diagram representation. This option shows the information of variables near the rectangles representing states. If the labels option is enabled information on variables is hidden and a listener is activated. It listens for mouse over state events. In the case of a hit (mouse coordinates inside the rectangle of state) a popup label with the information of that state is shown.

The animation of the diagram representation is done by highlighting the color of states when passing by them, and changing the position of the scrollbar of the pane that includes the diagram to keep the highlighted states visible at all times.

## 2.4. Tabular Representation

In a tabular representation data is shown in a table (see figure 12). The columns headers show the state numbers. The beginning of a loop is shown adding a \* to the respective state number. A cell in white shows that the value of that variable in that state changed when compared to the previous state. A cell in grey shows that the value of variable remained the same when compared to the previous state.

	1	2*	3	4	5	6	7	8	9	10	11	12	13
ALTDial.action		set(1)	set(2)			set(1)		set(1)	set(2)				set(1)
ALTDial.needle	5	1	2	2	2	1	1	1	2	2	2	2	1
asDial.action						set(5)				set(2)			
asDial.needle	2	2	2	2	5	5	5	5	5	2	2	2	2
crDial.action							set(-1)	set(-1)	set(0)				
crDial.needle	0	0	0	0	0	0	-1	-1	0	0	0	0	0
main.ALT	0	1	1	0	0	1	1	1	1	1	1	0	1
main.action		toggleALT		enterAC	enterIAS	toggleALT	enterVS	enterVS	enterVS	enterIAS	enterAH	toggleALT	toggleALT
main.oblenterAC	0	1	1	0	0	1	1	1	1	1	1	1	1
main.oblenterAH	0	0	0	0	0	0	0	0	0	0	0	0	0
main.pitchMode	ALT_HLD	ALT_HLD	ALT_HLD	ALT_CAP	IAS	IAS	VERT_SPD	VERT_SPD	VERT_SPD	IAS	ALT_HLD	ALT_HLD	ALT_HLD
plane.action				fly		fly	fly	fly	fly	fly		fly	
plane.airSpeed	2	2	2	5	5	5	5	5	5	2	2	2	2
plane.altitude	0	0	0	1	1	1	0	0	0	0	0	0	0
plane.climbRate	0	0	0	1	1	0	-1	-1	0	0	0	0	0

Figure 12 – Tabular representation.

In Figure 13 the result of a filter that returns the states where two variables have the values selected by the user, with the syntax “OR”, is shown. States which have at least one variable with the value selected are highlighted, for example state 10.

	1	2*	3	4	5	6	7	8	9	10	11	12	13
ALTDial.action		set(1)	set(2)			set(1)		set(1)	set(2)				set(1)
ALTDial.needle	5	1	2	2	2	1	1	1	2	2	2	2	1
asDial.action					set(5)					set(2)			
asDial.needle	2	2	2	2	5	5	5	5	5	2	2	2	2
crDial.action							set(-1)	set(-1)	set(0)				
crDial.needle	0	0	0	0	0	0	-1	-1	0	0	0	0	0
main.ALT	0	1	1	0	0	1	1	1	1	1	1	0	1
main.action		toggleALT		enterAC	enterIAS	toggleALT	enterVS	enterVS	enterVS	enterIAS	enterAH	toggleALT	toggleALT
main.oblenterAC	0	1	1	0	0	1	1	1	1	1	1	1	1
main.oblenterAH	0	0	0	0	0	0	0	0	0	0	0	0	0
main.pitchMode	ALT_HLD	ALT_HLD	ALT_HLD	ALT_CAP	IAS	IAS	VERT_SPD	VERT_SPD	VERT_SPD	IAS	ALT_HLD	ALT_HLD	ALT_HLD
plane.action				fly		fly	fly			fly		fly	
plane.airSpeed	2	2	2	5	5	5	5	5	5	2	2	2	2
plane.altitude	0	0	0	1	1	1	0	0	0	0	0	0	0
plane.climbRate	0	0	0	1	1	0	-1	-1	0	0	0	0	0

Figure 13 – Filter “States where main.pitchMode=IAS Or plane.altitude=1”.

## 2.5. Tree Representation

In a tree representation data is shown in several trees, one for each state. The interactors' variables are shown in red and their actions in blue. Initially all the trees are expanded but the user can collapse states or *interactors* if he desires. Using the clean button, on the top of the pane, all the trees are again expanded. This representation is shown in Figure 14.

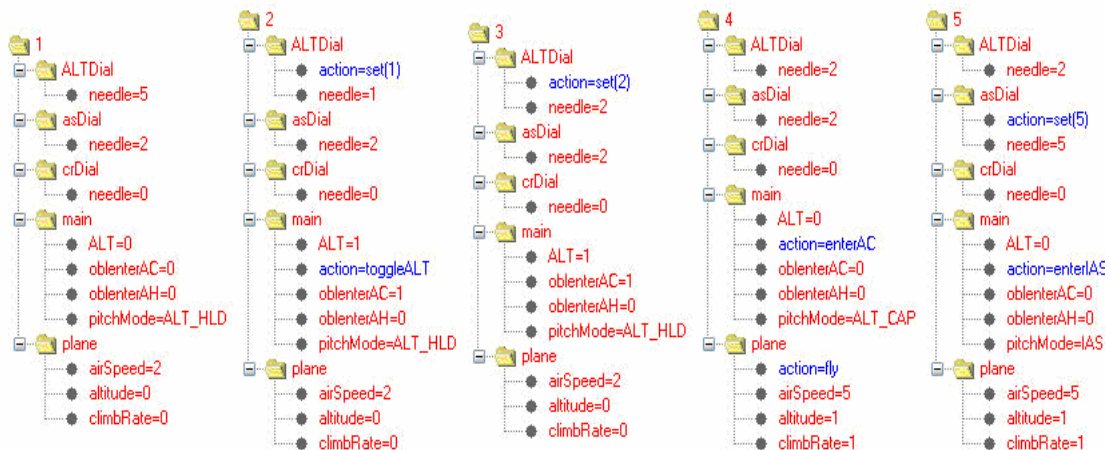


Figure 14 – Tree representation.

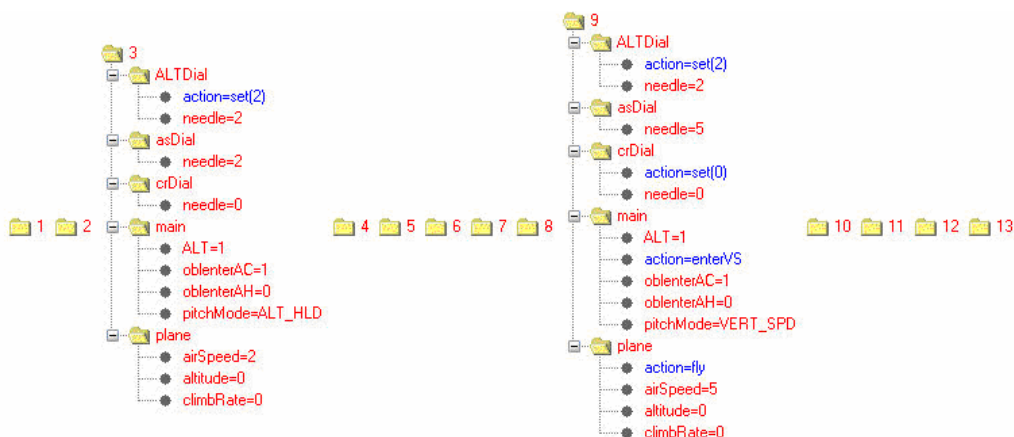


Figure 15 – Filter “States where ALTDial.needle changed to 2”.

In the case of filters that return *interactor's* variables as result, those variables are highlighted with magenta color, in all the states.

In the implementation of filtering, the filters that return states were implemented with the idea of expanding the states in the filter's results and collapsing the other ones. In figure 15, we can see an example of that. To expand or collapse all nodes of the JTrees, the code from [Chan02] was used.

## 2.6. User Input in Filters

When using filters, the input from the user is read with a **VariablesTree** class that contains all the variables of the interactors. A variables wizard with two consecutive panels is used. The first panel shows the tree with all the variables (see figure 16) and the user can select which variables will be used in the filtering criteria. The second panel shows the possible values on the selected variables (see figure 16). Then, the finish button gives the results of the filter.

In some cases the input is done asking only for a state number (see figure 17).

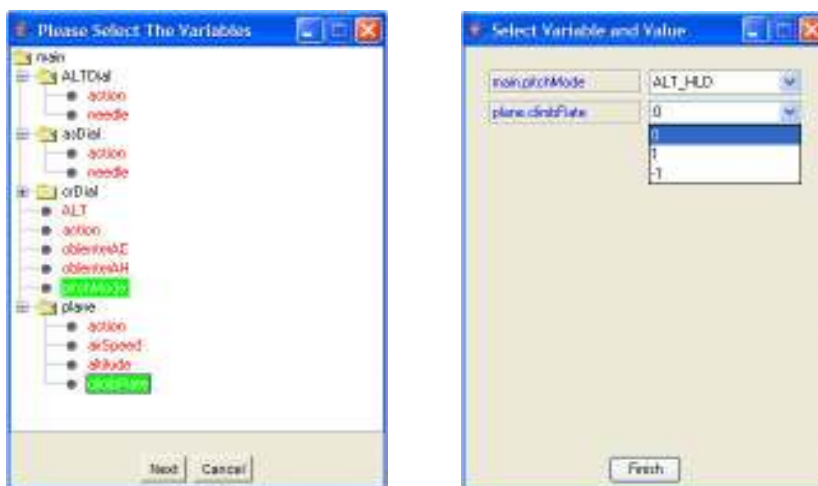


Figure 16 – First and second panels.

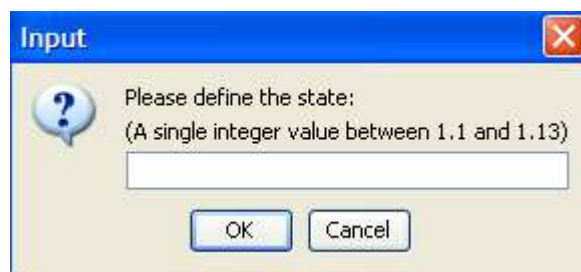


Figure 17 – Input with state number.

## 2.7. Bridge between early and final Trace Visualiser

We will now highlight the main changes introduced in the final Trace Visualiser.

### 2.7.1 Implications in the Code

The size of the code from the early Trace Visualiser was reduced both in the number of lines needed, and in its complexity. The classes that implement the visual representations were made independent of the structural code by, for example, eliminating shared variables. This way, it is possible to integrate them in the rest of the structure, without having the need to understand the rest of code. For example, we now have a `JFrame` with a `JSplitPane`, which hold the visual representation and trace analysis mechanisms areas. The code used for doing that construction and management is independent from the code used for any visual representation. A visual representation is done implementing a set of methods which represent its functionality in terms of animation, popups, markers and filters. The impact, on the architecture of the Trace Visualiser, of adding a new visual representation is minimal (adding of a total of thirteen lines of code, which are of simple understanding, in two classes).

The code from the classes was studied to eliminate its unnecessary complexity and it was possible to reduce, significantly, the number of lines needed. Also, the code is now more documented with useful comments.

The structure of packages was reformulated to better structure the classes and their responsibilities.

### 2.7.2 Implications in the Graphical Interface

In the graphical aspect, all the buttons used were put on two toolbars and are now independent of each visual representation. The first toolbar holds a `comboBox` for switching between visual representations and a `textArea` to hold the formula of the trace. The second toolbar holds buttons for animation and closing of tabs, a `checkbox` for the popups functionality and a `textArea` for filter description. The `contentPane` of the mainframe now holds a `SplitPane` only, which is divided in two areas. The left area is for showing the current visual representation and the right area is for adding markers (trace analysis mechanisms similar to filters but more powerful) and filters. To all buttons, icons were added in order to provide a more pleasant application appearance.

Regarding menus, a new menu was added to enable users to choose the LookAndFeel of the application. Also two new menu options on the File menu were added to close a trace and to export a visual representation to an image file.

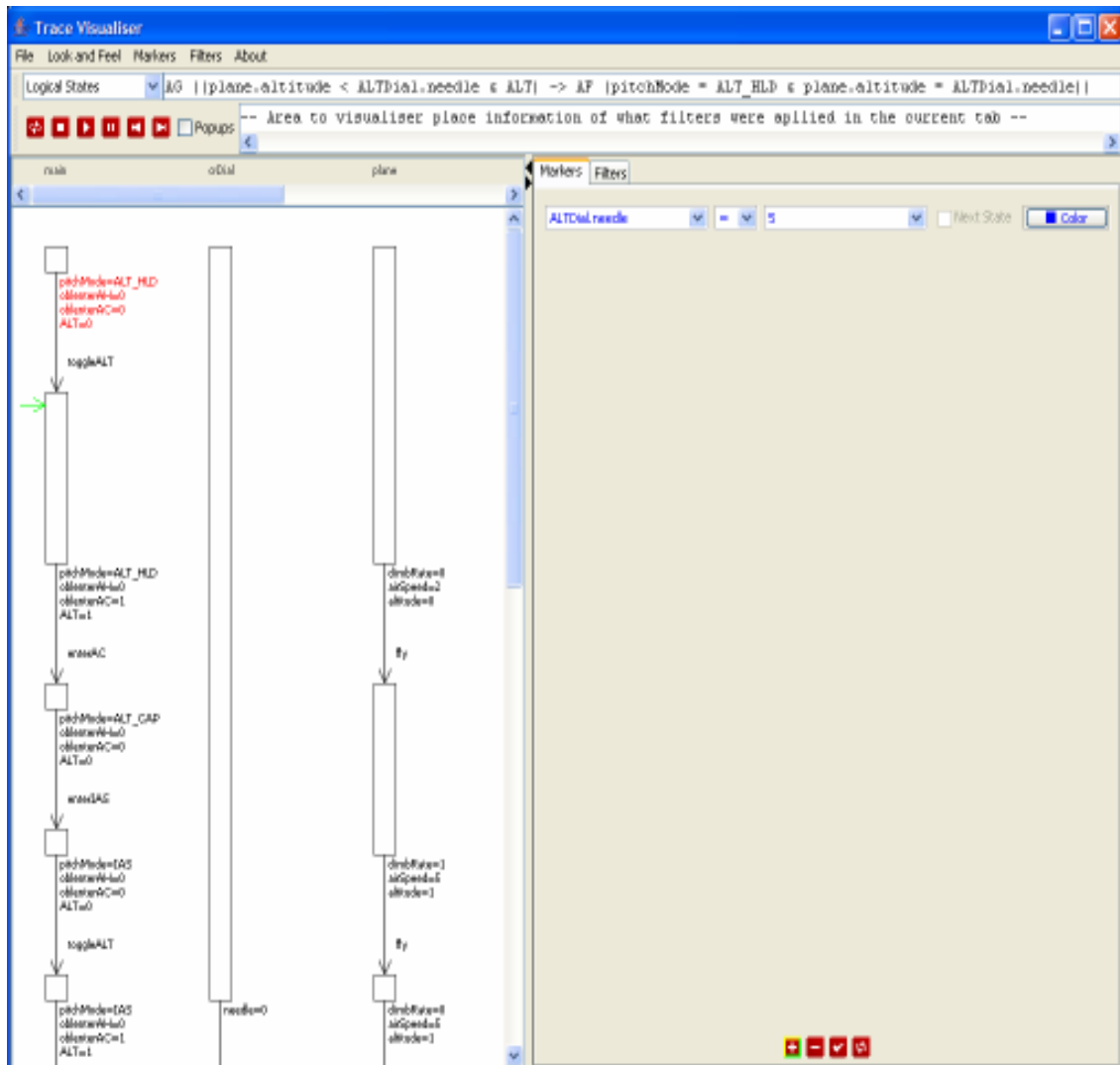


Figure 18 – GUI of Final Trace Visualiser

### 2.7.3 Changes in the Visual Representations

Two new visual representations were created: Logical States and Activity Diagram. The first one is similar to Diagram representation (in current Trace Visualiser it is called Physical States) and the last one is based on UML 2.0 Activity diagrams.

In all the visual representations based on states (Physical States, Logical States and Activity Diagram), the interactor names were placed on a separate panel from the visual representation panel to enable interactor names to always be visible at the top. In the previous visualiser, if the representation is scrolled down they disappear.

In the Tabular representation, the cells are now drawn with 3D rectangles to make them more perceptible. The two colors used to show the change (or not) of state have changed to dark gray and dark yellow, to provide a bigger contrast.

In the new Tree representation, we have only one tree, in which the children are nodes with state information (interactor's attributes). It was changed in this way to provide a stronger bound between all the states.

## 2.7.4 New Functionalities

In the tree representation listeners, to provide new functionality on collapsing/expanding states or *interactors* (all at once), were added.

Also, it is now possible to annotate visual representations with markers (trace analysis mechanisms), which appear as coloured circles or semicircles in the visual representation area.

In all the visual representations (except the Trace representation) the animation was standardized.

# **Part II – User Manual of the Final Trace Visualiser**



# 1. Introduction

This is a user manual for the IVY's Trace Visualiser component.

When a property does not hold, the SMV model checker tool usually produces a trace showing behaviour that falsifies the property in question. A SMV trace consists of a sequence of states of the model. In terms of the analysis of interactive systems, this trace can be seen as a usage scenario that falsifies the property under consideration. These scenarios must then be analysed in terms of their cognitive plausibility, and impact on the design.

The Trace Visualiser component is responsible for presenting the results of the verification process to the IVY user in a manner that facilitates the understanding of meaning of the trace.

This user manual intends to be sufficient clear so that any kind of user (users with different qualifications) can understand how to use the Trace Visualiser. First the graphical interface of the tool will be explained in detail (menus, buttons, toolbars and panels). Next the visual representations of the visualiser will be fully described. Finally an example of utilization will be provided to explain how the Trace Visualiser can be used to discover the problems pointed out on a trace file.

## 2. Graphical Interface

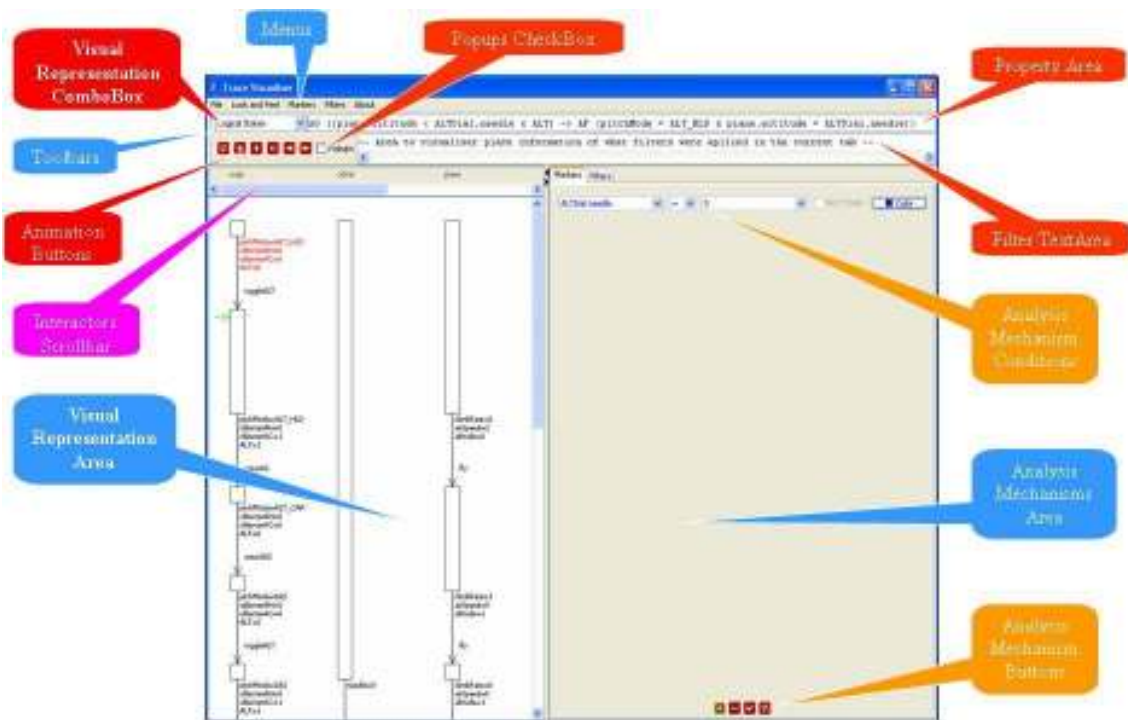


Figure 19 – Main frame.

The Trace Visualiser main frame has the following components:

- **Toolbars** → used to hold animation controls, text information areas and the comboBox for switching between visual representations;
  - *Visual Representation ComboBox* → this sub-component is used for switching between visual representations. All the visual representations that the Trace Visualiser has are shown and the user can select any one of them. If filters or markers were applied, the new visual representation applies them to;

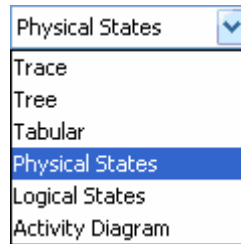








Figure 20 – Visual Representations ComboBox.

- *Formula Area* → is a JTextArea in which the formula, written in CTL, is displayed. This is the formula that was proved to be false because the trace file is only generated when a formula is false. This information helps to detect the problem because it is possible to add markers related to sub-conditions present on it.
- *Buttons for Animation* of visual representations:
  -  → stops the animation;
  -  → plays the animation;
  -  → pauses the animation;
  -  → goes one step backward in the animation;
  -  → goes one step forward in the animation;
  -  → cleans markers, filter and animation operations.
- *Popups checkbox* → is used to switch between two modes: popups (hides some information and shows it as popup labels), or no popups (all the information is shown directly on the representation). In some visual representations the checkbox is disabled because popups are always used or not used at all (this last situation happens only in the case of textual representations - Trace). The state of the checkbox is preserved when switching between representations.
- *Filter TextArea* → area to place information on what filters were applied. If more than one filter was applied to the visual representation, then the conjunction of all the filters is shown.
- **Menus** → used to provide operations on files (open, close and export), on the LookAndFeel, on filters and markers and to display information about the application. In all menu operations hot keys are provided to speed up access to the operations, and enable use of the application without the mouse (the hot keys are presented in the images that follow);

- *File* → used to open and close trace files and for export the visual representations to image files.

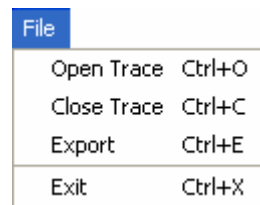


Figure 21 – File Menu.

- *View* → to choose the LookAndFeel of the main frame;



Figure 22 – View Menu.

- *Markers* → to use the marker operations. These operations are used to analyse the trace and can all be accessed through the buttons on the Markers tab on Analysis Mechanisms Area also, see figure 19;

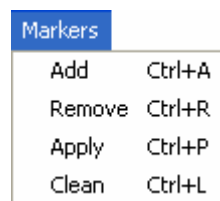


Figure 23 – Markers menu.

- *Filters* → to use the filter operations. All the filter queries to analyse the trace are available here (as with the markers these operations can all be accessed through the buttons on Filters tab on Analysis Mechanism Area also, see figure 19);

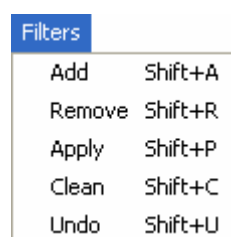


Figure 24 – Filters menu.

- *About* → to show information about the application.

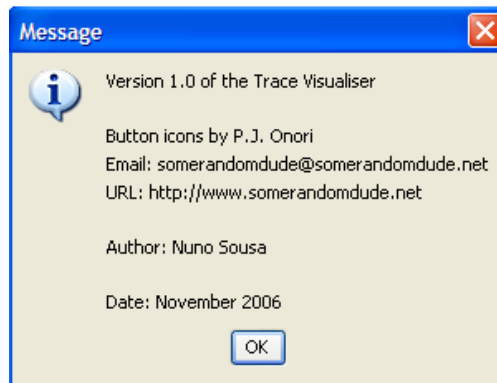


Figure 25 – About message.

- **Visual representation area** → is a panel that shows the current visual representation. It is the main area of the application because it is where the analysis is done and the visual representations shown. If the visual representation cannot be fully shown on screen, scrollbars (at the top and right of the panel), are used to allow scrolling the representation.
- **Analysis Mechanisms Area** → is a tabbedPane that holds two analysis mechanisms, each one in a different tab. They are: Markers and Filters.
  - **Markers** → is a panel that holds the conditions that will generate markers. The markers are used to mark states in relation to criteria defined over state attributes. The criteria are defined over states establishing relations (=, > and <) between attribute pairs or between values and attributes. A color is associated to each criteria, and all states that verify one given criteria are annotated with the colour associated to it. In the case of attributes comparison two filled semicircles are drawn, with the chosen color. Each semicircle is drawn near each of the attributes, this way the attributes are visibly related by the condition. In the case of comparison between values and attributes, filled circles with the chosen color are drawn. If the popups option is enabled it is possible to see the condition represented by each marker placing the mouse over it. At the bottom there are buttons to add, remove, apply and clean marker conditions to the visual representation.



Figure 26 – Markers conditions.

- *Markers conditions* → the first comboBox is used to choose the attribute of an interactor, the second comboBox is used to choose the comparison operator (=, < and >), the third comboBox is to choose a value or an attribute of an interactor (see figure 26). The checkbox Next State is used to choose if we want to compare attributes of interactors in two consecutive states (the first attribute on the first comboBox is associated with the current state and the second attribute on the third comboBox is associated with the next state). The Next State checkBox is enabled only if the parameter in third comboBox is an attribute and not a value. The Color button is used to choose the color to be associated with the condition and its respective marker. It makes a JColorChooser appear wich includes a recent colors panel to choose desired colors.

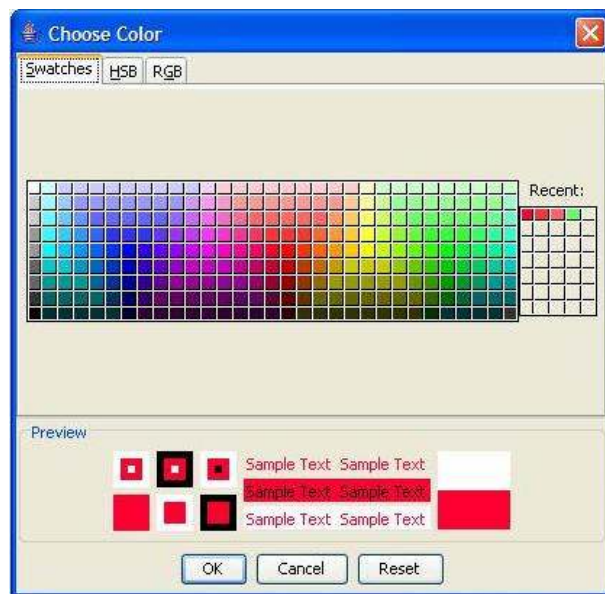









Figure 27 – Marker ColorChooser.

- *Markers buttons* :
  -  → to add a condition marker;

-  → to remove a condition marker;
  -  → to apply to the visual representation area the necessary changes; that is, annotate the visual representations with the markers generated from the conditions;
  -  → to clean all the conditions markers on the panel.
- *Marker types :*
-  → marker originated from a condition with syntax <attribute><op><value>;
  -  → markers originated from a condition with syntax <atribute1><op><atribute2>. Each semicircle will be, in the visual representation, near the attribute it is associated with;
  -  → markers originated from a condition with syntax <atribute1><op><atribute2>, but with the Next State checkbox ticked. This means that we will make a comparison between attributes in two consecutives states. Also placed near the attributes.
- **Filters** → is a panel that holds the conditions that will generate a filter result. The filters are used to highlight the states in relation to criteria defined over state attributes. The criteria are defined over states establishing relations (= and “changed to”) between attributes and values. A default color (orange) is used to highlight all the states in a filter’s result. The states of interactors that are in filter result and that have an attribute in the condition that generated the result, are highlighted with a different color (blue). The bottom of the panel holds the buttons to add, remove, apply, clean and undo filters applied to the visual representations.
- *Filter conditions* → the first comboBox is used to choose the attribute of an interactor, the second comboBox is used to choose the operator of comparison (= or ‘changed to’) and the third comboBox is used to choose a value or the attribute (see figure 31). The fourth comboBox is used to choose the type of relation of all the conditions, “AND’ or ‘OR’. If ‘AND’ is chosen then all the states have to respect all the input conditions. If ‘OR’ is chosen, it means that a state that respect at the least one of the set of conditions belongs to the filter’s result.

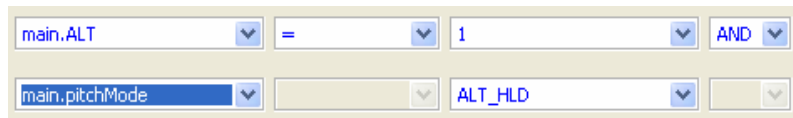







Figure 28 – Filter conditions.

- *Filter buttons* :
  -  → to add a filter condition;
  -  → to remove a filter condition;
  -  → to apply to the visual representation area the necessary changes; that is, highlight the states, returned by the filter;
  -  → to undo the last filter applied;
  -  → to clean all the filter operations on the panel.

## 2.1. Visual Representations

The images of the following visual representations were generated by the Trace Visualiser and exported to image files (with export option on File Menu) or captured directly from the screen. To all of them a filter and markers were applied.

The markers used can be seen in figure 26. They have the three usual comparison operators (=, < and >). The markers are more generic than the filters because filters only have comparisons of equality with the syntax <attribute>=<value> and the operator 'changed to' (used to return the states when the attribute of an interactor changed to a given value, comparing to the previous state).

The filter used can be seen in figure 28. It returns all states where main.ALT=1 and main.pitchMode=ALT\_HLD.

### 2.1.1 Tree

In a tree representation data is shown in several trees, one for each state. The interactors' variables are shown in red and their actions in blue. Initially all the trees are expanded but the user can collapse states or *interactors* if he desires.

In this representation the states that are in the filters result are expanded and the other collapsed. For example, the states 4 and 5 are not present in the filter's result, and because of that they are collapsed.

The markers are put near the attributes of an *interactor*.



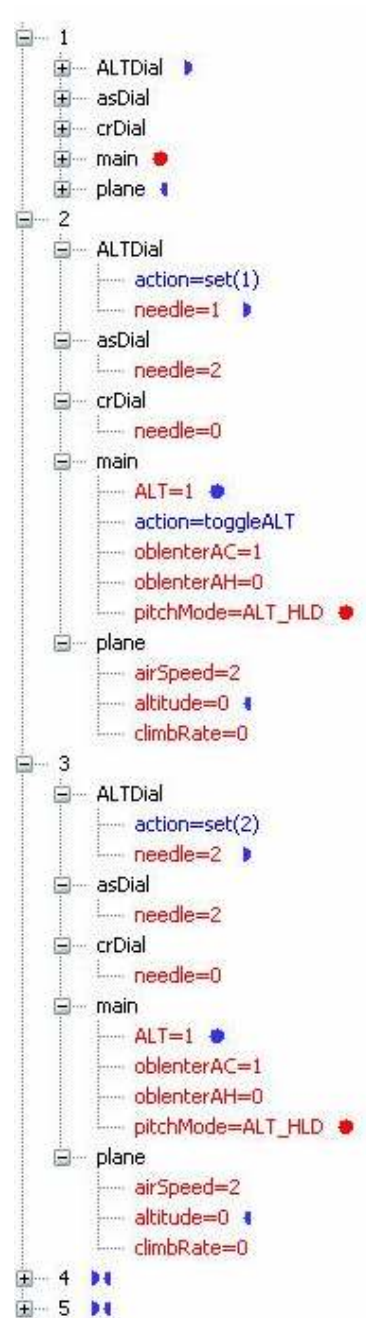


Figure 29 – Tree representation with filter and markers.

If, in a given state, the user right clicks with the mouse on an *interactor* node (for example ALTDial), then all *interactors* in that state are collapsed and the markers at the attributes level go up to the *interactors* level (see state 1 in figure 29). If the user right clicks again the nodes are expanded. The mouse right button fires collapse/expand events at the level of *interactor* in the whole state (collapses or expands all the *interactors* inside a state). This functionality is provided also for the states level (see figure 30).

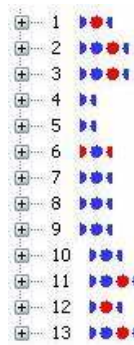


Figure 30 – Collapsing states.

To see what condition the marker represents it is possible to place the mouse over the marker circle or semicircle to make the desired information appear as a *tooltip* text. The popup label shows the information for the condition represented by the marker. Two semicircles in the visual representation mean that we are making a comparison between two attributes. In figure 31 the comparison is `plane.altitude<ALTDial.needle` and a semicircle exists in cell `needle=2` and other in `altitude=0`. If each marker condition uses a different color, the relation between markers is easily identified by the color of the markers.

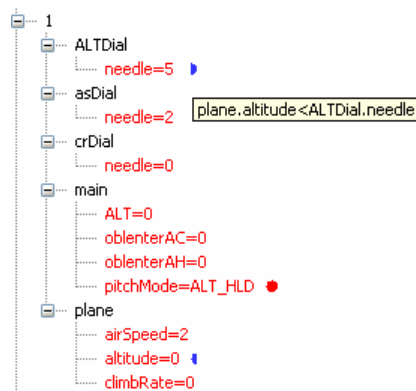


Figure 31 – Showing marker condition.

It is also possible to see an animation of this representation. It consists on collapsing all the states and expanding in sequence the states with a constant time. For example, collapse all the states, expand state 1, collapse state 1, expand state 2 and so on to the final state. If a loop is present, then when reaching to the final state the animation begin again from the initial point of the loop and enters on a cycle until the stop button is pressed.

## 2.1.2 Tabular

In a tabular representation data is shown in a table (see figure 32). The columns headers show the states numbers. The beginning of a loop is shown adding a \* to the respective state number.

In the states in which the value remains unchanged comparing to the previous state, the background color of the cell is dark gray. When an attribute value of an interactor changes, then the background color of the cell used is dark yellow. The cells are drawn as 3D Rectangles to make them more perceptible. The cells in yellow are the filter results. If an attribute has a marker, the marker is placed in the cell. To see marker conditions the same approach as for the Tree representation is used.

	1	2*	3	4	5	6	7
ALTDial.action		set(1)	set(2)			set(1)	
needle	5	1	2	2	2	1	1
asDial.action					set(5)		
needle	2	2	2	2	5	5	5
crDial.action							set(-1)
needle	0	0	0	0	0	0	-1
main.ALT	0	1	1	0	0	1	1
action		toggleALT		enterAC	enterIAS	toggleALT	enterV5
oblenterAC	0	1	1	0	0	1	1
oblenterAH	0	0	0	0	0	0	0
pitchMode	ALT_HLD	ALT_HLD	ALT_HLD	ALT_CAP	IAS	IAS	VERT_SPD
plane.action				fly		fly	fly
airSpeed	2	2	2	5	5	5	5
altitude	0	0	0	1	1	1	0
climbRate	0	0	0	1	1	0	-1

Figure 32 – Tabular representation with filter and markers.

The animation on this representation is done highlighting the current state column, using a color with inferior alpha component to make it look different from the others column states (see state 4 in figure 33). The loop behaviour is also reproduced here.

	2*	3	4	5	6
ALTDial.action	set(1)	set(2)			set(1)
needle	1	2	2	2	1
asDial.action				set(5)	
needle	2	2	2	5	5
crDial.action					
needle	0	0	0	0	0
main.ALT	1	1	0	0	1
action	toggleALT		enterAC	enterIAS	toggleALT
oblenterAC	1	1	0	0	1
oblenterAH	0	0	0	0	0
pitchMode	ALT_HLD	ALT_HLD	ALT_CAP	IAS	IAS
plane.action			fly		fly
airSpeed	2	2	5	5	5
altitude	0	0	1	1	1
climbRate	0	0	1	1	0

Figure 33 – Tabular animation.

### 2.1.3 Physical States

In this representation (see figure 34), for each interactor there is a column with a state diagram-like representation showing the respective variables and their values, and also their actions. In the case of interactors, the variables and their values are shown near the rectangle. The actions are shown as labels on the arrows between consecutive states. The arrows are

shown only if there exists an action between states. When no arrow is shown, the nil action is performed (see below for a discussion on physical states). In the case of loops two more arrows are created representing the beginning and the end of the loop.

This representation shows the states of *interactors* at physical level. At the level of MAL *interactors* (logical level) the actions of the different interactors can happen in an asynchronous way. So, one *interactor* can execute one action while the others remain inactive. At the level of SMV (physical level), however, the state transitions occur in a synchronous way. To model asynchronous state transitions, it is necessary to introduce a special action *nil* that at the level of MAL interactors corresponds to nothing happening, but at the level of SMV represents one state transition (to a state with the same attribute values). This way, the SMV module corresponding to an interactor can suffer one state transition associated with a given action, while the others execute the transition associated to *nil* (that is, stays in the same logical state).

The states highlighted in yellow color are present on the filter's result and therefore drawn in a different way. The first column Global is the global state (includes all values of *interactors* attributes in each state) and acts as an index.

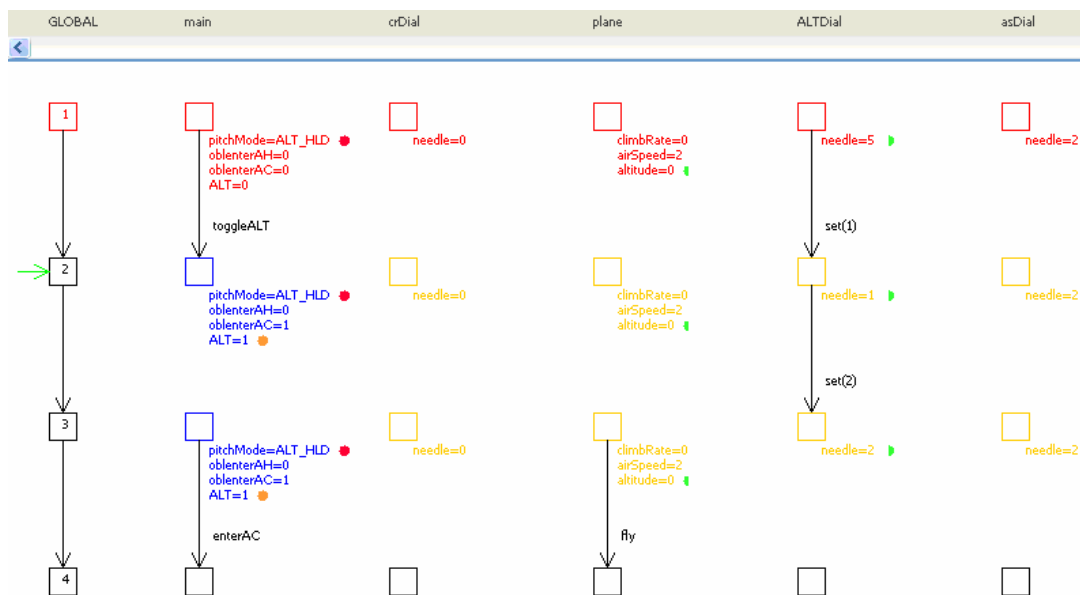


Figure 34 – Physical States representation with filter and markers.

The animation on this visual representation is done by sequentially placing a large rectangle, with a background color with a minor alpha component to be a little transparent, behind states or transitions.

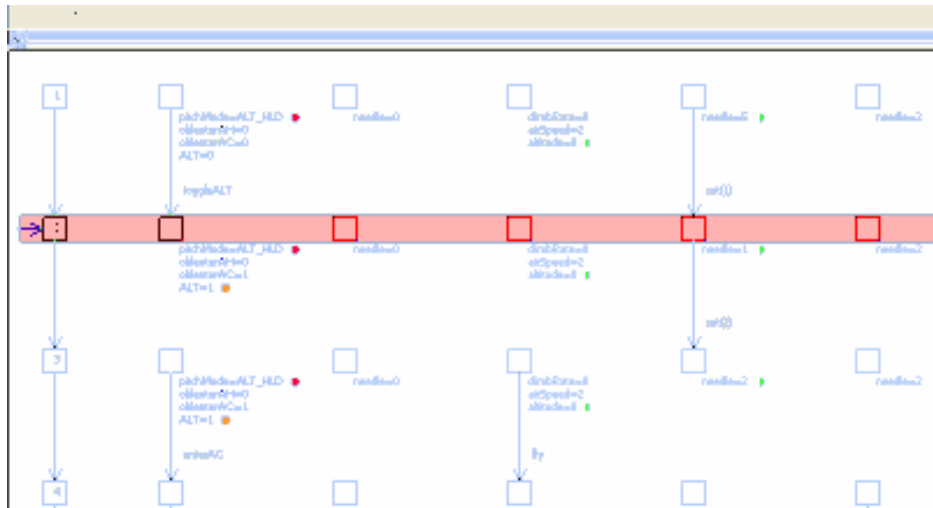


Figure 35 – Physical States animation.

This strategy of animation is used in all of the following visual representations except on the Activity representation.

With popups option selected it is possible to hide states information and see that information only if the mouse passes over the states or markers.

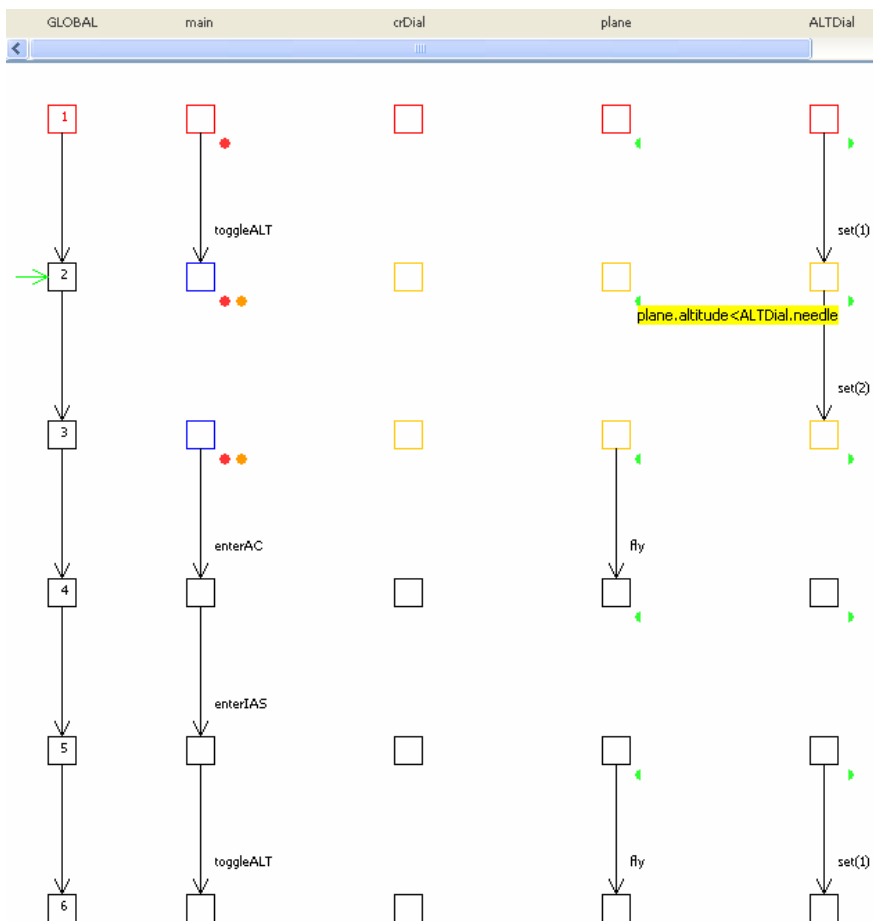


Figure 36 – Showing marker information with popups option.

## 2.1.4 Logical States

This representation is similar to the previous visual representation with the difference that it shows the logical states instead of physical states (see section 2.1.3 for an explication on logical and physical states). This means that sequences of states without transitions between them disappear. Instead of that, a singular state that covers all the physical states that happen at SMV level is presented.

The states highlighted in yellow and blue are present on the filter's result. The markers are represented by the lines, circles and semicircles.

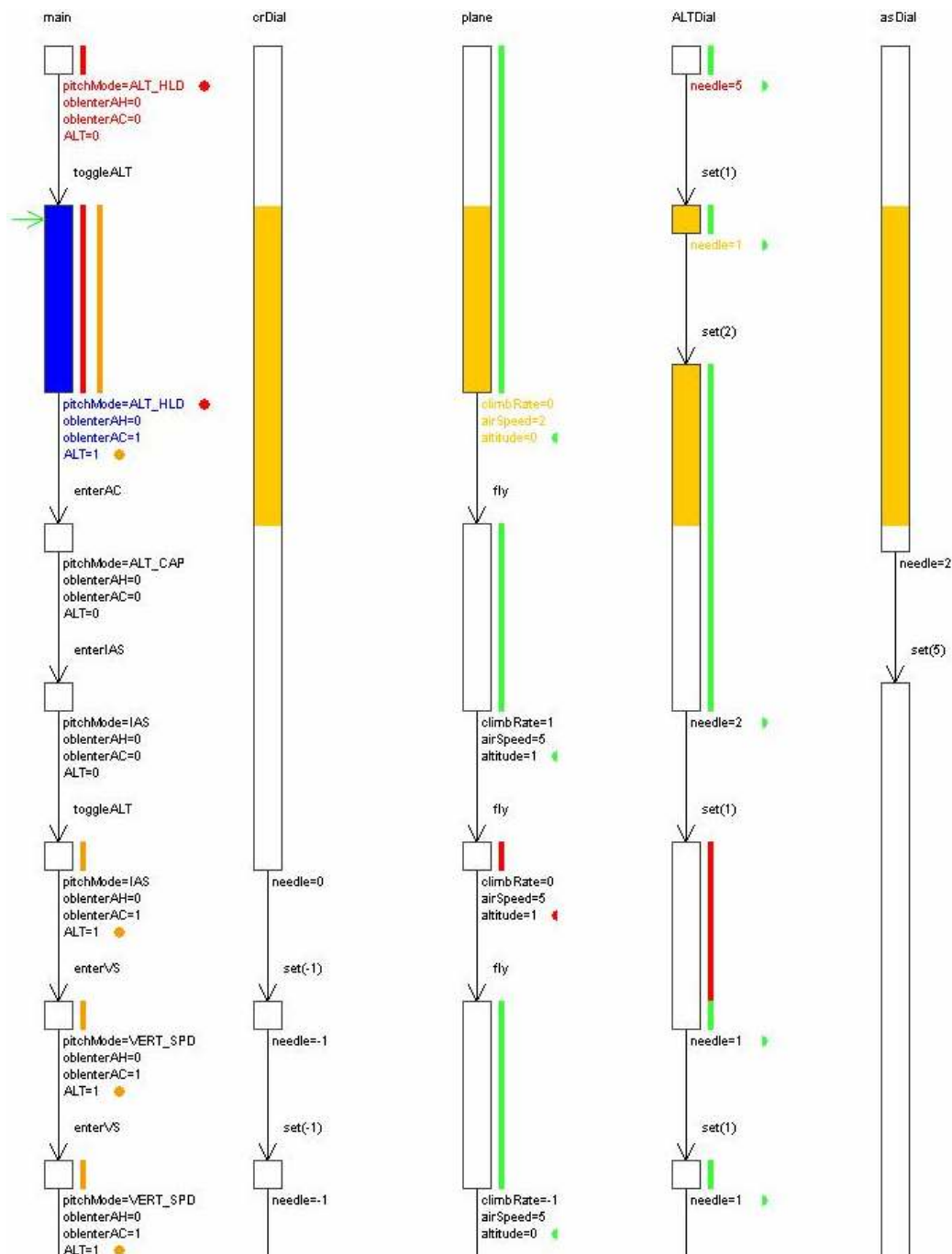


Figure 37 – Logical States representation with filter and markers.

The colored lines are the representation of the markers at the level of states. They complement the circles (representing the same marker) that are near the attributes and show where in the state the marker is active.

The animation is done with the strategy used in Physical States. This representation uses the popups behaviour of Physical States representation.

### 2.1.5 Activity Diagram

This representation is centred on actions and makes use of UML 2.0 Activity Diagrams [OMG05] (for one introduction on UML see [Fowler04]). The rectangles with yellow color are present on the filter result.

In each activity (represented by a round rectangle with the activity name) two small rectangles exist that represent the state before and after it occurs.

In this representation, the information of states is always hidden, but can be shown using the popups functionality. Lines before and after sync bars contain information that will be shown when the mouse passes over. These lines contain the information of the global state of all interactors.

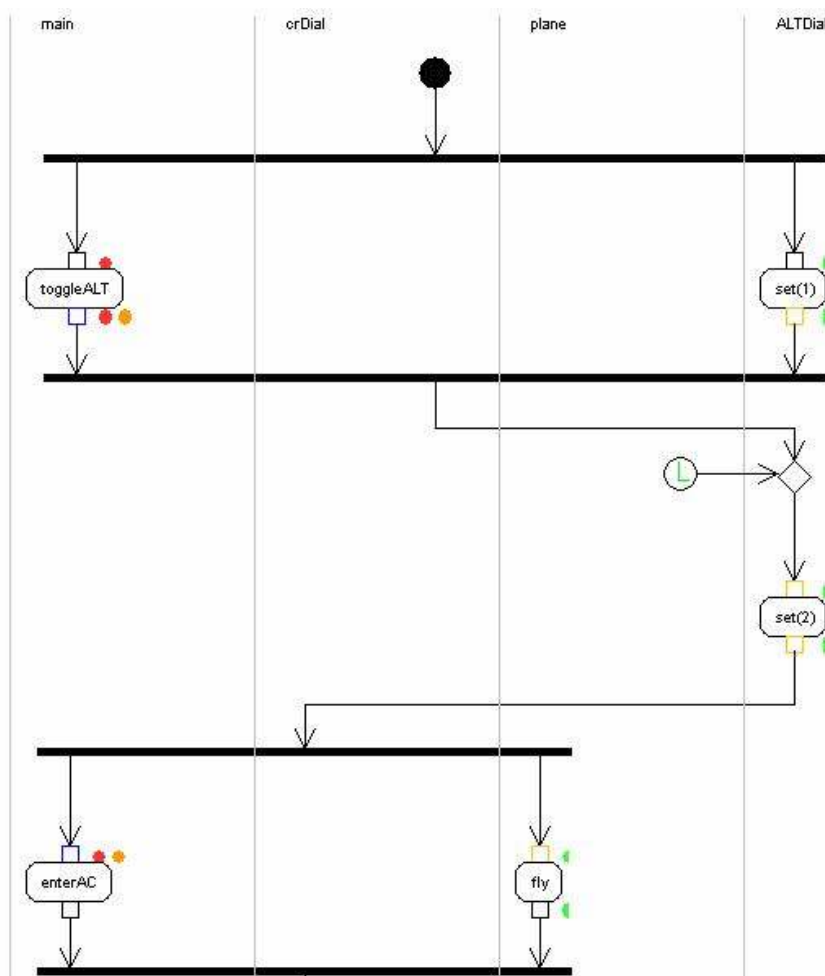


Figure 38 – Activity Diagram.

The animation on this visual representation (see figure 39) is done putting one large rectangle with a background color, which has a minor alpha component to be a little transparent, behind the space between the two sync bars of a transition (if in a state we have only a transition then only it's space is used for the rectangle). The current state is changed sequentially by a time interval.

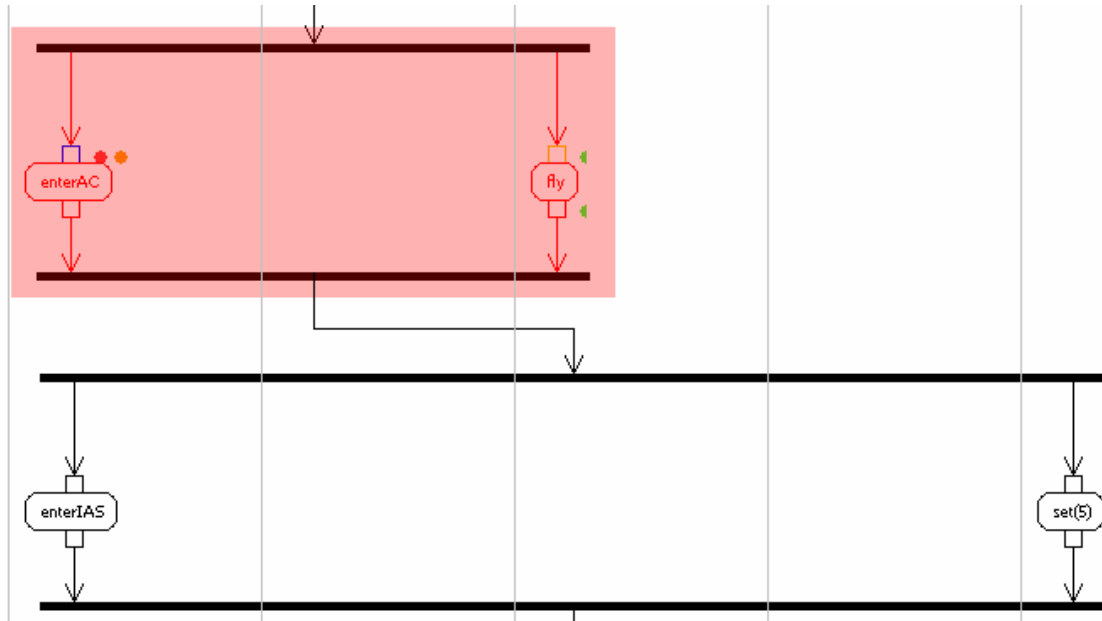


Figure 39 – Activity Diagram animation.

## 2.2. MCP Example Analysis

To illustrate the application of the Trace Visualiser, it will now be applied on an example presented in [Campos01].

Describing the example in full is not relevant in the present context. Only the analysis phase will be described here. After editing and compilation of the model, one of the properties that were checked in the model was the following: whenever the automatic pilot is programmed to achieve a desired altitude, the plane will fly to that altitude and maintain it.

This property can be expressed in CTL as:

$$AG(\text{plane.altitude} < \text{AltDial.needle} \ \& \ \text{ALT} \ \rightarrow \ \text{AF}(\text{plane.altitude} = \text{AltDial.needle} \ \& \ \text{pitchmode} = \text{ALT\_HLD}))$$

It is important to mention that the IVY tool will provide a properties editor that will allow writing of properties without the need of using CTL directly.

When the verification of the formula is tried, SMV informs that it is false and produces a counter-example (in this case with 13 states).



Now, using the visualiser, we will try to discover what problem is pointed out by the trace. Two different trace analysis mechanisms (Markers and Filters) will be used to analyse the trace. We will do that to have an idea of which mechanism is better for doing the analysis.

## 2.2.1 Markers

In a first phase, an attempt was made to identify which states verified the conditions of the property (plane programmed to achieve a desired altitude – **plane.altitude < AltDial.needle & main.ALT**). For that, marking criteria for the sub-formulas of that conjunction were created and the green color associated with them.

The result of the marking showed that the condition holds soon in the second state. The best representation to show that fact was revealed to be the one based on state diagrams. The figure 40 shows the resulting diagram.

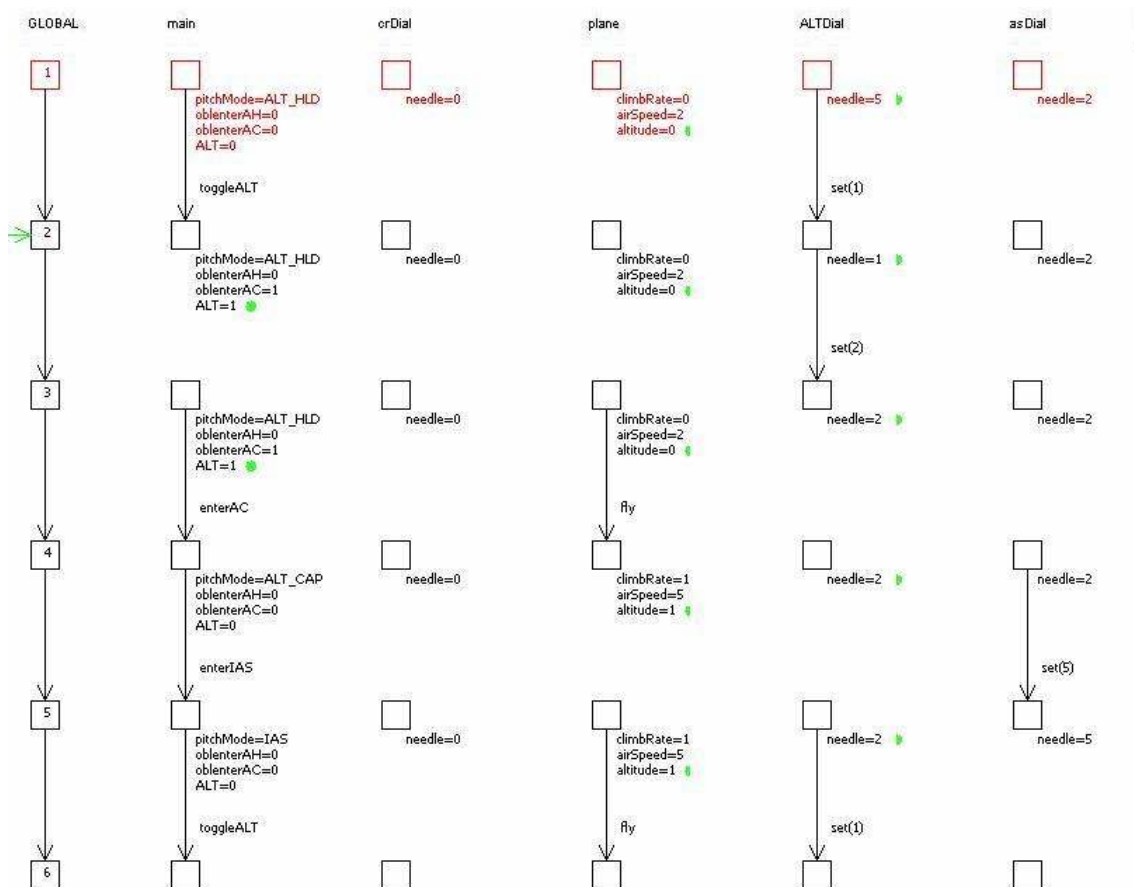


Figure 40 – Verification of property in Physical States representation.

In the initial phase of the model (see top of diagram) only the semicircles resulting from the first criterion (relating two attributes) appear. After the **toggleAlt** action, the marking starts to contain not only the semicircles of the first criterion, but also the circle related to the second criterion.

After this we wanted to confirm that in fact a state with the plane stabilized at the intended height doesn't happen in the trace. Two new markings were created for the expressions **plane.altitude=AltDial.needle** and **main.pitchmode=ALT\_HLD**, both associated with the red color. As it would be expected (see figure 41) none of the states was annotated with two red marks (sign that the conjunction of the two expressions doesn't occur in the trace).

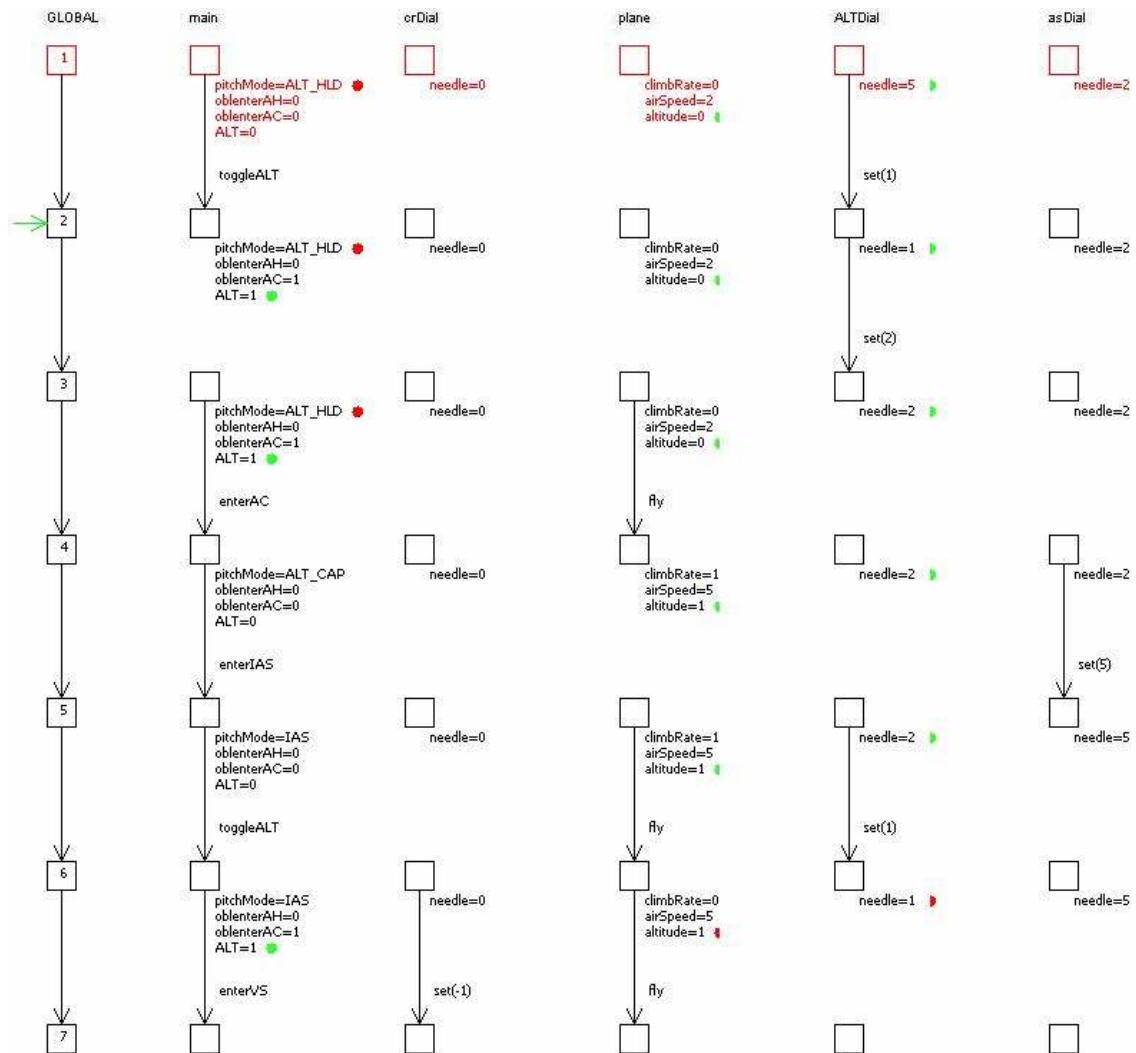


Figure 41 – Verification of the property using Physical States.

Finally, to understand the reason for the problem, the attribute ALT was investigated. This attribute models the fact of the altitude capture of the automatic pilot is armed. The color of the criterion on ALT with was changed to orange. This distinguishes it from the criterion related to altitude.

In this case we opted for a Activity Diagram representation (see figure 42). The analysis of the resultant markings called the attention to what happens when the **enterAC** event occurs (event that occurs in an automatic manner and that is responsible for the activation of an intermediate mode – **ALT\_CAP** – of final approximation to the desired altitude).

In fact, after that event, the altitude capture (**ALT**) is turned off, despite that the plane still in an approximation phase to the programmed altitude and the flight mode is not ALT\_HLD yet (the orange marking disappears, but the red markings are not present).

What the trace shows is that if in that moment the vertical velocity is modified (event set(1)) the automatic pilot changes to maintaining vertical velocity mode, losing the **ALT\_CAP** mode and the altitude capture. After that moment it is possible that the plane exceeds the altitude initially programmed in the automatic pilot (see figure 42) because the automatic pilot is no longer programmed to stop in the altitude indicated in the MCP panel.

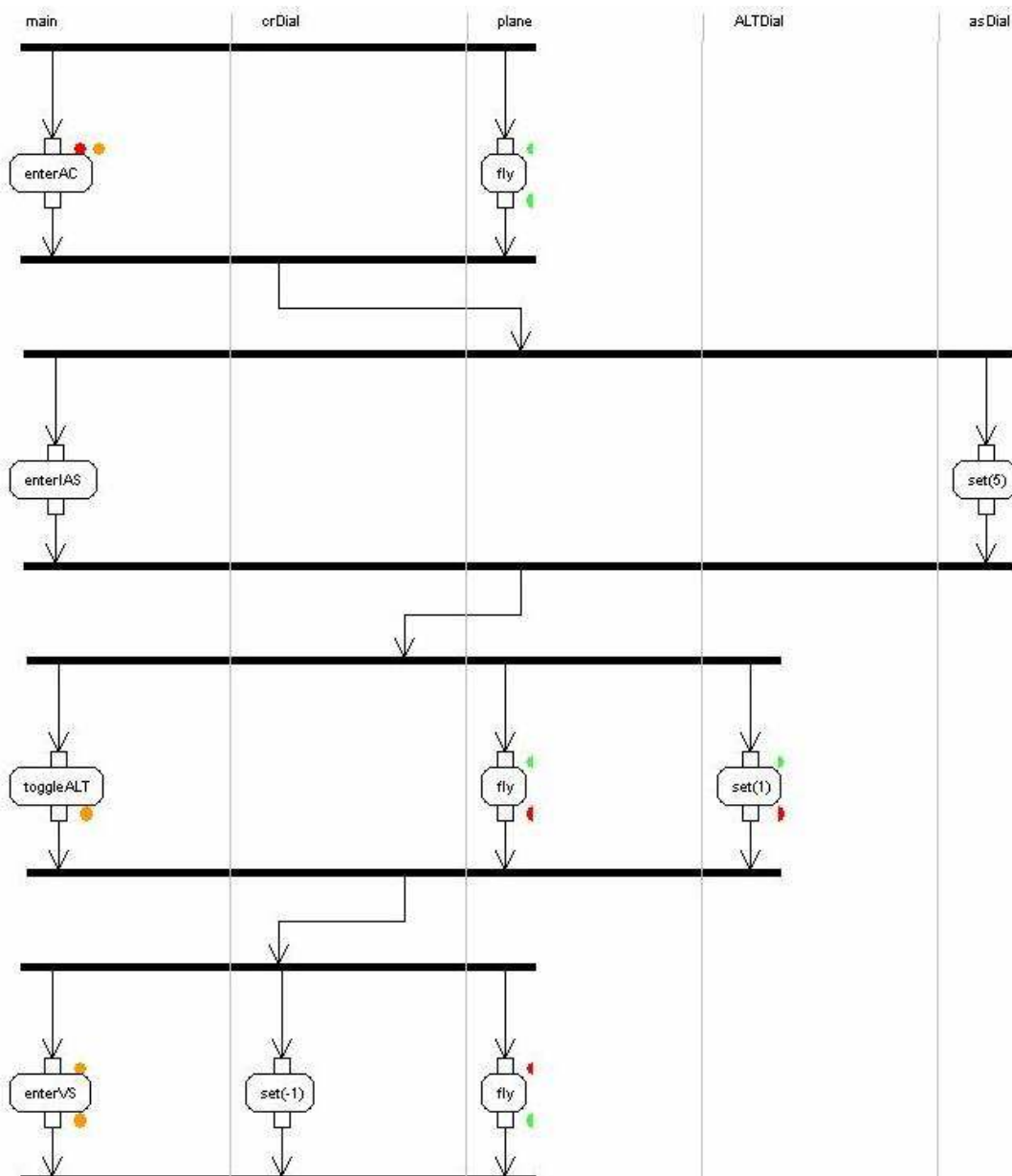


Figure 42 – Verification of the property using Activity Diagram.

## 2.2.2 Filters

We will now perform the same type of analysis with filters. As before, in a first phase an attempt was made to identify which states verified the initial conditions of the property (plane programmed to achieve a desired altitude – **plane.altitude < AltDial.needle** & **main.ALT**). For that only one filter criterion was created. This criterion regards **main.ALT**. Because the filters don't implement the '<' operator, the expression **plane.altitude < AltDial.needle** had to be verified visually by the user. The result of the marking showed that the condition exists soon in the second state. Now a Logical States representation was chosen because it appeared to be more useful when used together with the filters. Figure 43 shows the resultant diagram.

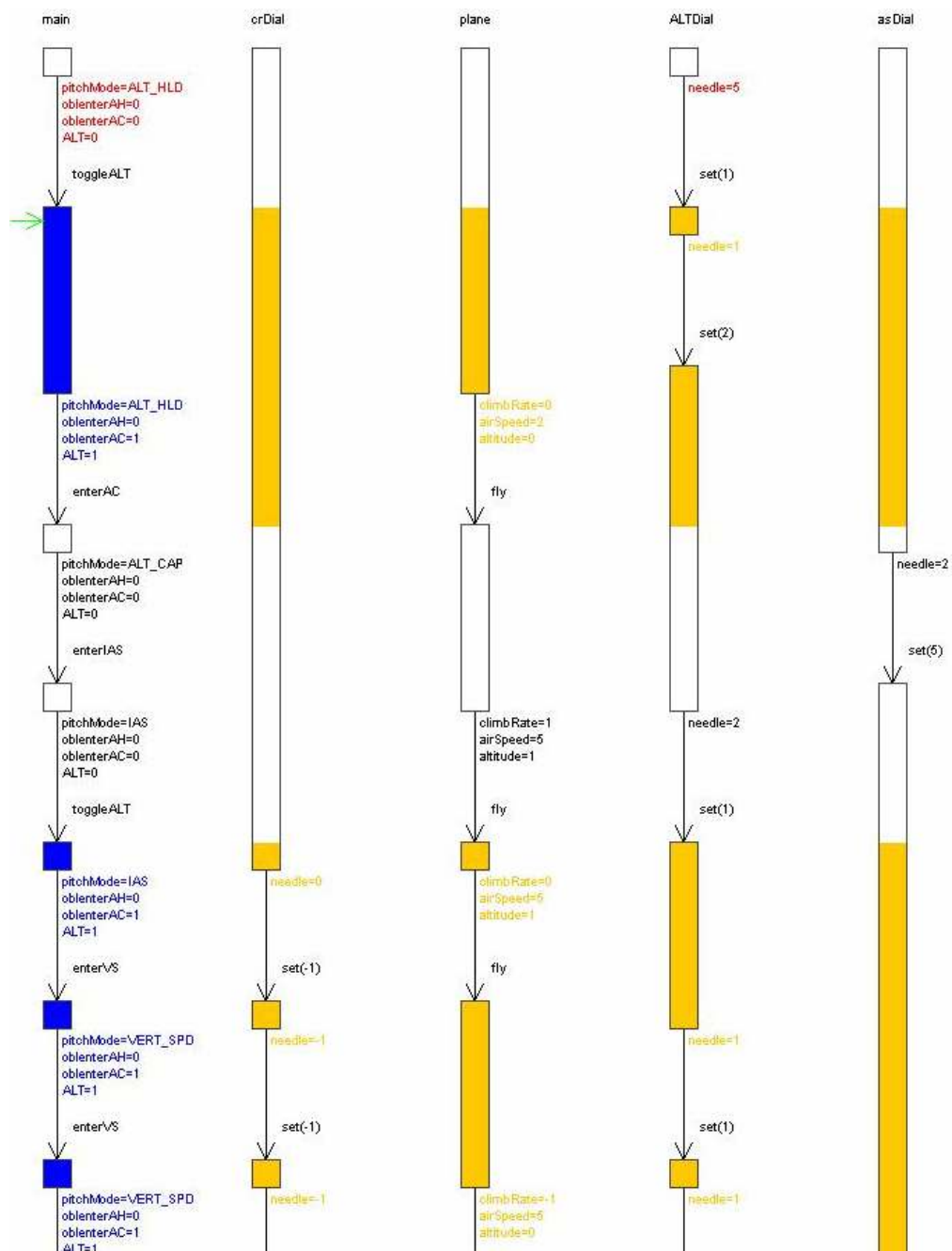


Figure 43 – Verification of the property conditions using Logical States.

A lesson that is possible to learn is that the expressiveness of filters is very limited because they place on the user (in this example), the work of detecting one of the assumptions of the property (**plane.altitude<AltDial.needle**). This is a very negative point, especially when comparing filters's expressive power to markers' support for more comparison operators. The analysis with that limitation is very time consuming and not very useful, the user has to spend a lot of time searching for the states that meet the two conditions at the same time.

An idea to improve the filters is to implement all the comparison operators that markers implement. Other improvement is to associate to each filter condition a different color in order to differentiate between filter conditions (at the moment users cannot associate colors to filter conditions). The last improvement would enable the analysis performed on the previous section on **ALT** attribute to be performed using filters.

After the first step above, it was necessary to check that in fact a state with the plane stabilized at the intended height does not happen in the trace. Again, because filters do not allow for comparisons between attributes, only one new filter criterion on **main.pitchmode=ALT\_HLD** was created. To analyse the result (see figure 44) it is necessary to visually check, in each of the states of the result, if **plane.altitude=AltDial.needle** occurs in any state. This is a time-consuming task.

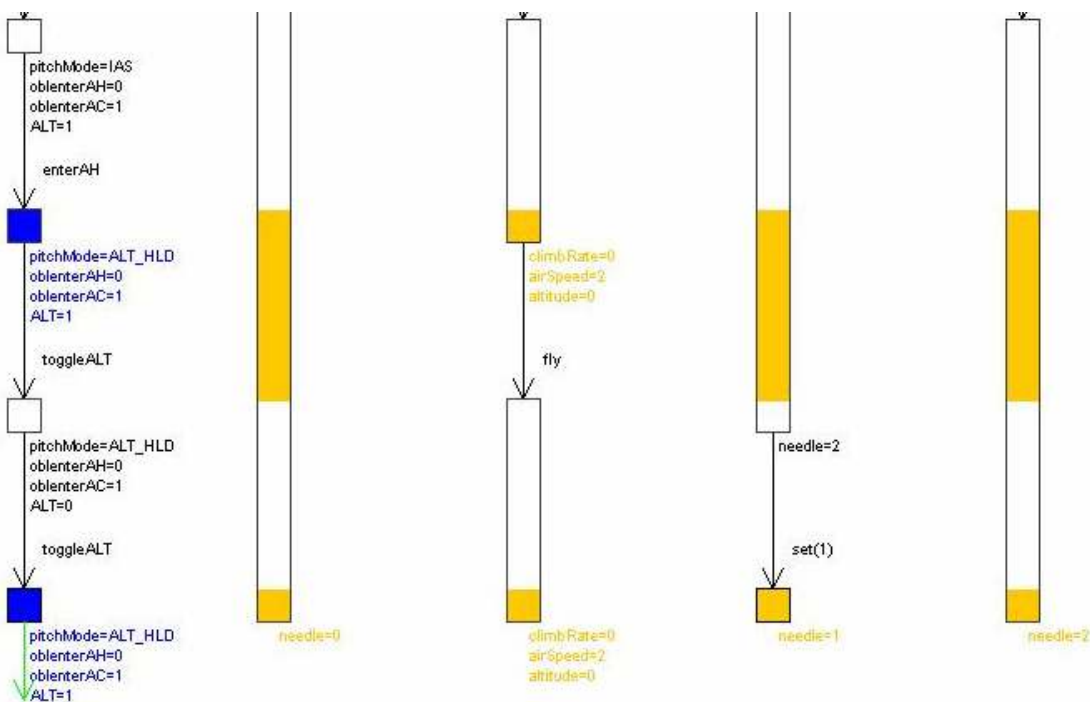


Figure 44 – Verification of the property using Logical States.

Concluding, the comparison between markers and filters is favourable to the first. Filters have many limitations that make an analysis with them very difficult and time-consuming.

# **Part III – Technical Manual for the Final Trace Visualiser**

# 1. Introduction

In this manual the architecture of the final Trace Visualiser is discussed (see figure 45). First a high level view and then the class design is explained. Also a small tutorial to teach how to make a new visual representation is provided.

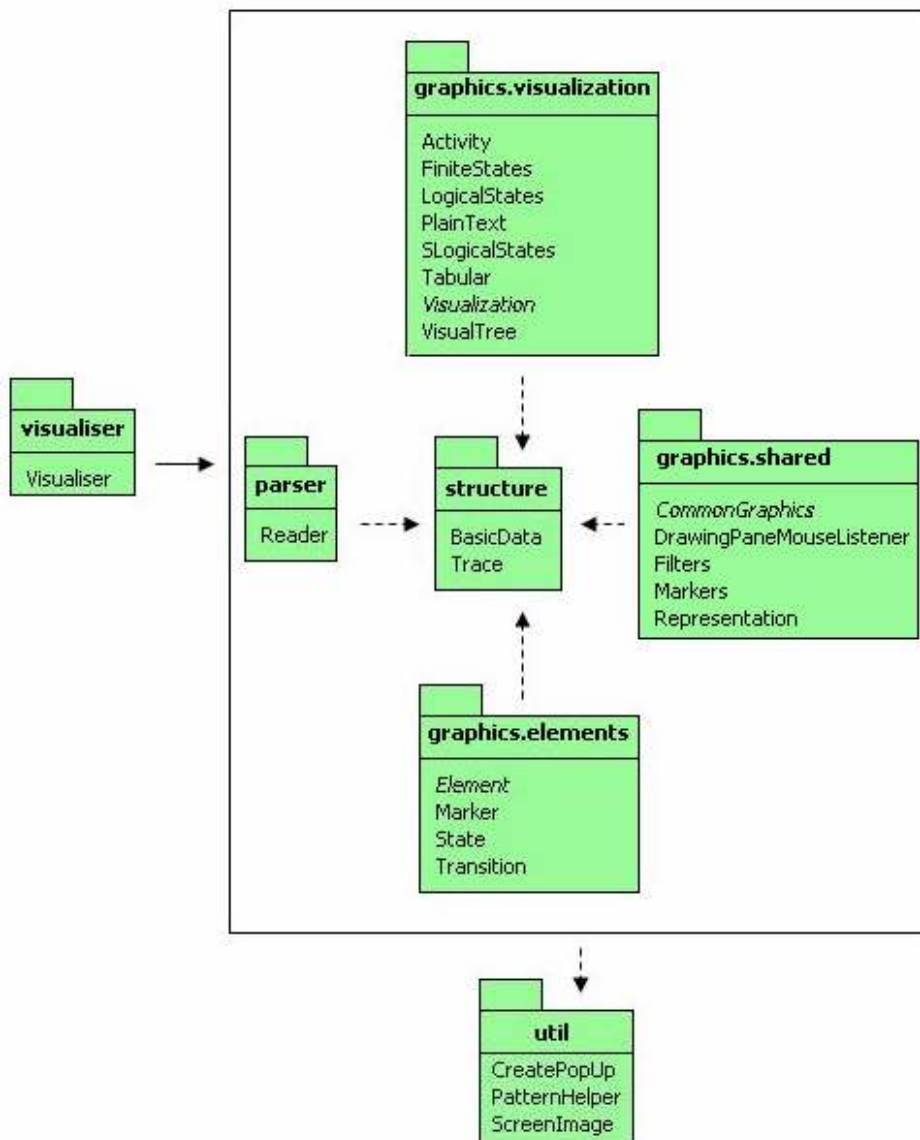


Figure 45 – Package View.

## 2. High Level View

At the highest level, the program must perform the following steps when used:

1. Read and parse a trace file;
2. Store the parsed data in a structure;
3. Visualise the data (visual representations).

The high level view of the architecture has three main components: Parser, Graphics and Structure and a secondary one: Util. For a package view see figure 45.

The Visualiser package is the starting point of the application. It creates the main window that will contain all the visual representations.

The Parser package is responsible for doing the parsing of the trace file.

The Structure package is responsible for storing the information, resulting from the parsing of the trace file.

The Graphics component is responsible for the visualization of the information, obtained by the other components and used by the different visual representations (for example: activity diagram, physical states diagram, etc). It is composed by three sub-components:

- Visualization → is responsible for the visual representations of the traces;
- Shared → is responsible for providing common functionalities to all visual representations;
- Elements → is responsible for the implementation of the graphical objects that are used by the visual representations, for example: Transition (which can be drawn as an arrow).

The Util component provides services that are used by the main components of the Trace Visualiser.



## 2.1. Class Design

To more easily explain the individual packages, we now give their class diagrams and a description of their responsibilities.

### 2.1.1 Visualiser

The **Visualiser** class contains the main routine which creates an instance of the subclass **MainWindow**.

Class **MainWindow** is responsible for the creation of the main window and for starting the initial representation. It allows the user to load trace files (creating the initial representation for them), use the menuBar and end the program. After the initial representation appears the user can switch to the visual representation he wants on a *comboBox* with the available options. It calls the Reader class to construct an instance of Trace that stores the data extracted from the input trace file.

When a file is opened, the **TraceFileFilter** subclass ensures that instead of all types of files, only files which end on .trace are shown. If the option Export on **File** menu is selected, the **TraceFileFilterJPG** subclass ensures that instead of all types of files, only files which end on .jpg are shown.

There is also a **LookListener** subclass that listens to **View** menu selections and changes the LookAndFeel of main window, according to the selection made.

Also an instance of the class **CreatePopUp** is created to be used when showing error messages (for example: file not found).

The class **ScreenImage** is used for exporting the current visual representation to an image file.

### 2.1.2 Parser

Class **Reader** reads the trace file and then calls **Parser** to create the **Trace** structure. It is responsible for the parsing of the trace file and initialization of the Trace structure, which will hold the information retrieved from the file.

The **Parser** subclass has methods to parse all possible declarations on a trace file. The method *parseLine* parses one line of the file and tests the declaration present on it to see which pattern it matches. After doing that, it adds to the Trace structure the information obtained from the declaration. For example, the method *mainActionFound* adds the action of the *interactor* main to the Trace structure. After parsing all the lines of the file, the Trace structure is complete and ready to be used by the visual representations.

### 2.1.3 Structure

Class **Trace** is responsible for the storage and manipulation of a trace's parsed data. It functions as a factory, when a new visual representation is needed the **Trace** class creates it and gives it a copy of the original **BasicData** generated from the trace file.

The **BasicData** class is the most low-level part of the application. It stores the parsed data and allows for simple manipulation of the data. The data stored consists of the names of the variables and the values they have. Furthermore a list of state IDs is kept. These are needed to identify to which states the values in the table belong, as states can be collapsed (if two consecutive states have the same values, the second state can be removed).

The **BasicData** class has its data stored in a hashtable that holds the values for all the interactor's attributes (each interactor attribute has a vector). The key used in the hashtable is the complete name of the interactor attribute. Figure 46 shows the vector in which the state names are stored, and one element of the attributes hashtable (a vector with all the state ordered values of an interactor attribute – for example, in state 1.1 **ALTDial.needle** has value 5, in state 1.2 has value 1, and so on).

<b>States:</b> [1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, 1.11, 1.12, 1.13]
<b>Hashtable Element:</b> <b>ALTDial.needle</b> =[5, 1, 2, 2, 2, 1, 1, 1, 2, 2, 2, 1]

Figure 46 – Data structure.

**BasicData** provides the following methods for querying it:

- **getActionsState()** – returns all the interactor's actions in a given state.
- **getState()** – returns all the interactor's attributes in a given state.
- **getValues()** -- returns all the values for given interactor's attributes.
- **getChangeStatesValue()** – returns all states where a set of interactor's attributes changed to given values (the final set of states is the union of the sets of states returned by each individual interactor attribute).
- **getAllChangedStatesValue()** -- returns all states where a set of interactor's attributes changed to given values (the final set of states is the intersection of the sets of states returned by each individual interactor attribute).
- **getStatesValue()** -- returns all states where a set of interactor's attributes are equal to given values (the final set of states is the union of the sets of states returned by each individual interactor attribute).
- **getAllStatesValue()** -- returns all states where a set of interactor's attributes are equal to given values (the final set of states is the intersection of the sets of states returned by each individual interactor attribute).

**BasicData** has also some other query methods that are:

- **getAtribInteractor()** → returns a vector with all attribute names of an interactor;
- **getInteractors()** → returns a vector with all interactor names present on a trace;
- **getAtribsStateInteractor()** → returns a vector with all attributes of an interactor, and their respective values, in a given state.
- **getVariables()** → returns a vector with all attributes names on a trace in the following form: <interactor>.<attribute>;
- **getRow()** → returns a vector with the values in all states of an interactor attribute;
- **getColumn()** → returns all the values for all attributes of interactors in a state.

The **update()** method is used to update the BasicData information to reflect a filter's result. For example, if a filter returns states 1, 2 and 3 as result and the **update()** method is executed, the following queries to the BasicData instance will only return results if they are related with those three states. This method is used to implement subfiltering.

**PatternHelper** class contains pattern methods which are used by some classes. Essentially contains methods for working with strings.

## 2.1.4 Graphics.shared

All the instances of the Representation class have to extend the abstract **CommonGraphics** class. By keeping track of the currently activated (static) **CommonGraphics** object, filtered data can be given to the correct graphics class. For example, if the user is working with a Tabular Representation, the *currentGraphic* object is set to that representation, ensuring that when a filter is applied, it is applied to that object.

The **CommonGraphics** class creates the toolbars used for animation, switching between visual representations and formula and filter *textAreas*. It also creates an instance of *JSplitPane* class that has on the left a panel (which holds the current visual representation) and on the right a tabbedPane (to use markers, which will be explained later on this chapter, and filters).

The **Representation** class (see figure 47) extends **CommonGraphics** class and is responsible for showing the visual representation that the user has chosen in the *comboBox* for that effect. To do that it has an instance of an abstract **Visualisation** class which has seven concrete classes that represent each of the visual representations. It also receives the calls to filter methods and calls the relevant method on the current visual representation.

A listener of the *comboBox* with all the possible visual representations, changes the current visual representation, according to the selection by the user. In the new visual representation, all filter and marker operations previously selected by the user are applied.





The **DrawingPaneMouseListener** class is used to show popups with state or marker information, when the popups option is enabled. The information is shown when the user places the mouse over the graphical objects representing the states or the markers. It is used on all visual representations based on states. These are: Activity Diagram, Physical States and Logical States.

## 2.1.5 Graphics.visualisation

The abstract **Visualisation** class is the “mother” of all visual representations. It provides services related with filters that all the representations use. To implement a new visual representation for the visualiser it is necessary to extend this class, and in the **Trace** class add one method that returns an instance of the visual representation with a copy of the original **BasicData** instance (all the information from the trace file without filters applied).

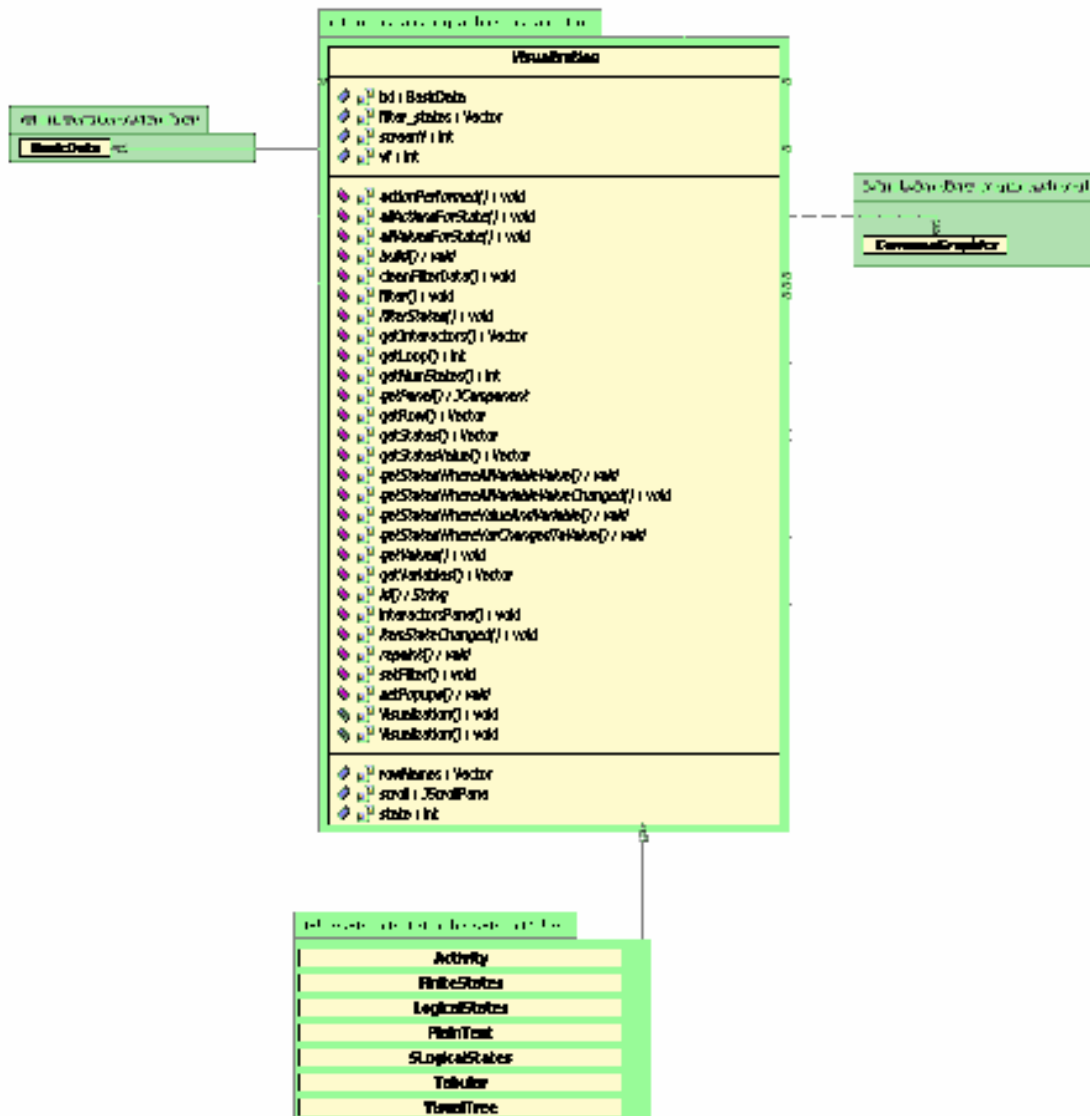


Figure 50 – Visualisation class diagram.

The abstract **Visualisation** class (see figure 50) has abstract methods to work with filters that must be implemented to create a new visual representation. In the remaining of this section these will be enumerated.

The method **build()** has to be implemented to create the component that will hold the visual representation (for example a panel, a tree, a table, etc.). To draw a visual representation that uses a panel, it is necessary to implement the **paint(Graphics g)** method.

The abstract method **getScroll()** has to be implemented and it should return the component (inside a *scrollPane*) that the **Representation** class will use. This method is needed to make the visual representation scrollable.

The method **getPanel()** has to be implemented and it should return the JComponent, which contains the visual representation, that will be exported to an image file.

The abstract method **actionPerformed(ActionEvent e)** has to be defined to implement the functionality associated to animation controls.

The abstract method **itemStateChanged(ItemEvent e)** has to be implemented to deal with the popups functionality.

The method **interactorsPane(Graphics g)** is used to show interactors names when exporting the visual representation to an image file.

A **BasicData** class instance is used to perform queries on it. This instance is needed to construct the visual representations, and to work with filters. This **BasicData** instance is updated when a filter is applied.

The following visual representations, available in the Trace Visualiser, implement all the necessary methods:

- **Trace** → the textual representation originally produced by the SMV;
- **Tree** → tree representation of trace states;
- **Tabular** → tabular representation similar to the one existing on the SMV from Cadence Labs;
- **Physical States** → graphical state-based representation of trace states;
- **Logical States** → similar representation to the previous where the trace states are pre-processed to eliminate artificial states introduced by the compilation process;
- **Activity Diagram** → representation centred on actions based on UML 2.0 Activity Diagrams [OMG05] (for one introduction on UML see [Fowler04]).

The three visual representations, based on states, have a similar structure. These representations are: Physical States (**FiniteStates** class), Logical States (**SLogicalStates** class) and Activity diagram (**Activity** class). They all have a subclass **DrawingPane**, that extends JPanel, and implements the **paint()** method to draw the specific representation. They also have an instance of BasicData to do the queries needed to build the representation or to work with filters.

The **VisualTree** class implements the tree representation. The tree is constructed with the **BasicData** information and uses cell *renderers* and *tooltips* to implement the filters and markers functionality.

The **Tabular** class implements the tabular representation and is similar to the previous representation on its structural design (also uses cell renderers and tooltips). The only difference is that the component that holds the representation is a `JTable` instead a `JTree`.

## 2.1.6 Graphics.elements

The abstract **Element** class is extended by **State**, **Transition** and **Marker** classes and contains shared methods for coloring and storage of displayable data for labels and mouse over events. It also has the abstract method "hit", used to define when a mouse is inside the graphical object.

Class **State** has methods which are typical for the state object, mainly concerning the coordinates of the rectangle (or other graphical object) that represents the state in the graphics.

Class **Transition** has methods which are typical for the transition object, mainly concerning the coordinates of the arrow (or other graphical object) that represents the transition in the graphics.

Class **Marker** has methods to draw itself in a given panel in any visual representation. The markers are used to mark states with relation to criteria defined over state attributes. The criteria are defined over states, establishing relations ( $=$ ,  $>$  and  $<$ ) between attribute pairs or between values and attributes. A color is associated with each criterion, and all states that verify one given criterion are annotated with the colour associated with it.

In the case of attributes comparison two filled semicircles are drawn, with the chosen color. Each semicircle is drawn near each of the attributes, this way the attributes are visibly related by the condition. In the case of comparison between values and attributes, filled circles with the chosen color are drawn. If the popups option is enabled it is possible to see the condition represented by each marker putting the mouse over it (using the **hit()** method).

## 2.1.7 Util

Class **PatternHelper** contains pattern matching methods which are used by some classes. These are essentially methods for working with strings.

Class **CreatePopUp** creates simple popups to show any desired information such as state or error messages information.

Class **ScreenImage** creates a *BufferedImage* for any Swing component and is used to export the visual representation to an image file.



## 2.2. How to make a new visual representation?

To create a new visual representation we have to answer the following questions: Which geometric shapes to associate with a state or a transition? How to show a filter's result? How to perform the animation of the representation? How to add the markers to the representation? Which behaviour do we want in the popups functionality?

With the answers to the previous questions and a simple analysis of the implementation code, we can easily make a new visual representation. It is possible to do complex visual representations as well as simple ones. The difference is the time and effort we want to dispend on it.

The visual representation that will be created, in this section, is the Activity Diagram representation (**Activity class**). To create it there are some alterations and implementations to make. These are the following:

- **Class Representation**
  - Add some lines in the `ItemStateChanged()` method to make it possible to choose the new visual representation in the `comboBox` used for that effect;
- **Class Trace**
  - Implement the **`getActivity()`** method that returns a instance of the Activity Diagram representation, including one copy of the original `basicData`.
- **Class Activity**
  - Implement the **`id()`** method that returns the name of the visual representation;
  - Implement the **`build()`** method, which constructs the visual representation. It uses a `JSwing` visual component to hold the visual representation, and creates graphical objects to represent the states and actions of an interactor;
  - Implement the **`getPanel()`** method, which returns the `JComponent` that holds the visual representation. It is used to export the visual representation to an image file;
  - Implement the **`getScroll()`** method, which gives the `JComponent` that holds the visual representation, inside a `JScrollPane`. It is used to add the visual representation to the *Visual Representation Area (left panel of a `JSplitPane`)*, and to make it possible to use the scrollbar because normally the visual representations area is larger than the screen size.
  - Implement the **`filterStates()`** method, which reflects the results of the filter in the visual representation. For example, it can colorize the graphical

objects (representing the states) that are on a filter's result with a different color.

- Implement the **setPopups()** method, which is used to know what behaviour to have, in the visual representation, when popups checkbox is selected;
  - Implement the **repaint()** method, which is used to repaint the JComponent that holds the visual representation. Its task is reflecting the changes in the visual representation, when using filter, marker or animation functionalities.
  - Implement the **actionPerformed()** method to know what action to perform when a animation button is pressed. Also the action related to clean button, used in filters cleaning, is implemented here;
  - Implement the **itemStateChanged()** method, which is used to know what action to take when the popups' checkbox state changes. Normally, the visual representation has two sub representations that are related with the state of the popup checkbox.
  - Implement all the filter methods (see 2.1.3 for a description of each filter method), which are the following:
    - **getValues();**
    - **getStatesWhereValueAndVariable();**
    - **getStatesWhereAllVariableValue();**
    - **getStatesWhereAllVariableValueChanged();**
    - **getStatesWhereVarChangedToValue().**
- **Class <Name of the New Graphical Object>** *the implementation of this class is optional, but if implemented has to extend **Element** class.*
    - Implement the **hit()** method (to use on Popups functionality), which tells when the mouse is inside this new graphical object;
    - Implement the **draw()** method to draw the new graphical object in any JComponent.

Now, a more detailed description of how to do the enumerated changes and implementations is provided.

First the new representation is added to the set of choices provided by the Trace Visualiser. To do that some lines must be added in the **Representation** class. The lines are added in the **itemStateChanged(ItemEvent e)** method. These lines are to make it possible to switch to the Activity representation in the comboBox used for switching between visual representations. The lines are:

```
Line 1: else if (item.compareTo("Activity Diagram") == 0) { vis = 7; }
```

```

switch (vis) {
...
Line 2: case 7:
    Line 3: visual = structure.getActivity(screenY);
    Line 4: buildLState("Activity Diagram");
    Line 5: varcolor.enableButtons();
    Line 6: popups.setSelected(true);
    Line 7: popups.setEnabled(false);
    Line 8: varcolor.apply();
    Line 9: jc.setSelectedItem("Activity Diagram");
        break;
}

```

Line 1 associates the name of the visual representation with a number, to use in the switch instruction.

Line 2 adds, to the switch instruction, a new case holding the necessary code for switching to the new visual representation.

Line 3 gets, from the **Trace** class, an instance of the new visual representation (in this example Activity diagram), which includes a copy of the original BasicData, obtained from the trace file. For that the implementation of the method `get<NameVisualRepresentation>` is needed. That method can have parameters that the new visual representation needs to do its drawing. The following lines in **Trace** class are needed to do that.

```

public Visualization getActivity(int sc) {
    return new Activity(original.getBasicData(), sc);
}

```

Line 4 calls a method that creates a JComponent that holds the visual representation (from variable **visual** obtained before). This method can also construct a JPanel that contains the names of the *interactors* if it is needed.

Line 5 enables the markers buttons, available on the right panel.

Lines 6 and 7 tell that popups will be always enabled in the new visual representation.

Line 8 is used to apply the markers to the new visual representation.

Line 9 tells the *comboBox* for selecting between visual representations that the currently selected representation is the new one.

Now the class of the new visual representation must be created. This class has to extend the **Visualization** class. Class Visualization has the abstract methods that **Activity** must implement to be a concrete visual representation.

The first method to implement is the **id()** method, which returns the name of the visual representation. It is used to identify the visual representation because some behaviour is shared by visual representations, but some times it is needed to distinguish between them and implement small differences on it. For example, the code to save the visual representations to an image file is equal in all of them except on Tabular representation.

```

public String id() {
    return "Activity Diagram";
}

```

The second method to implement is the **build()** method, which draws the visual representation. It can create a JPanel and redefine its **paint()** method to do the drawing of the representation. It is also possible to use other JSwing component such JTree or JTable to hold the representation. **DrawingPane** in the following code is a class that extends JPanel and uses information obtained with the **buildDiagram()** method. That method creates the graphical objects representing the states and transitions and the class **DrawingPane** uses these on its paint method.

```

public void build() {
    inter = bd.getInteractors();
    buildDiagram();
    drawingPane = new DrawingPane();
    ...
    drawingMouseListener = new DrawingPaneMouseListener(drawingPane, stat);
    drawingPane.addMouseMotionListener(drawingMouseListener);
}

```

The third method to implement is the **getPanel()** method, which returns the JComponent that holds the visual representation. It is used to save the visual representation (inside the JComponent) to an image file.

```

public JComponent getPanel() { return drawingPane; }

```

The fourth method to implement is the **getScroll()** method, which returns the JComponent that holds the visual representation, inside a JScrollPane. It is used to add the visual representation to the left component of the JSplitPane on the main frame. The implementation is:

```

public JScrollPane getScroll() { return drawingPane.scroller(); }

```

The next methods to implement are the filter methods:

- `getValues(Vector<String> selection);`
- `getStatesWhereValueAndVariable(Vector<String> variables, Vector<String> values);`
- `getStatesWhereAllVariableValue(Vector<String> variables, Vector<String> values);`
- `getStatesWhereAllVariableValueChanged(Vector<String> variables, Vector<String> values);`
- `getStatesWhereVarChangedToValue(Vector<String> variables,`

```

Vector<String> values);
    • filterStates(Vector<Vector<String>> stat).

```

One example of the implementation of one of the filter methods is the following code:

```

public void getStatesWhereValueAndVariable(Vector<String> variables,
    Vector<String> values) {
    getStates(1, variables, values);
    filterStates();
}

```

The other methods are implemented similarly. The **getStates()** method (implemented in the Visualization class) is used to fill the vector of strings **filter\_states** (each string representing a state number) that holds the filter result. The filter result is obtained calling specific methods in the **BasicData** instance. The **states()** method (implemented on the Activity class) is used to reflect the result in the visual representation and for that uses the **filter\_states** vector.

### Visualization

```

public Vector<Vector<String>> getStates(int f, Vector<String> variables,
    Vector<String> values) {
    Vector<Vector<String>> states = new Vector<Vector<String>> ();
    String s;
    switch (f) {
        case 1:
            for (int i = 0; i < variables.size(); i++) {
                s = variables.elementAt(i);
                Vector<String> st = bd.getStatesValue(s, values.elementAt(i));
                states.add(st);
            }
            break;
        ...
    }
    filter_states = states;
    bd.update(states, vf);
    return states;
}

```

### Activity

```

private void states() {
    cleanModel();
    String ID;
    Integer na;
    Vector<String> vta;

    for (int m = 0; m < filter_states.size(); m++) {
        vta = filter_states.elementAt(m);
    }
}

```

```

    for (int j = 0; j < vta.size(); j++) {
        ID = vta.elementAt(j);
        ID = ID.substring(ID.lastIndexOf('.') + 1);
        na = new Integer(ID);
        allValuesForState(na.intValue());
    }
}

drawingMouseListener.enabled = true;
drawingPane.bbox = false;
drawingPane.repaint();
}

```

The method **filterStates()** is implemented to know how to reflect, in the visual representation, the results of the filter. The implementation of **filterStates()** method in this case is:

```

public void filterStates(Vector<Vector<String>> stat) { states(); }

```

The next method to implement is **setPopups()**, which is used to know what behaviour to implement, in the visual representation, when popups checkbox is selected. In the present example of Activity diagram, a MouseMotionListener is activated to listens mouse over events from the states representations and present the state information as a popup label.

```

public void setPopups() {
    drawingMouseListener.enabled = true;
    drawingPane.bbox = false;
}

```

The next method to implement is **repaint()**, which is used to repaint the JComponent that holds the visual representation. Its task is reflecting the changes in the visual representation, when using filter, marker or animation functionalities.

```

public void repaint() { drawingPane.repaint(); }

```

Other method to implement is **actionPerformed()**, which is used to know what action to take when animation buttons are pressed. Also the action related to clean button, used in filters cleaning, is implemented here. The method implementation is the following:

```

public void actionPerformed(ActionEvent e) {
    String status = e.getActionCommand();
    if (status.compareTo("stepBackward") == 0) {
        drawingPane.stepBackward();
    }
    else if (status.compareTo("stepForward") == 0) {
        drawingPane.stepForward();
    }
}

```

```
    }  
    ...  
    drawingPane.repaint();  
}
```

Finally the last method to implement is **itemStateChanged()**, which is used to know what action to take when popups checkbox state changes. Normally, the visual representation has two sub representations that are related with the state of the popup checkbox. In this example nothing changes if the checkbox is selected or not. But in other visual representations it may be desirable to hide some information, when popups checkbox is selected, and show it as popup labels. In this case the implementation of the method is empty.

By implementing all these methods we have a new visual representation ready to be used.

If new graphical objects are needed, for a new visual representation, the **Element** class needs to be extended by the class that will represent the graphical object. Two methods must be implemented: the **hit()** method, which tells when the mouse is inside this new graphical object; and a **draw()** method to draw the new graphical object in any JComponent. The creation of new graphical objects can also be made in an easier way, using the classes already implemented (**State**, **Transition** and **Marker**) and drawing them in different shapes. For example, the **State** class may have different methods to be drawn as a rectangle, square, circle, oval, etc.

# **Part IV – Conclusions and Future Work**



# 1. Conclusions and Future Work

The Trace Visualiser is responsible for presenting the results of the verification process to the IVY user in a manner that facilitates the understanding of the meaning of the trace. In the present work, new visual representations were added to an early Trace Visualiser, developed by the present author, in a previous work. The idea was to create standard visual representations, to allow everyone to easily understand the problems pointed at by a trace. To achieve that, a new visual representation based on Activity Diagrams of UML 2.0 was created.

Also, an important analysis mechanism was added, the markers. This powerful mechanism helps in the identification of undesirable scenarios, which the engineers can use to obtain information on how they impact on the system's design, and know how to correct problems. The markers proved to be more generic and useful comparing to the other mechanism, filters. For example, they can work with conditions of the types (>,<=,<) and the filters only work with equality conditions.

The objectives fulfilled in the present work were:

- to analyse the implementation code of the early version of the Trace Visualiser.
- to improve it's visual representations;
- to add to it more traces analysis mechanisms;
- to improve it's filters functionality;
- to improve the animation in all the visual representations;
- to improve it's graphical interface;
- to implement sub-filtering on it's functionality of filters;
- to add more standard visual representations to the visualiser.

Now the graphical aspect of the Trace Visualiser is more appealing. This was done by adding icons to buttons and also toolbars to include the buttons. The internal frames were replaced by one JSplitPane that holds only a visual representation at each time. Also, it is possible to dynamically change the LookAndFeel of the application.

In terms of future improvements, the visualiser should be able to provide type validation for attributes comparison, when we are adding markers from conditions based on them. For doing that a file, with information oo the types of all attributes, is needed. The idea is for the i2smv compiler to create a XML file with all the information of *interactors's* attributes. Then, queries to the file will obtain the necessary information for doing the validation.

Finally I want to refer that a paper was submitted to the Interacção'2006 Conference and approved for publishing [Sousa06a]. The article describes the present Trace Visualiser (without

some improvements currently implemented because the work proceeded after the submission of the paper) and shows how it can be used to analyze an example of an interactive system (using markers). It also describes each visual representation and discusses there advantages and disadvantages in terms of graphical representation. The article was a way to present to the academic community the work made on the Trace Visualiser and to know what kind of acceptance the IVY tool may have when fully implemented.

## References

- [Campos01] Campos, J. C., Harrison, M. D. Model Checking Interactors Specifications. *Automated Software Engineering*, 8(3-4): 275-310, August, 2001.
- [Campos04] Campos, J. C. Análise de Usabilidade Baseada em Modelos, *Interação 2004*, 1ª Conferência Nacional em Interação Pessoa-Máquina, 171-176, Grupo Português de Computação Gráfica, July, 2004.
- [Chan02] Chan, Patrick. The Java(TM) Developers Almanac 1.4, Volume 1: Examples and Quick Reference (4<sup>th</sup> Edition), 2002.
- [Cheaney91] Cheaney, E. 'ASRS Introduces...'. *ASRS Directline* (1), 1995.
- [Clarke86] Clarke, E. M. Emerson, E. A., Sistla, A. P. Automatic Verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2): 244-263, 1986.
- [Clarke99] Clarke, E. M. Grumberg, O., Peled, D. *Model Checking*. MIT Press, Cambridge, Massachusetts, U.S.A, 1999.
- [Dwyer97] Dwyer, M. B., Carr, V., and Hines, L. Model Checking Graphical User Interfaces Using Abstractions. In: M. Jazayeri and H. Schauer (eds.): *Software Engineering – ESEC/FSE, 97, N° 1301* in Lecture Notes in Computer Science. Springer, pp. 244-261, 1997.
- [Fowler04] Fowler, M. *UML Distilled, third edition*. Object Technology Series, Addison-Wesley, 2004.
- [Heitmeyer98] Heitmeyer, C., Kirby, J., and Labaw, B. Applying the SRC Requirements Method to a Weapons Control Panel: An Experience Report. In: *Proceedings of the Second Workshop on Formal Methods in Software Practice (FFMS '98)*. pp. 92-102, 1998.
- [McMillan93] McMillan, K. L. *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic, 1993.
- [OMG05] Object Management Group, *Unified Modelling Language: Superstructure, v. 2.0*. OMG Specification: formal/05-07-04, August, 2005.
- [Palmer95] Palmer, E. "Oops, it didn't arm." – A case study of Two Automation Surprises. In: R. S. Jensen and L. A. Rakovan (eds.): *Proceedings of the Eighth International Symposium on Aviation Psychology. Columbus, Ohio*, pp. 227-232, 1995.
- [Ridder05] de Ridder, A., Posadas, F. M., Campos, J. C. *Technical Guide for the Visualiser Component*. IVY technical report IVY-TR-5-01, June, 2005.

- [Roever98] de Roever, W.-P. The Need for Compositional Proof Systems: A Survey. In: W.-P. de Roever, H. Langmaack, and A. Pnueli (eds.): *Compositionality: The Significant Difference*, Vol. 1536 of *Lecture Notes in Computer Science*. Springer, pp. 1-22, 1998.
- [Ryan91] Ryan, M., Fiadeiro, J., Maibaum, T. Sharing actions and attributes in modal action logic. *Theoretical aspects of Computer Science*, Vol. 256 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 569-593, 1991.
- [Silva06] J. C. Silva and J. C. Campos and J. Saraiva. Combining Formal Methods and Functional Strategies Regarding the Reverse Engineering of Interactive Applications. In Gavin Doherty and Ann Blandford, editors, *Interactive Systems: Design Specification and Verification*, vol. 4323 of *Lecture Notes in Computer Science*, Springer-Verlag, 2006 (in press).
- [Sousa06] Sousa, Nuno M. E. IVY Trace Visualiser, Relatório de Opção III, DI/UM, February, 2006.
- [Sousa06a] Sousa, Nuno M. E., Campos, J. C. IVY Trace Visualiser, In Chambel, Nunes, Romão and Campos (eds.): *Interação 2006 – Actas da 2ª. Conferência Nacional em Interação Pessoa-Máquina*, Grupo Português de Computação Gráfica, pp. 181-190, Outubro, 2006.
- [SWEBOK01] Guide to the Software Engineering Book of Knowledge, trial version 1.0, IEEE, May, 2001.
- [Woods94] Woods, D. D., L. J. Johannesen, R. I. Cook, and N. B. Sarter. Behind Human Error: Cognitive Systems, Computers, and Hindsight. State-of-the-Art Report SOAR 94-01, CSERIAC, 1994.

## Acronyms

CTL	Computacional Tree Logic
MAL	Modal-Action Logic
IVY	Interactors VerifYier
SMV	Symbolic Model Verifier
HCI	Human-Computer Interaction
ISO	International Organisation for Standardisation
DIS	Draft International Standard
POSC	Programa Operacional Sociedade do Conhecimento
FEDER	Fundo Europeu de Desenvolvimento Regional
FCT	Fundação para a Ciência e Tecnologia