Universidade do Minho
Escola de Engenharia

Nuno Miguel Eira de Sousa

WildAniMAL
MAL Interactors Model Animator

Novembro de 2012

Universidade do Minho
Escola de Engenharia
Departamento de Informática

Nuno Miguel Eira de Sousa

WildAniMAL
MAL Interactors Model Animator

Dissertação de Mestrado
Mestrado em Engenharia Informática

Trabalho realizado sob orientação de

Professor José Creissac Campos

Novembro de 2012

Acknowledgements

I want to thank my thesis advisor José Creissac Campos for continuously helping me while doing this work, by making many corrections and valuable suggestions for the implementation of the WildAniMAL and for pointing errors and problems during the design and implementation of the tool.

I also want to thank Manuel Sousa for providing me with a practical example of an Ipod device model, that enabled me to show how the WildAniMAL plugin can be used. Additionally I thank him for testing the WildAniMAL plugin, with interactor models that he developed in his own work.

Finally, I want to thank my employer - Computer Graphics Center – for giving me a day a week for education, during the last three years, thus enabling me to conclude this master's course with the current work.

Abstract

The IVY Workbench is a tool for modeling and analysis of interactive systems which has been developed at the Department of Informatics of the University of Minho (http://ivy.di.uminho.pt). It's a platform developed in Java, using a plugins mechanism. The available plugins include a set of editors (textual and graphical) and tools to analyse the behaviour of the models. The experience on using the tool has demonstrated the need for a model animator which could enable a first interactive evaluation of the models. Therefore this dissertation describes the design and implementation of WildAniMAL - a MAL (Modal Action Logic) interactors models animator – as a plugin for the IVY Workbench. The plugin uses the NuSMV model checker simulations capabilities, and enables users to explore the formal models interactively.

Resumo

A IVY Workbench é uma ferramenta de modelação e análise de sistemas interativos que tem vindo a ser desenvolvida no Departamento de Informática da Universidade do Minho (http://ivy.di.uminho.pt). Trata-se de uma plataforma desenvolvida maioritariamente em Java, utilizando um mecanismo de plugins. Os plugins existentes incluem um conjunto de editores (em modo texto e gráfico), e de ferramentas de análise do comportamento dos modelos. A experiência de utilização da ferramenta tem, no entanto, demonstrado a necessidade de um animador de modelos que permita efetuar uma primeira validação interativa dos mesmos. Sendo assim, esta dissertação descreve o desenho e implementação do WildAniMAL – um animador de modelos de MAL (Modal Action-Logic) Interactors – como plugin para a IVY Workbench. O plugin usa as capacidades de simulação do model checker NuSMV, e permite aos utilizadores explorar os modelos formais de forma interativa.

Index

Acknowledgements ______ *i*

Abstract ______ *ii*

Resumo ______ *iii*

Index ______ *iv*

Figures ______ *vii*

Acronyms ______ *ix*

Chapter 1 – Introduction ______ *1*

1.1. Goal ______ **2**

1.2. Structure Of The Document ______ **2**

Chapter 2 – Theoretical Background ______ *3*

2.1. Model Checking ______ **3**

2.2. NuSMV ______ **5**

2.3. Finite State Machine ______ **7**

2.4. Binary Decision Diagrams ______ **9**

2.5. MAL Interactors ______ **11**

2.6. SMV Language ______ **12**

2.7. CTL ______ **13**

2.8. Conclusion ______ **14**

Chapter 3 – IVY Workbench ______ *16*

3.1. The IVY Workbench Approach ______ **16**

3.1.1 Creating Models ______ 17

3.1.2 Expressing Properties 19

3.1.3 Verification 20

3.1.4 Trace Analysis 21

3.2. How To Create An IVY Workbench Plugin 25

3.3. Conclusion 28

Chapter 4 – Related Work 29

4.1. CTTE 29

4.2. AniMAL 32

4.3. Conclusion 34

Chapter 5 – WildAniMAL Implementation Approaches 35

5.1. Implementation Approaches 35

5.1.1 Generating a Finite State Machine 36

5.1.2 NuSMV Binary Decision Diagrams 37

5.1.3 NuSMV Simulation Capabilities 37

5.2. NuSMV Interactive Shell 39

5.2.1 Model Reading And Building 39

5.2.2 Simulation Commands 40

5.2.3 Simulation Example 41

5.3. Conclusion 46

Chapter 6 – WildAniMAL Implementation 47

6.1. WildAniMAL's Architecture 47

6.2. WildAniMAL's Source Code Description 53

6.2.1 Animator Package 53

6.2.2 Constraints Package 62

6.2.3 NuSMV Package 64

6.3. Conclusion 68

Chapter 7 – Using WildAniMAL 70

7.1. WildAniMAL's Usage Example 70

7.2. Conclusion 80

Chapter 8 – Conclusions and Future Work 81

8.1. Goal ______ **81**

8.2. Results ______ **81**

8.3. Future Work ______ **83**

References ______ ***84***

Appendix I – Build.xml ______ ***87***

Appendix II – Ipod interactors model ______ ***93***

Figures

Figure 1 Model checking system, adapted from [9], with the IVY Workbench approach. 4

Figure 2 A graph of an extremely basic process in a finite state machine. 8

Figure 3 FSM example of a parser recognizing the world "nice". 9

Figure 4 Diagram for $A \vee \bar{B}C$, taken from [16]. 9

Figure 5 ROBDD for (x1⇔y1)^(x2⇔y2) with variable ordering x1<x2<y1<y2, taken from [17]. 10

Figure 6 IVY Workbench architecture. 17

Figure 7 Model Editor plugin (Graphical). 18

Figure 8 Model Editor plugin (Text). 18

Figure 9 Properties Editor. 19

Figure 10 Behaviour trace. 21

Figure 11 Trace Analysis mechanisms (markers). 21

Figure 12 Tree visual representation. 22

Figure 13 State Based visual representation. 23

Figure 14 Logical States visual representation. 23

Figure 15 Tabular visual representation. 24

Figure 16 Activity Diagram visual representation. 24

Figure 17 IVY Workbench Plugins Framework. 25

Figure 18 CTTE tool, taken from [25]. 30

Figure 19 An example of a task model, taken from [25]. 30

Figure 20 Overview of the CTT notation, taken from [28]. 31

Figure 21 A simple ConcurTaskTree task model simulator, taken from [28]. 32

Figure 22 Thermometer, taken from [29]. 33

Figure 23 Prototype of an air condition control panel, taken from [29]. 34

Figure 24 NuSMV simulation example. 38

Figure 25 Interactors model of a garage gate. 42

Figure 26 State Diagram of gate model. 42

Figure 27 NuSMV specification of the gate model. 43

Figure 28 The result of pick_state –i –a command. 44

Figure 29 The result of simulate –i –a –k 1 command. 44

Figure 30 The result of choosing state 0. 45
Figure 31 State Based diagram showing the simulation path. 45
Figure 32 WildAniMAL plugin architecture as a package diagram. 48
Figure 33 Main package class diagram. 49
Figure 34 NuSMV package class diagram. 50
Figure 35 Constraints package class diagram. 51
Figure 36 Traces package class diagram. 52
Figure 37 Graphical user interface implemented by GUI class. 55
Figure 38 Ipod model creation with Model Editor plugin. 71
Figure 39 Ipod interactor model compilation with Properties Editor plugin. 72
Figure 40 Result of pressing "Get Initial State" in WildAniMAL. 73
Figure 41 Simulation reached a "Too Many States" situation. 74
Figure 42 State Based representation of the trace created in the simulation. 75
Figure 43 Tabular representation of the trace created in the simulation. 75
Figure 44 Log representation of the simulation. 76
Figure 45 Constraints insertion. 77
Figure 46 Result of application of the constraints sent. 77
Figure 47 State Based representation of a trace originated in a simulation. 79

Acronyms

BDD	Binary Decision Diagrams
CMU	Carnegie Mellon University
CTEE	ConcurTaskTrees Environment
CTL	Computacional Tree Logic
CTT	ConcurTaskTrees
CUDD	CU Decision Diagram Package
ETC	Enabled Task Collection
FSM	Finite State Machine
IRST	Istituto per la Ricerca Scientica e Tecnologica
IVY	Interactors VerifYier
LTL	Linear Temporal Logic
MAL	Modal-Action Logic
RBC	Reduced Boolean Circuit
SAT	Satisfiability Problem
SMV	Symbolic Model Verifier
UI	User Interface
UML	Unified Modeling Language

Chapter 1 – Introduction

Developing complex systems will always be a complex endeavour. When developing interactive devices, we are faced with the challenge of understanding not only how the device must be built, but also how it will interact with its users, and how both device and users (the interactive system) will influence each other.

Formal (mathematically rigorous) methods have long been proposed as a means of dealing with complexity. When considering the behaviour of systems, model checking [1] has gained particular popularity. Several approaches to the application of model checking to reason about interactive systems (or interactive devices) have been put forward over the last seventeen years. See, for example, the work in [2], [3], or [4]. However, applying model checking is in itself a complex task. Both systems and Properties to be verified must be expressed in appropriate logics. In order to make model checking of interactive systems feasible, we must provide tools that help in the modelling and analysis process.

The IVY Workbench tool supports the modelling and verification approach put forward in [3]. The main goal of the tool is the detection of potential usability problems early in development of any interactive system. For that, the tool enables the automated inspection of interactive systems models. The tool supports a modelling and analysis cycle where the models are obtained by a modelling process based on an editor, the properties are expressed using a dedicated properties editor, the analysis is performed using the SMV model checker [5] (more specifically NuSMV [6], a reimplementation of that tool, that is available at http://nusmv.fbk.eu), and the counter-examples visualized using a dedicated traces visualiser. The tool has been applied to the analysis of different devices, from control panels in the automotive industry [7], to medical devices such as infusion pumps [8]. While model checking through NuSMV, enables a thorough analysis of all possible behaviours of a model, the continuous use of the tool has highlighted the need for a lighter weight approach to the initial validation of the models. In fact, experience has shown that before the analysis of a given design begins, there usually happens a first phase of model validation, where the interest is in establishing that the model behaves as expected. Experience also shows that doing this through model checking becomes cumbersome. What is needed is the possibility of interactively explore the behaviour of the models: manually trigger events and observe how the system evolves. Hence the need was identified of developing a component aiming at assisting the modelling and analysis process, by providing functionalities to simulate and validate the model being created: WildAniMAL (Watch It vaLiDation Animator for MAL).

1.1. Goal

The goal of this work is to develop a new plugin – WildAniMAL – for the IVY Workbench tool, supporting the animation of MAL interactor models. In order to implement it, three possibilities will be studied:

a) representing a MAL interactors model as a Finite State Machine (FSM) and use that to drive the animation;

b) use the BDD (Binary Decision Diagrams) representation of the MAL interactors model, created by the NuSMV model checker to perform the animation;

c) use the NuSMV model checker simulations commands, available on its interactive mode, to perform the animation.

1.2. Structure Of The Document

This first chapter has presented the motivation and goals of the work. The remaining of the dissertation is structured as follows. Chapter 2 introduces the main concepts needed to understand the work. Chapter 3 introduces the IVY workbench tool. Chapter 4 describes some related tools. Chapter 5 discusses alternatives to implementing the WildAniMAL plugin, and chapter 6 the implementation produced. Chapter 7 describes an usage example. The dissertation ends, in Chapter 8, with a discussion of results and ideas for further work.

Chapter 2 – Theoretical Background

This chapter presents the theoretical background needed to explain the WildAniMAL implementation and all the concepts related to its use, and the use of the IVY Workbench, in which it will be integrated.

Section 2.1 presents Model Checking the technology used by the IVY Workbench to perform verification. Section 2.2 presents NuSMV that is the model checker used in IVY Workbench, and therefore also used to implement WildAniMAL's functionalities. Section 2.3 presents Finite State Machines, a mathematical model of computation, and also a state's representation, widely used to describe computer programs. Section 2.4 presents Binary Decision Diagrams, the data structure used to represent a Boolean function. Two representations used in NuSMV as internal representations. Section 2.5 presents the MAL interactors language used to create models that will be simulate in the WildAniMAL plugin, and Section 2.6 presents the SMV language, the language into which MAL interactors models are compiled for verification and (now) animation.

Finally, CTL is presented, that is a temporal logic that is used to express properties over the interactors model. These properties can be verified using NuSMV model checker.

2.1. Model Checking

Clarke [9] formally describes the Model Checking problem as:

Let M be a Kripke structure (i.e., state-transition graph). Let f be a formula of temporal logic (i.e., the specification). Find all states s of M such that M; s |= f.

That is, given a state-transition graph and a specification, we want to find all states in M that satisfy the specification.

The structure of a typical Model Checking system, as Clarke defined it [9], is described in Figure 1. There the two mains components of a model checking system are presented:

- A **preprocessor** that extracts a state transition graph from a program or circuit;
- A **model checker,** that is, an engine that takes the state transition graph and a temporal formula and determines whether the formula is true or not (in the case of the IVY Workbench this is NuSMV).

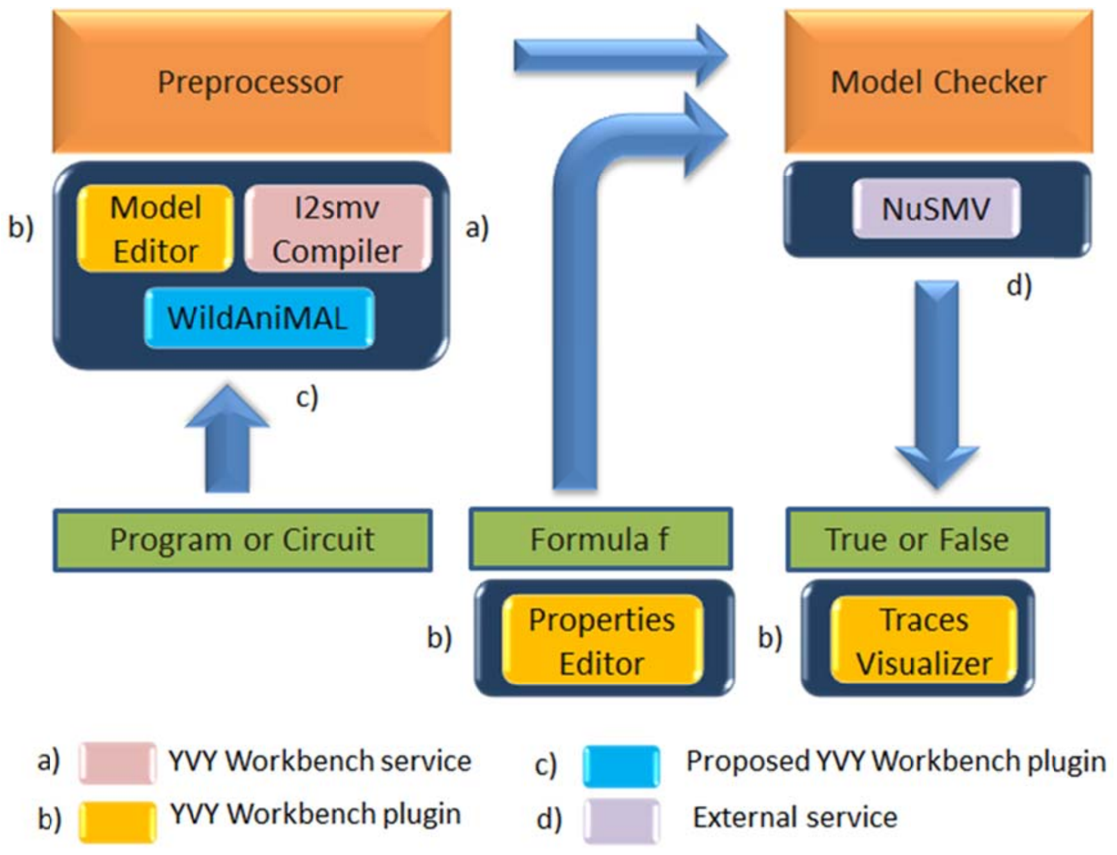


Figure 1 Model checking system, adapted from [9], with the IVY Workbench approach.

Figure 1 includes IVY Workbench's plugins and services that help in the several steps of the model checking process. The IVY Workbench approach to Model Checking is presented in Section 3.1.

There are other verification techniques other than Model Checking, such as Automated Theorem proving or Proof Checking. Therefore is useful to present the advantages that Model Checking has when compared to them. Some of these advantages are:

- It provides **counterexamples**. In a model checker, a counterexample (an execution trace) is produced to show why a specification does not hold. This is a great advantage because counterexamples are great to debug complex systems. Some people use Model Checking just for this feature;
- It uses **Temporal Logics** that can easily express properties for proving over the behaviour of modelled systems. One example of these Temporal Logics is CTL, which is described in Section 2.7. CTL is used in the IVY Workbench tool.

In the opposite side there are also some disadvantages and one of the major ones is **State Explosion.** In [3] the authors describe this problem as related to the size of the finite state machine (this concept will be described in Section 2.3) needed to specify a given system. A specification can generate state spaces so immense that it becomes impossible to analyse the entire state space. To attenuate this problem, Symbolic Model Checking was developed. When the traversal of the state space is done considering large sets of states at a time, and is based on representations of states sets and transition relations as formulas, binary decision diagrams or other related data structures, the model-checking method is considered **Symbolic**. With that technique state spaces as large as 10^{20} may be analysed [10]. NuSMV is a model checker that uses that method and will be described in the following Section.

2.2. NuSMV

NUSMV is a symbolic model checker that was first presented in [6] and [11]. It is the result of a joint project between Carnegie Mellon University (CMU) and Istituto per la Ricerca Scientica e Tecnologica (IRST) and is the final product of an effort of reengineering, reimplementation and extension of CMU SMV, the original BDD-based model checker developed at CMU [5].

Over the years NuSMV had several contributions that improved it with more functionality, as can be seen in its official site[1]. Now it combines a BDD-based model checking component that

[1] http://nusmv.fbk.eu/ Last visited in 10-28-2012.

exploits the CUDD[2] library developed by Fabio Somenzi at Colorado University, and a SAT-based model checking component that includes an RBC-based Bounded Model Checker, which can be connected to the Minisat SAT Solver[3] and/or to the ZChaff SAT Solver[4]. The University of Genova has contributed SIM, a state-of-the-art SAT solver used until version 2.5.0, and the RBC package used in the Bounded Model Checking algorithms.

In [12] we can see the current main functionalities that it provides:

- allows for the representation of synchronous and asynchronous finite state systems;
- allows for the analysis of specifications expressed in Computational Tree Logic (CTL) and Linear Temporal Logic (LTL), using BDD-based and SAT-based model checking techniques.
- provides Heuristics for achieving efficiency and partially controlling the state explosion;
- provides a textual (interactive mode) and a batch mode interface to interact with its users.

NuSMV, as a model checker, can verify properties of a finite system and for that to be possible a model of the system (in fact, in terms of model checking, a *specification* of the system) has to be created. NuSMV uses the SMV Language (see Section 2.7) to define the specifications used as input. In [13] it is described how this language can be used to allow for the description of Finite State Machines (FSM) which can be completely synchronous or completely asynchronous. More specifically the SMV Language is used to describe the transition relation of the FSM that describes the valid evolutions of the state of the FSM.

In the IVY Workbench, that model is created in the MAL Interactors language (see Section 2.5), that is easier to learn and can be compiled (using the IVY Workbench i2smv service) into a SMV specification. After having a SMV specification, NuSMV can verify that a model satisfies a set of desired properties specified by the user. For that, it uses two Temporal Logics: CTL or LTL.

One useful feature that NuSMV has is that it provides the user with the possibility of simulating a NuSMV specification. As stated in [13], this way the user can explore the possible

[2] http://vlsi.colorado.edu/~fabio/CUDD/ Last visited in 10-28-2012.

[3] http://minisat.se/ Last visited in 10-28-2012.

[4] http://www.princeton.edu/~chaff/zchaff.html Last visited in 10-28-2012.

executions (traces) of the NUSMV specification. In this way, the user can check the specification correctness, before actually engaging in the verification of properties. An example of the use of this feature can be seen in Section 5.2.3.

2.3. Finite State Machine

When modelling the behaviour of systems, S*tate Machines* are one of the oldest and best ways known. They define the state of a system at a particular point in time and characterize its behaviour based on that state.

If we want to model and design software systems we can apply the State Machines method by identifying the states the system can be in, which inputs or events trigger state transitions, and what system behaviour is expected in each state. The execution of the software can be seen as a sequence of transitions that move the system through its various states.

The characteristics of a system that enable it to be modelled as a Finite State Machine (FSM) are [14]:

- The system must be specifiable as a finite set of states;
- The system must have a finite set of inputs and/or events that can trigger transitions between states;
- The behaviour of the system at a given point in time depends upon the current state and the input or event that occur at that time only;
- For each state the system may be in, behaviour is defined for each possible input or event;
- The system has a particular initial state.

Figure 2 illustrates the main concepts that a Finite State Machine is known for.

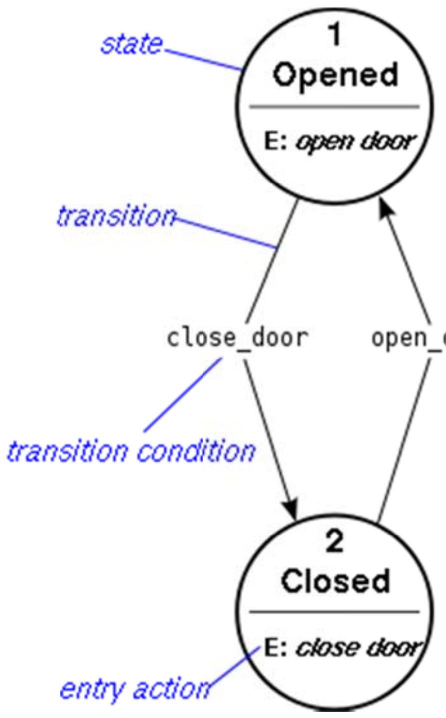


Figure 2 A graph of an extremely basic process in a finite state machine.

The conceptual definition of the FSM can be expressed more formally (in this case mathematically) [15] as a quintuple $(\Sigma, S, s_0, \delta, F)$, where:

- Σ is the input alphabet (a finite, non-empty set of symbols).
- S is a finite, non-empty set of states.
- s_0 is an initial state, an element of S.
- δ is the state-transition function: $\delta : S \times \Sigma \rightarrow S$ (for a nondeterministic finite automaton it becomes $\delta : S \times \Sigma \rightarrow \mathcal{P}(S)$, i.e., δ returns a set of states).
- F is the set of final states, a (possibly empty) subset of S.

An example of the graphical representation of a FSM is presented in Figure 3.

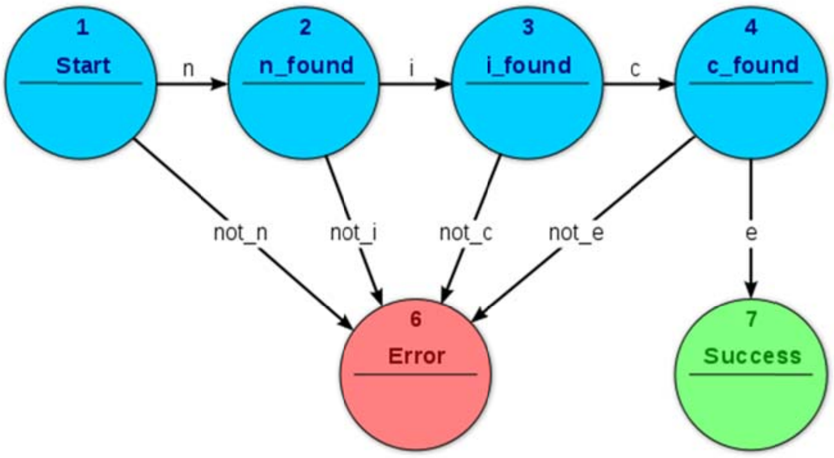


Figure 3 FSM example of a parser recognizing the world "nice".

2.4. Binary Decision Diagrams

Binary Decision Diagrams (BDD) [16] can be defined as a data structure that is used to represent a Boolean function (see Figure 4 for an example). We can also say it a compressed representation of sets or relations.

Andersen [17] provides a formal definition of a BDD. He defines it as a rooted, directed acyclic graph with:

- one or two terminal nodes of out-degree zero labeled 0 or 1, and
- a set of variable nodes u of out-degree two. The two outgoing edges are given by two functions ***low(u)*** and ***high(u)***. A variable ***var(u)*** is associated with each variable node.

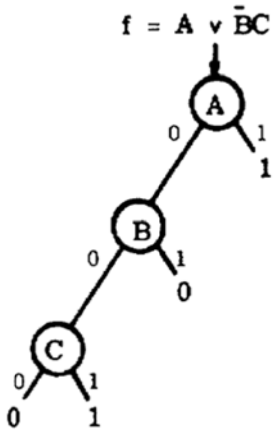


Figure 4 Diagram for $A \vee \bar{B}C$, taken from [16].

When mentioning BDDs it is important to mention if they are ordered or not. Andersen defines a BDD as Ordered (OBDD) if on all paths through the graph the variables respect a given linear order x1 < x2 < … < xn.

An **(O)BDD** is Reduced **(R(O)BDD) (**see Figure 5) if

- (uniqueness) no two distinct nodes u and v have the same variable name and low- and high-successor, i.e.,

  var(u) = var(v); low(u) = low(v); high(u) = high(v) implies u = v;

and

- (non-redundant tests) no variable node u has identical low- and high-successor, i.e. low(u) ≠ high(u).

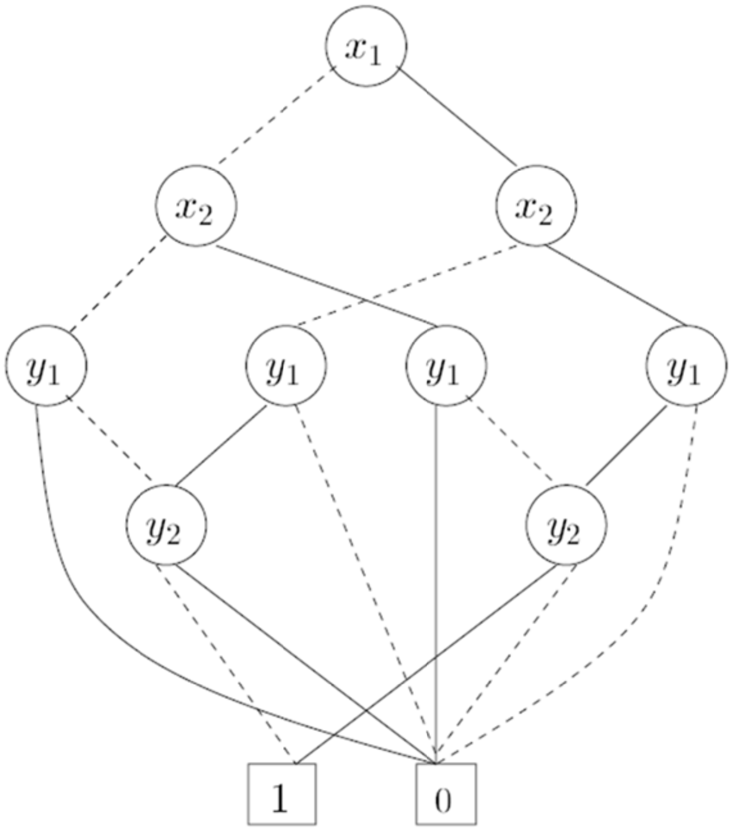


Figure 5 ROBDD for (x1⇔y1)^(x2⇔y2) with variable ordering x1<x2<y1<y2, taken from [17].

In most cases, when BDDs are referred to, it is implied that we are referring to Reduced Ordered Binary Decision Diagrams.

Bryant [18] studied the BDD potential for being used to create efficient algorithms. He introduced a fixed variable ordering (for canonical representation) and shared sub-graphs (for compression). After that he extended the sharing concept to several BDDs, i.e. one sub-graph by several BDDs and, doing that, he defined the data structure Shared Reduced Ordered Binary Decision Diagram. That new structure is normally what people have in mind when mentioning BDDs.

The NuSMV model checker uses BDDs, because they are very efficient and can be used to create efficient algorithms, as shown in [18]. The efficiency of algorithms is important in the area of Model Checking, and because of that the use of BDDs by NuSMV was an obvious choice.

2.5. MAL Interactors

MAL interactors follow from the notion of interactor put forward in [19]: an object with the capability of rendering part of its state to some presentation medium. A MAL interactor is defined by:

- a set of typed attributes that define the interactor's state;
- a set of actions that define operations on the set of attributes;
- a set of axioms written in MAL [20] that define the semantics of the actions in terms of their effect on interactor's state.

The mapping of the interactor's state to the presentation medium is accomplished by decorating the attributes with modality annotations. MAL axioms define how the interactor's state changes in response to actions being executed on the interactor. In [3] the axioms are defined in five types. In the syntax of each type, the notation *prop(expr1,..,expr$_n$)* is used to denote a formula on expressions *expr1 to expr$_n$* using propositional operators only. Also, the names **a1** to **a$_n$** denote interactor attributes and **ac** denotes an action. The five types are:

- **Invariants** – these are formulae that do not involve any kind of action or (reference) event (i.e. simple propositional formulae). They must hold for all states of the interactor;
    - o Syntax: prop(a1,..,a$_n$).

- **Initialisation axioms –** these are formulae that involve the reference event ([]). They define the initial state of the interactor;
    - Syntax: [] prop(a1,..,a_n).
- **Modal axioms –** these are formulae involving the modal operator. They define the effect of actions in the state of the interactor;
    - Syntax: prop([ac] a1,..,[ac]$a_{g,}$ $a_{h,}$..,a_n).
- **Permission axioms –** these are deontic formulae involving the use of **per.** They define specific conditions for actions to be permitted to happen;
    - Syntax: per(ac) → prop(a1,..,a_n)
- **Obligation axioms –** these are deontic formulae involving the use of **obl.** They define the conditions under which actions become obligatory.
    - Syntax: prop(a1,..,a_n) → obl(ac)

2.6. SMV Language

The SMV language will be used as an intermediate representation of the MAL interactors model. Therefore an explanation of the main aspects of the SMV language is needed. The following description of the language is adapted from [3] and [12].

An SMV specification is defined as a collection of modules. Each module defines a Finite State Machine (FSM) and consists of a number of state variables, input variables and frozen variables, a set of rules that define how the module makes a transition from one state to the next and Fairness conditions that describe constraints on the valid paths of the execution of the FSM.

A state model is defined as an assignment of values to a set of state and frozen variables. State variables can change their values throughout the evolution of the FSM. Frozen variables cannot, as they retain their initial value, and that is what distinguishes the two. Input variables are used to label transitions of the Finite State Machine.

An example of an SMV specification is the following:

```
MODULE main
  – attributes
  VAR
    currentSong: 0..5;
    lastDisplay: {MainMenu, Music, Playing, OFF};
    playbackState: {playing, paused, stoped};
    display: {MainMenu, Music, Playing, OFF};

  – actions
  VAR
    action: {pause, longPlay, play, nil};

  – axioms
  INIT display = OFF
  INIT playbackState = stoped
  INIT lastDisplay = MainMenu
  INIT currentSong = 0

  TRANS next(action)=pause -> playbackState = playing
  TRANS next(action)=play -> playbackState = stoped | playbackState =
paused
  INIT action = nil
```

To create a SMV specification the following list of declarations is used:

- **VAR** → allows the declaration of *state variables*;
- **IVAR** → allows the declaration of *input variables*;
- **FROZENVAR** → allows the declaration of *frozen variables*;
- **INIT** → allows the definition of the initial states of the model;
- **INVAR** → allows the specification of invariants over the state.
- **TRANS** → allows the definition of the behaviour of the model. In these definitions, the operator next is used to refer to the next state;
- **FAIRNESS** → allows the declaration of fairness constraints, that is, conditions that must hold infinitely often over the execution paths of the model.

2.7. CTL

When reasoning about the behaviour of a system is needed, CTL can be used to express the properties for that purpose. The detailed description of CTL and its formal description are available in [21]. A more compact description of its operators is given here. As other similar languages CTL provides propositional logic connectives but it also allows for operators over the computation paths that can be reached from a state.

- **A** - for all paths (universal quantifier over paths);
- **E** - for some path (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 provide for an expressive language because combining them it is possible to express important concepts such us:

- **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).

2.8. Conclusion

This chapter presented all the theoretical background needed to explain the WildAniMAL implementation and the tool in which it is integrated – the IVY Workbench.

Section 2.1 presented Model Checking that is the area in which this work is framed, and

Section 2.2 presented NuSMV that is the model checker used widely in IVY Workbench, and which will also be used in the WildAniMAL plugin.

Sections 2.3 and 2.4 presented Finite State Machine and Binary Decision Diagrams, two representations studied as possible approaches for WildAniMAL's internal data structure. BDD is used also in NuSMV as one of its data structures.

Section 2.5 presented the MAL interactors language used to create interactor models, and Section 2.6 presented the SMV language that will be used as an intermediate representation of the first one, because it is the language NuSMV uses.

Finally, CTL was presented. This language is used to express properties over the interactor models, created with the MAL interactors language, and compiled to a NuSMV specification.

Chapter 3 – IVY Workbench

This chapter presents the IVY Workbench tool that supports the modelling and analysis of interactive systems. It is a plugins platform (developed in Java) that includes a set of editors and tools to analyse the models' behaviour.

Section 3.1 presents the IVY Workbench approach, relating to model checking, that consists on creating a MAL model, expressing properties over it, making a verification with the help of the NuSMV model checker and analysing its results.

Section 3.2 describes how to create a new plugin for the IVY Workbench, as this is useful to know how to implement the proposed WildAniMAL plugin.

3.1. The IVY Workbench Approach

In [3] and [4] an approach to the application of model checking to the analysis of interactive systems is put forward. The approach is based in the development of models of the interactive device, and in their verification trough model checking against properties that encode assumptions about the usages of the device.

Figure 6 shows the architecture of the tool, added with the proposed WildAniMAL plugin. As it can be seen, the tool consists on a number of plugins, and uses NuSMV as the verification engine. In this section the different plugins are described (except WildAniMAL, which will be discussed later, see Chapter 7).

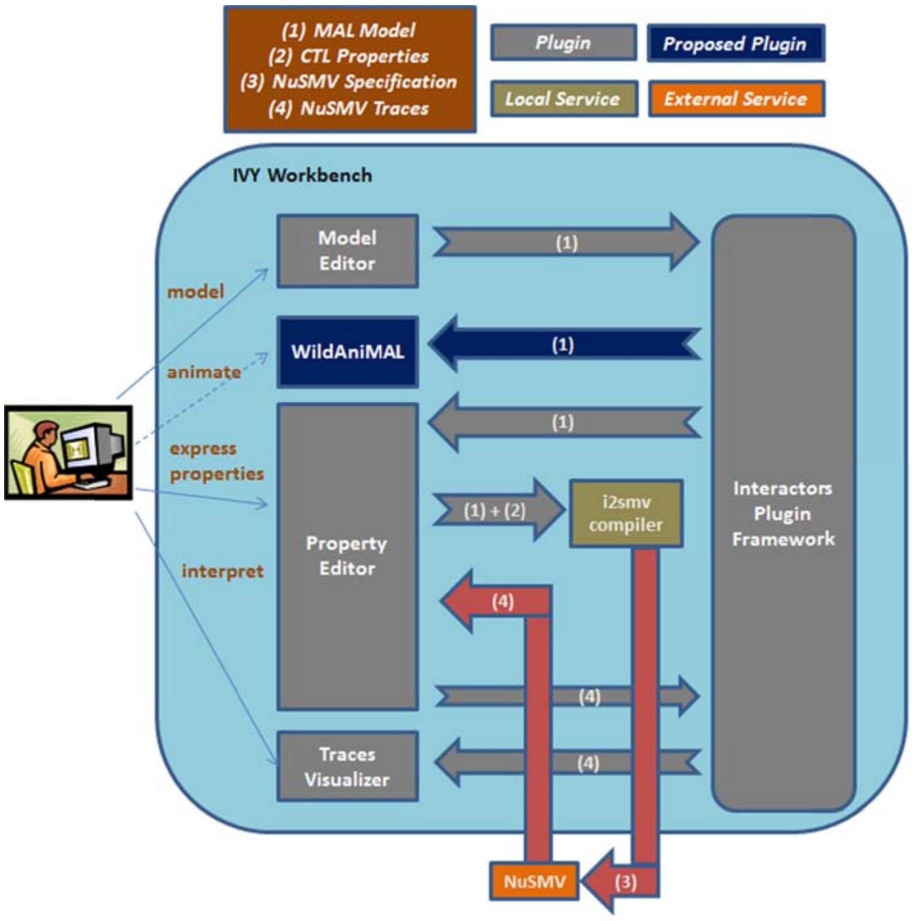
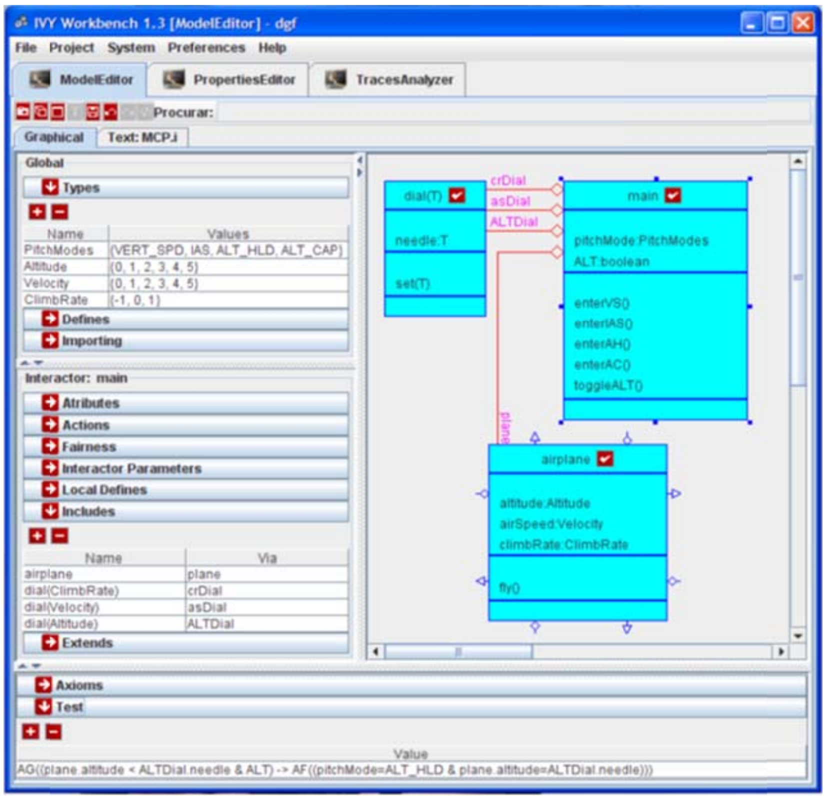


Figure 6 IVY Workbench architecture.

3.1.1 Creating Models

A MAL model is constructed composing interactors in a hierarchical form. The process of MAL model creation is supported by the Model Editor plugin of the tool. The plugin has two modes: Graphical (see Figure 7) that uses a notation similar to UML class diagrams (described in [22]) and Text (see Figure 8) that provides code completion facilities.

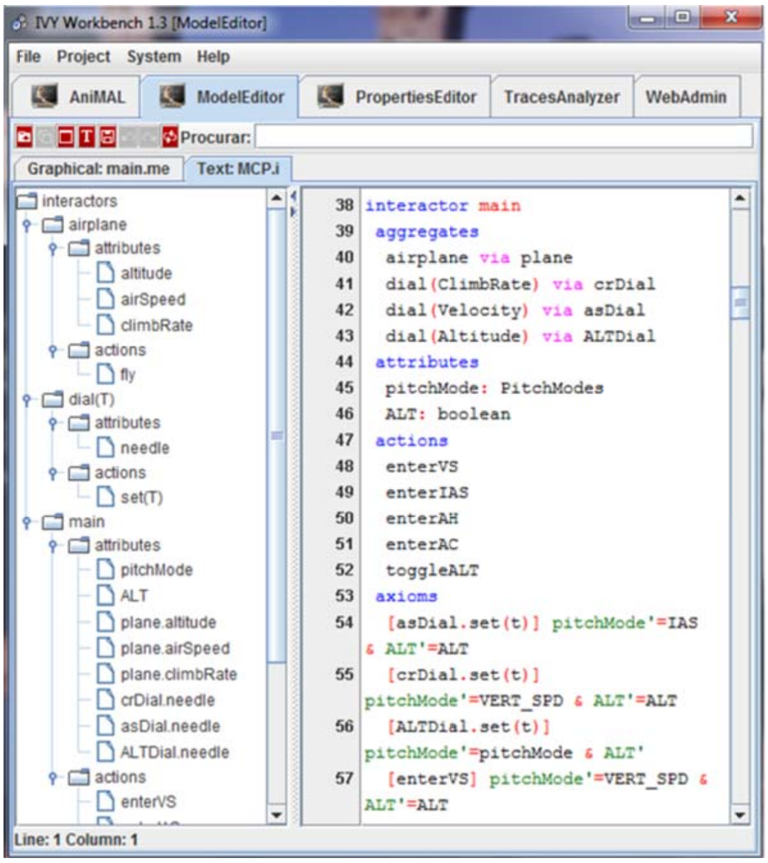


Name	Values
PitchModes	(VERT_SPD, IAS, ALT_HLD, ALT_CAP)
Altitude	(0, 1, 2, 3, 4, 5)
Velocity	(0, 1, 2, 3, 4, 5)
ClimbRate	(-1, 0, 1)

Name	Via
airplane	plane
dial(ClimbRate)	crDial
dial(Velocity)	asDial
dial(Altitude)	ALTDial

Value
AG((plane.altitude < ALTDial.needle & ALT) -> AF((pitchMode=ALT_HLD & plane.altitude=ALTDial.needle)))

Figure 7 Model Editor plugin (Graphical).



```
interactors
  airplane
    attributes
      altitude
      airSpeed
      climbRate
    actions
      fly
  dial(T)
    attributes
      needle
    actions
      set(T)
  main
    attributes
      pitchMode
      ALT
      plane.altitude
      plane.airSpeed
      plane.climbRate
      crDial.needle
      asDial.needle
      ALTDial.needle
    actions
      enterVS
```

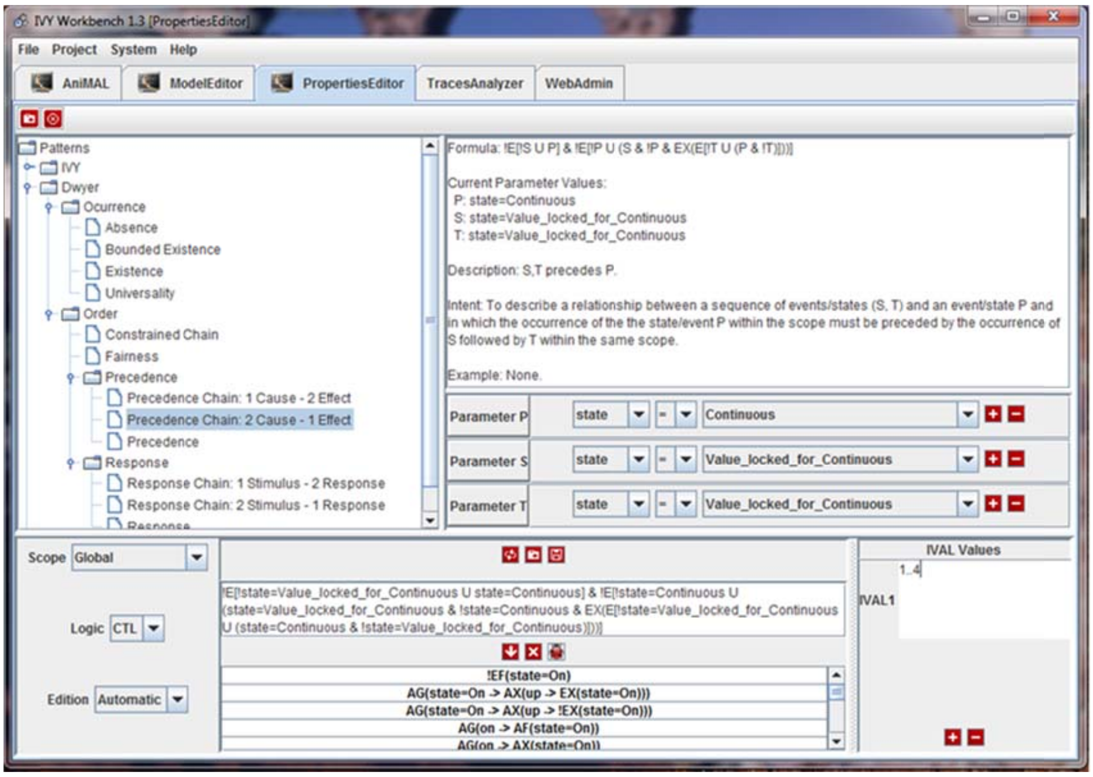
```
interactor main
 aggregates
  airplane via plane
  dial(ClimbRate) via crDial
  dial(Velocity) via asDial
  dial(Altitude) via ALTDial
 attributes
  pitchMode: PitchModes
  ALT: boolean
 actions
  enterVS
  enterIAS
  enterAH
  enterAC
  toggleALT
 axioms
  [asDial.set(t)] pitchMode'=IAS & ALT'=ALT
  [crDial.set(t)] pitchMode'=VERT_SPD & ALT'=ALT
  [ALTDial.set(t)] pitchMode'=pitchMode & ALT'
  [enterVS] pitchMode'=VERT_SPD & ALT'=ALT
```

Figure 8 Model Editor plugin (Text).

3.1.2 Expressing Properties

The properties for verification are written in CTL [1]. Properties are written that express assumptions about the expected behaviour of the device.

The process of expressing properties is supported by the Properties Editor plugin of the IVY Workbench tool (see Figure 9). The plugin is based on sets of patterns that capture usual properties typically verified of any interactive system. The patterns used by the IVY workbench are described in [23]. Each pattern describes its intent, provides a practical example and has some parameters. After choosing the most suited pattern for the property he or she wants to express, the user of the tool has only to define the values of the parameters of the pattern. For doing that, the tool has an assisted mode, in which the user selects attributes and actions from the model for the parameters of the pattern.



Formula: !E[!S U P] & !E[!P U (S & !P & EX(E[!T U (P & !T)]))]

Current Parameter Values:
P: state=Continuous
S: state=Value_locked_for_Continuous
T: state=Value_locked_for_Continuous

Description: S,T precedes P.

Intent: To describe a relationship between a sequence of events/states (S, T) and an event/state P and in which the occurrence of the the state/event P within the scope must be preceded by the occurrence of S followed by T within the same scope.

Example: None.

Parameter P: state = Continuous
Parameter S: state = Value_locked_for_Continuous
Parameter T: state = Value_locked_for_Continuous

Scope: Global
Logic: CTL
Edition: Automatic

!E[!state=Value_locked_for_Continuous U state=Continuous] & !E[!state=Continuous U (state=Value_locked_for_Continuous & !state=Continuous & EX(E[!state=Value_locked_for_Continuous U (state=Continuous & !state=Value_locked_for_Continuous)]))]

```
!EF(state=On)
AG(state=On -> AX(up -> EX(state=On)))
AG(state=On -> AX(up -> !EX(state=On)))
AG(on -> AF(state=On))
AG(on -> AX(state=On))
```

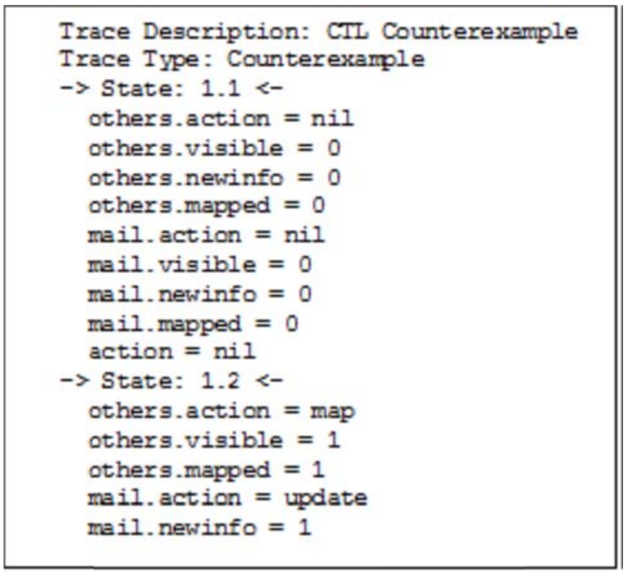
Figure 9 Properties Editor.

3.1.3 Verification

The verification step is performed by the NuSMV model checker. To make the verification possible, MAL interactor models are compiled to the SMV language. A detailed description of the verification approach is out of the scope of this dissertation. For the discussion that follows what is important is that, when a given property is not verified, NuSMV tries to provide a behaviour of the model (a trace) that demonstrates the falseness of the property in question. These traces (see Figure 10 for an extract) consist of a sequence of states of the model that violates the property under scrutiny.

Because of limitations on the SMV input language, when compared to MAL interactors, the compilation step mentioned above introduces a series of auxiliary variables in the model. This means that the trace is not at the same level of abstraction as the interactor model being verified. One aspect were this is particularly evident is the treatment of actions. Because SMV models do not have an explicit notion of action, the compilation process introduces a special attribute - action - used for modelling, in each state, which action has just occurred.

Another aspect that deserves mention is a mismatch in the execution models of both languages. At MAL interactors level, the actions of different interactors can happen in an asynchronous way. Thus, an interactor can execute one action while the others remain inactive. At the SMV level, however, the transitions occur in a synchronous way. This means that when a module performs a transition all modules in the model must also perform a transition. To model asynchronous state transitions, it becomes necessary to introduce a special action nil that at the MAL interactors level (what we will call the logical level from now on) corresponds to nothing happening, while at SMV level (what we will call physical level from now on) represents a state transition (to a state with the same attributes values i.e. to the same logical state). This way, the SMV module corresponding to an interactor can perform a state transition associated to a given action, while the others execute the action associated to nil (that is, maintain the state).



```
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  others.action = nil
  others.visible = 0
  others.newinfo = 0
  others.mapped = 0
  mail.action = nil
  mail.visible = 0
  mail.newinfo = 0
  mail.mapped = 0
  action = nil
-> State: 1.2 <-
  others.action = map
  others.visible = 1
  others.mapped = 1
  mail.action = update
  mail.newinfo = 1
```

Figure 10 Behaviour trace.

3.1.4 Trace Analysis

The traces produced by the verification process do, as we can expect, mention the variables and states existing at the SMV code level (some of which were introduced at the compilation step, as mentioned above). Thus, it is necessary to revert the compilation process so that the analysis of the trace's meaning can be performed at the level of abstraction of the original interactor model. A typical example would be elimination of attribute action, replacing it by some appropriate representation of the notion of action.

Counter-example traces can, however, reach sizes in the order of the hundreds of states, depending on the complexity of the model. Hence their analysis can become time consuming and complex. The Traces Visualizer plugin, of the IVY Workbench tool, aims at facilitating this analysis step, helping in determining what the problem is that is being pointed out by the trace, and in discovering possible solutions to it. To achieve these goals the plugin resorts to visual representations and trace analysis mechanisms (markers) that can be seen in Figure 11.

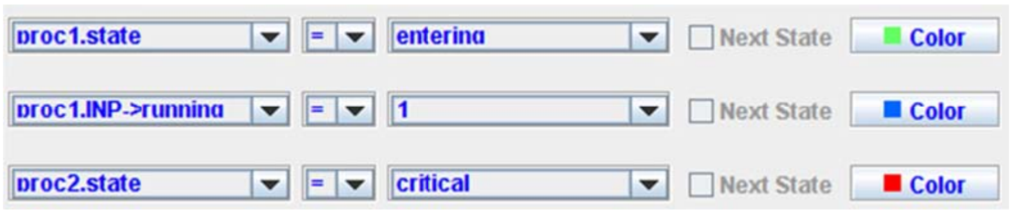


Figure 11 Trace Analysis mechanisms (markers).

The available visual representations are fully described in [24] and are the following:

- Trace - the original textual representation produced by SMV;
- Tree (see Figure 12) - tree representation of the trace states;
- State Based (see Figure 13) - graphical representation of the trace states;
- Logical States (see Figure 14) - representation similar to the previous one in which the trace states are pre-processed to eliminate artificial states introduced by the compilation process;
- Tabular (see Figure 15) - tabular representation similar to the one existing in the SMV of Cadence Labs;
- Activity Diagram (see Figure 16) - representation focused on actions that resorts to Activity diagrams (following the notation of UML 2.0 described in [22].

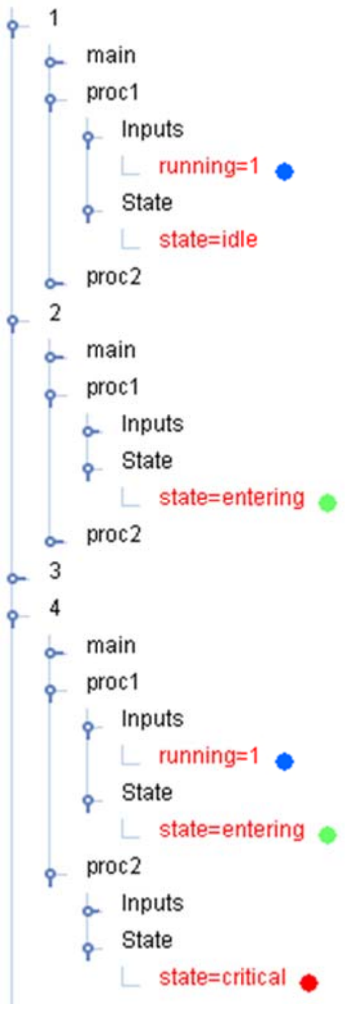


Figure 12 Tree visual representation.

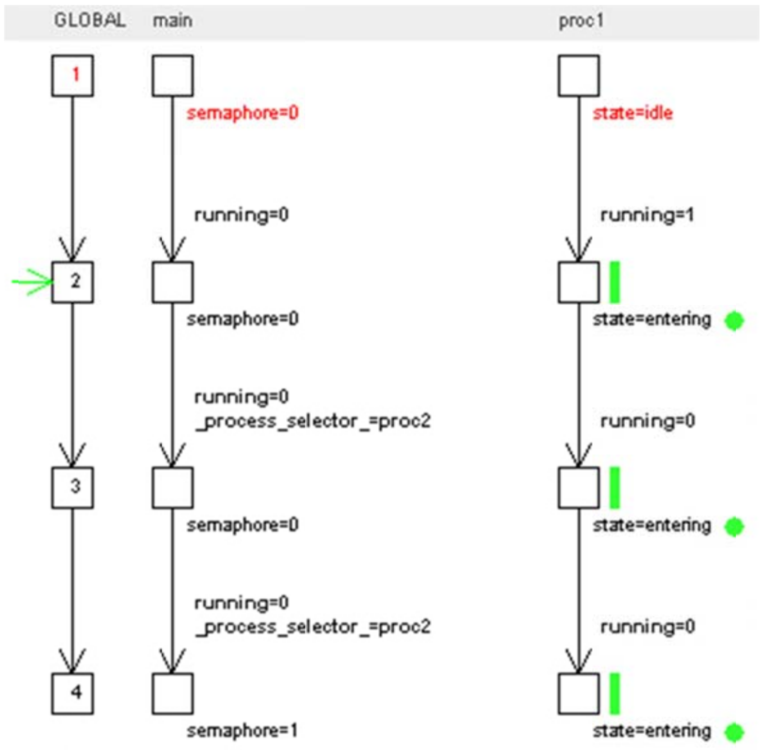


Figure 13 State Based visual representation.

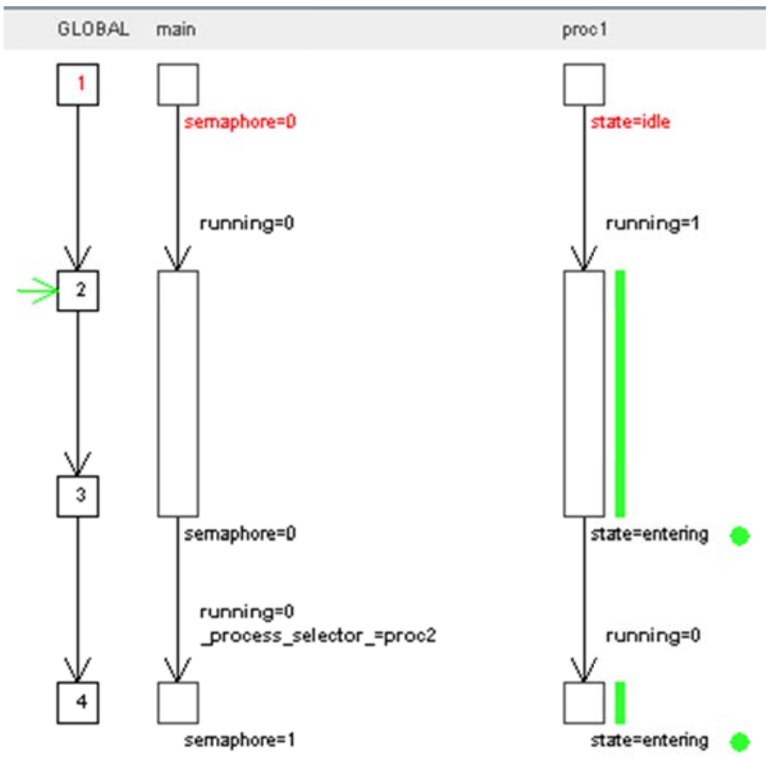
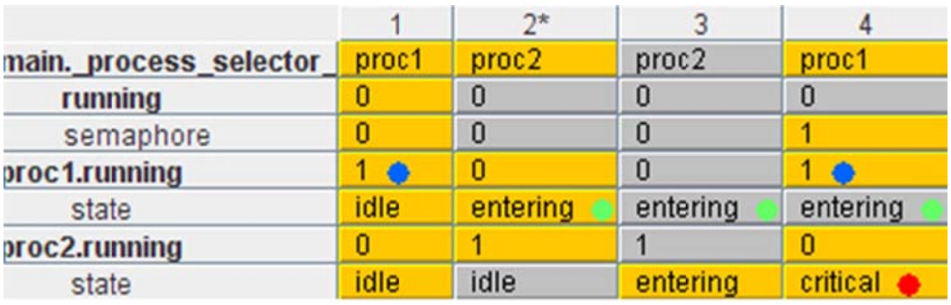


Figure 14 Logical States visual representation.



	1	2*	3	4
main._process_selector_	proc1	proc2	proc2	proc1
running	0	0	0	0
semaphore	0	0	0	1
proc1.running	1 ●	0	0	1 ●
state	idle	entering ●	entering ●	entering ●
proc2.running	0	1	1	0
state	idle	idle	entering	critical ●

Figure 15 Tabular visual representation.

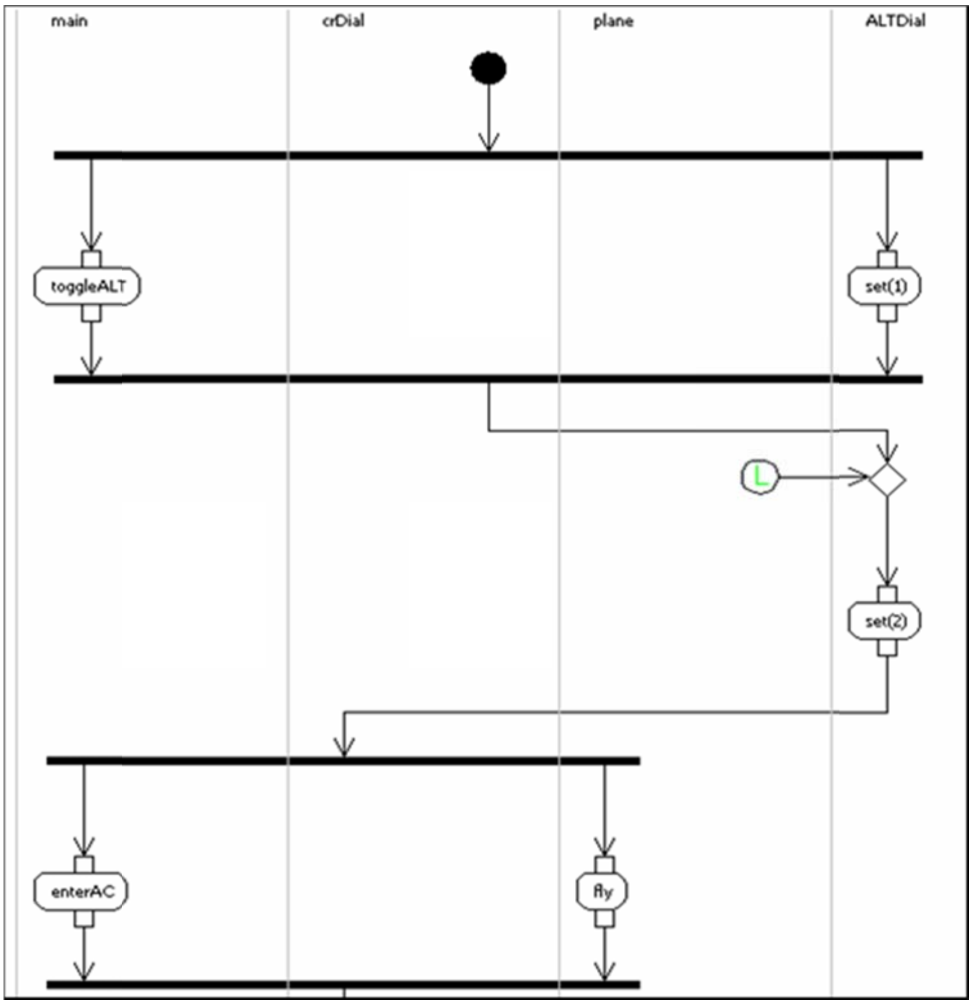


Figure 16 Activity Diagram visual representation.

3.2. How To Create An IVY Workbench Plugin

The IVY Workbench is a modular tool based on plugins. A plugin is integrated into the tool by implementing an interface that defines the methods needed for integration purposes. For simplification purposes a plugin can also be called a tool. Each tool will be placed on a tabbed pane so that the user can select between all the tools loaded into the framework. For example, in Figure 17, the **Model Editor**, **Properties Editor** and **Traces Analyzer** tools are represented.

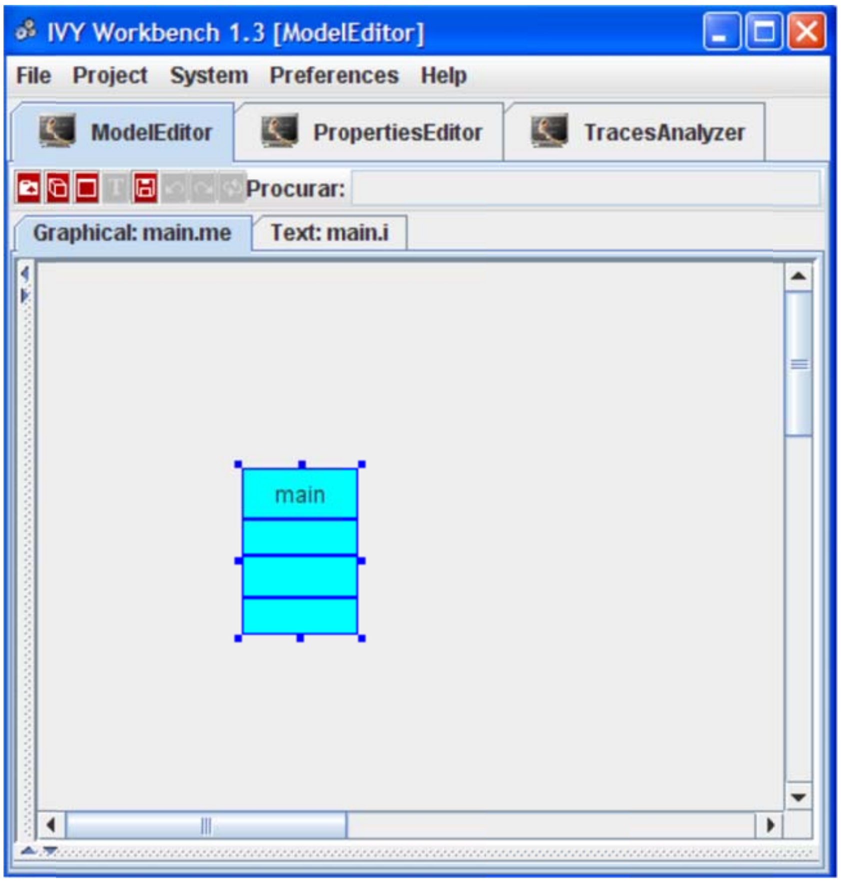


Figure 17 IVY Workbench Plugins Framework.

The interface methods that must be implemented to construct a tool are the following:

- <u>**public void init(IServer coreServer, IToolProperties prop) throws Exception**</u> → this method initializes the Tool. This method is called once in the life cycle of the tool. It receives a parameter that is the server used to handle the processing and also a parameter that contains a reference to the properties of this tool.

- **public void initGUI(JFrame main, JComponent rootContainer) →** this method is used to initialize the Graphic User Interface for the tool. This method is also called once only in the life cycle of the tool. It receives the main JFrame of the IVY Workbench tool and also receives the container in which the tool graphic component can be added.

- **public void gainFocus() →** this method is to be invoked whenever the tool is selected in the main tabbed pane of IVY Workbench tool. With this method we can control what we want to do each time the tool gains control. For example if some global data is changed by others tools then the current tool can also change its state (by changing graphical elements or internal data) to reflect them.

- **public void loseFocus() →** this method is used whenever the tool loses the control (is de-selected). With this method we can control what we want to do when the user switches to other tool. For example the current tool can put some data in a global area (common to all tools) so that the other tools can query if some global data is available, and if so reflect some changes on their own states, by changing graphical elements or internal data.

- **public boolean needsSaving() →** this method is used to tell if the tool needs to save its data when a project is being saved.

- **public boolean needsFocus(int event) →** this method is used to return the status related to focus. It receives a parameter that is the event by which the tool needs focus. The event codes are the following:
    - int EVENT_OPEN_PROJECT = 0;
    - int EVENT_NEW_PROJECT = 1;
    - int EVENT_SAVE_PROJECT = 2;
    - int EVENT_CLOSE_PROJECT = 3;
    - int EVENT_EXIT_PROGRAM = 0.

- **public void newProject(IProjectProperties proj) →** this method is invoked whenever the main application creates a new project. It receives the project properties (name, project working directory, author, etc.).

- **public void openProject(IProjectProperties proj, String[] files) →** this method is invoked whenever the main application opens a project. It receives the project properties and also the paths of the folders belonging to this tool.

- **public String[] saveProject(IProjectProperties proj) →** this method is invoked whenever the user wants to save the current project. It will be up to the tool to save its own data files. This method receives the project properties as a parameter and returns the paths of the folders belonging to this tool.

- **public void closeProject(IProjectProperties proj) →** this method is invoked whenever the IVY user wants to close the current project. It receives the project properties.

- **public void exit() →** this method is invoked whenever the user exits the IVY Workbench.

The configuration file **plugin.xml** is needed to properly configure the tool. The following text explains how to fill the data fields of this configuration file.

The structure of the **XML** file is the following:

```
<?xml version="1.0" ?>
<!DOCTYPE plugin PUBLIC "-//JPF//Java Plug-in Manifest 0.4"
"http://jpf.sourceforge.net/plugin_0_4.dtd">

<plugin id=’tool name’ version=’tool version’ >
<requires>
<import plugin-id="CoreSystem"/>
</requires>
```

```
<runtime>
<library id='tool library name' path='tool jar filename' type="code">
<doc caption="API documentation">
<doc-ref path="api/index.html" caption="javadoc"/>
</doc>
</library>
<library type="resources" path="icons/" id="icons"/>
</runtime>

<extension plugin-id="CoreSystem" point-id="Tool" id='tool name'>
<parameter id="class" value='tool java main class name' />
<parameter id="name" value='tool name' />
<parameter id="description" value='tool description' />
<parameter id="icon" value='tool icon filename' />
</extension>
</plugin>
```

The values between quotes have to be replaced to fill the configuration file. For example, to make the configuration file of **Model Editor** tool the values are instantiated in this way:

'tool name'= "ModelEditor"
'tool version'= "0.0.1"
'tool library name'= "Model Editor"
'tool jar filename'= "ModelEditor.jar"
'tool java main class name'= "Editor.Editor"
'tool description'= "Model Editor description"
'tool icon filename'= "modelEditor.gif"

In the tool's directory a **"build.xml"** file is also needed. This file is used to build the tool with the help of the **"plugin.xml"** configuration file. The **build.xml** (see Appendix I) is the same for any tool (only the project name can be changed).

3.3. Conclusion

This chapter presented the IVY Workbench tool that supports the modelling and analysis of interactive systems. Section 3.1 presented the model checking based approach supported by the tool. Section 3.2 described how to create a new plugin for that tool.

Chapter 4 – Related Work

This chapter describes CTTE (ConcurTaskTrees Environment) a task modeling tool that has animation and simulation strategies that are similar to the ones intended to be used on the proposed MAL models animator plugin. A previous IVY Workbench plugin - aniMAL - that had a similar goal to this work will also be described.

4.1. CTTE

CTTE[5] (see Figure 18) is an environment for editing and analysing task models. Its main goal is to support the design of interactive applications focusing in the humans and their activities.

In [25] the concepts behind tasks models are presented. In is an important model because it indicates the logical activities that an application can support. A task is defined as an activity that should be performed by the user to reach a goal in the system. A goal can be a desired modification of state or a query to obtain information on the current state of the system. Figure 19 presents an example of a Tasks model.

CTTE uses ConcurTaskTrees (CTT), introduced by Fabio Paternó in [26] and [27]. CTT is a graphical notation (see Figure 20 for an example) with a set of operators used to describe the relationships between tasks.

[5] Available at http://giove.isti.cnr.it/tools/CTTE (last visited 27/10/2012).

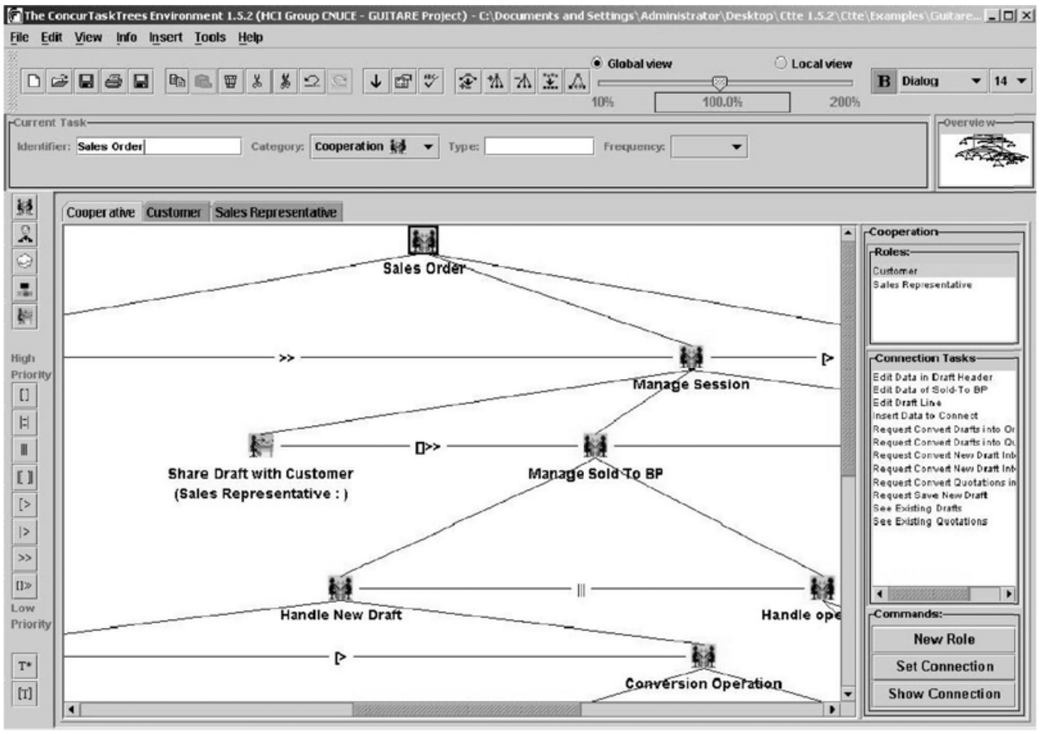


Figure 18 CTTE tool, taken from [25].

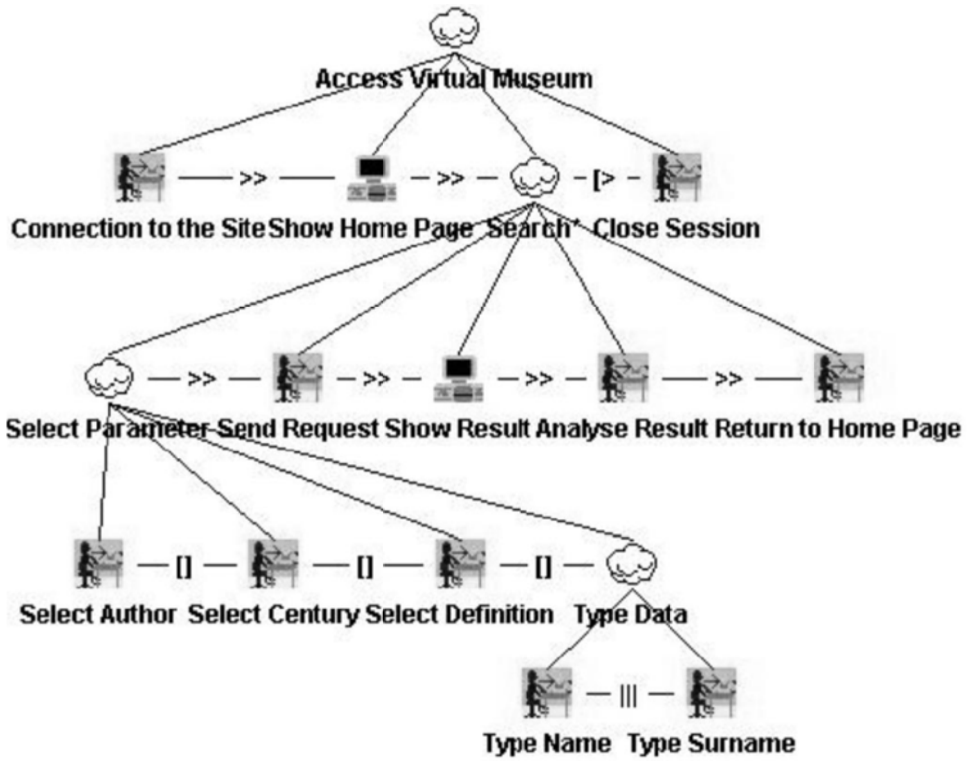
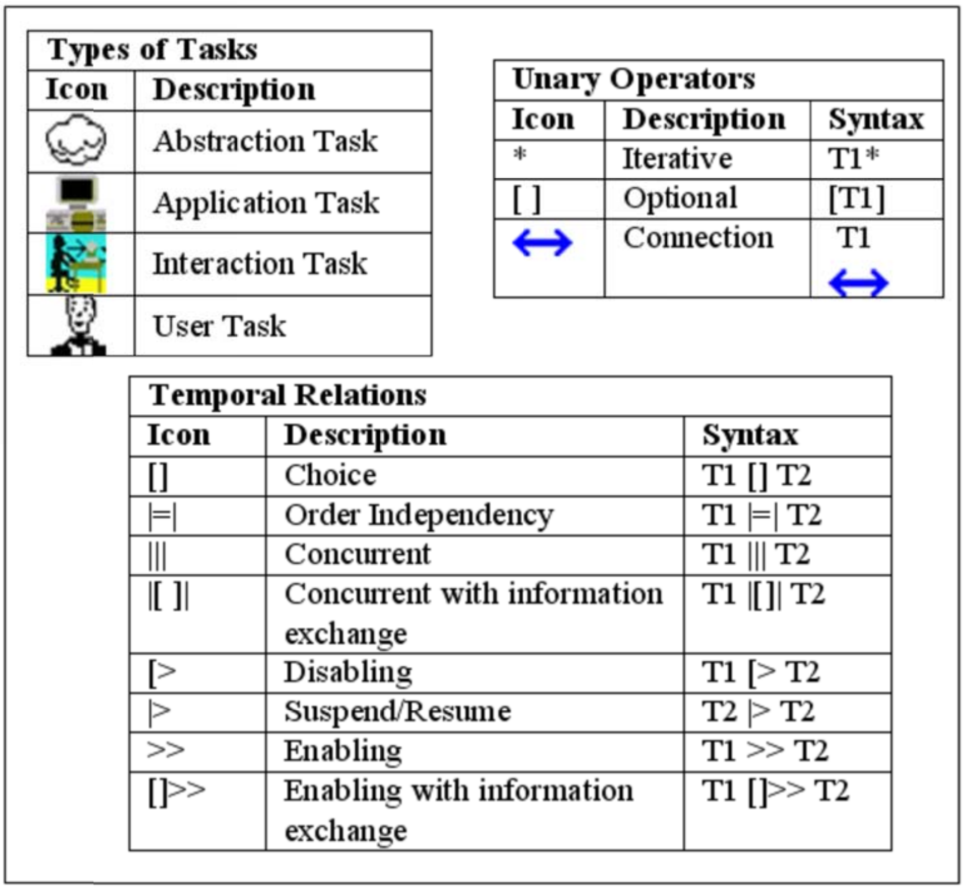


Figure 19 An example of a task model, taken from [25].



Types of Tasks	
Icon	Description
	Abstraction Task
	Application Task
	Interaction Task
	User Task

Unary Operators		
Icon	Description	Syntax
*	Iterative	T1*
[ ]	Optional	[T1]
↔	Connection	T1 ↔

Temporal Relations		
Icon	Description	Syntax
[]	Choice	T1 [] T2
|=|	Order Independency	T1 |=| T2
|||	Concurrent	T1 ||| T2
|[ ]|	Concurrent with information exchange	T1 |[]| T2
[>	Disabling	T1 [> T2
|>	Suspend/Resume	T2 |> T2
>>	Enabling	T1 >> T2
[]>>	Enabling with information exchange	T1 []>> T2

Figure 20 Overview of the CTT notation, taken from [28].

CTTE provides a simulation functionality that is described in [28]. The simulation a ConcurTaskTree involves simulating, in some way, the execution of specific tasks in order to reach a pre-defined goal. In a ConcurTaskTree, tasks are disposed in a hierarchical style. That is, depending on what tasks have been performed, some tasks are enabled and others are disabled. The first step in simulating ConcurTask-Trees is to identify the tasks that are logically enabled at the same time and that is called an enabled task set. The set of all enabled task sets for a specific task model is referred to as an enabled task collection (ETC).

CTTE's tasks simulator is a basic one (see Figure 21). It displays the currently enabled tasks in a list. Double-clicking on a task will simulate the performance of that task. When a task is performed, the enabled tasks are updated accordingly.

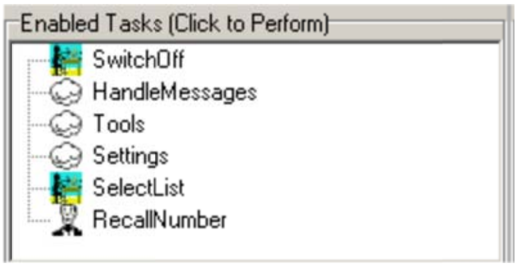


Figure 21 A simple ConcurTaskTree task model simulator, taken from [28].

CTTE is a good case study on how a MAL models animator should function. The relevance of the CTTE environment (see Figure 21) to the present work is its concept of enabled tasks and its simulation capabilities. This concept and the capabilities are described in [28]. The main differences to the proposed WildAniM plugin will be the supported model (Tasks in one case and MAL models in the other), and also that we have attributes in the states of the MAL model, something that does not happen in CTTE.

It is expected that the WildAniMAL plugin will have a similar behaviour to that of CCTE. The actions of the MAL interactors will be represented by similarly to CTTE tasks, and the possible reachable states (enabled by a interactor action on a specific state) will be similar to the enabled tasks of CTTE.

4.2. AniMAL

AniMAL, described in [29], is a prototype of a plugin that was developed for the IVY Workbench. Its most salient feature is that of supporting the definition, at runtime, of a prototype of the interface to be used during the animation. It allows the association, to each attribute and action, of a widget in order to create the prototype.

The AniMAL tool obtains the data that it needs to perform its function from the CoreSystem of the IVY Workbench. More specifically, from the IModel (interactors model) data structure. IModel data is updated by two other plugins of the IVY Workbench. The ModelEditor plugin updates it with model data, which consists of interactors, attributes and actions. The Traces Visualizer plugin updates it with fail traces (sequences of states, defined by their attributes' values, representing behaviours of the model).

What is interesting and useful in AniMAL is that it can generate a UI prototype from the data model it pulls from the CoreSystem. It uses a mapping generation strategy that can be automatic or manual. First it creates an interactor tree from the data model. Then we can opt between choosing which graphical elements will render each of the interactors' attributes and actions, or have the plugin perform that mapping automatically. If we request for the mapping to be done automatically, then the interactors' attributes and actions are rendered with default components.

The default components' rendering is as follows:

- interactor – rendered as panel;
- attributes – rendered using the default widget for their type;
- actions – rendered as buttons.

If the mapping is performed manually, we can choose the widget that is assigned to each attribute. For example, for a temperature attribute we can use a thermometer (see Figure 22) to show the changes in the temperature value, as modelled by the attribute.

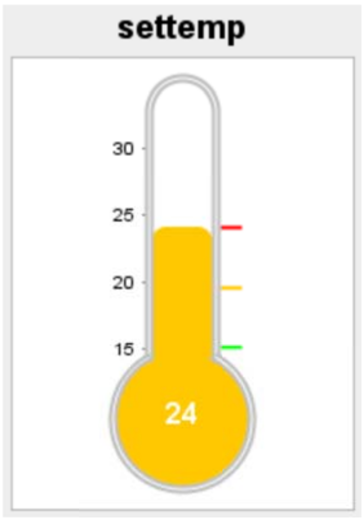
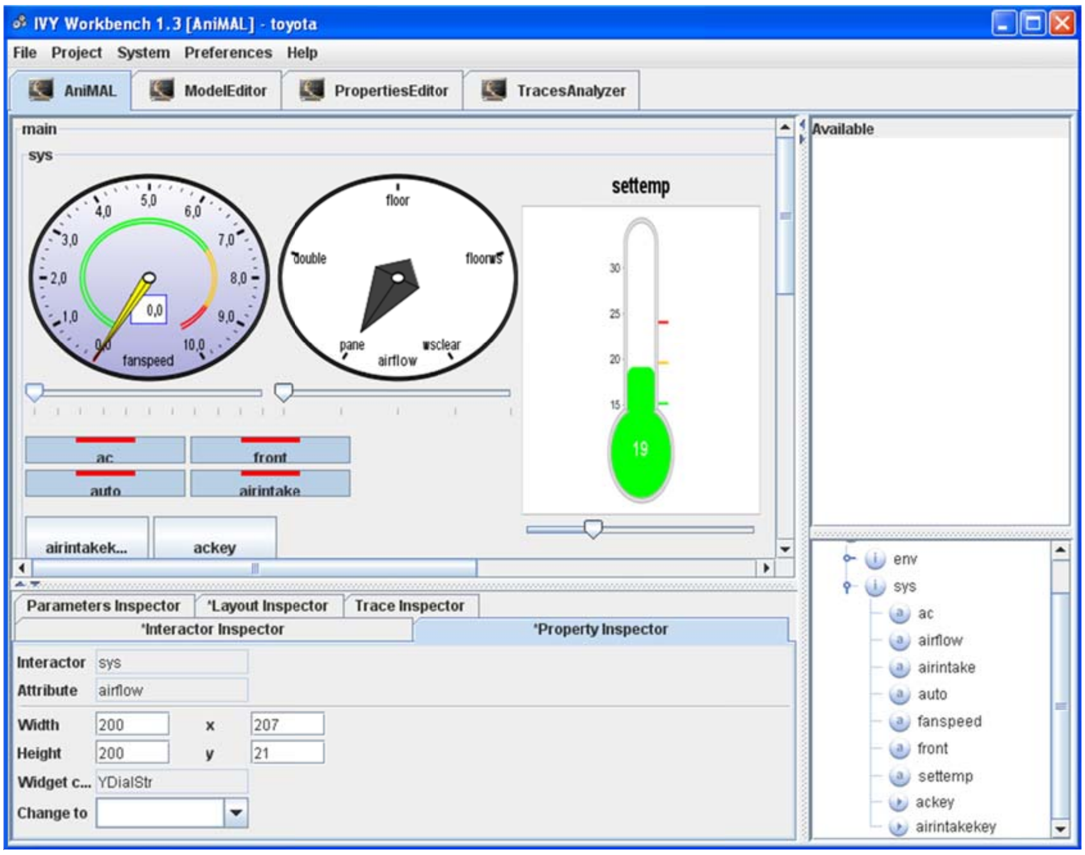


Figure 22 Thermometer, taken from [29].

The list of widgets in AniMAL is extensible, which makes it very appealing to provide a graphic evolution of the values of the attributes, instead of the traditional representations used in these cases (e.g. State or Activity Diagrams). Theses widgets are more familiar and potentially provide an easier insight into the behaviour of an interactors model. Figure 23 presents a UI proptotype with the different widgets used for each of the attributes of an Air-Conditioning interactors model.



Interactor: sys
Attribute: airflow
Width: 200 x: 207
Height: 200 y: 21
Widget c...: YDialStr
Change to:

Figure 23 Prototype of an air condition control panel, taken from [29].

AniMAL's animation capabilities, however, are limited. The tool is only able to animate fail traces, That is different from what has been defined as WildAnIMAL's goal: the capability to animate the interactors models themselves.

4.3. Conclusion

This Chapter described CTTE (ConcurTaskTrees Environment) a task modelling tool. A previous IVY Workbench animation plugin - aniMAL – was also described. Both plugins provide useful insights into what the WildAniMAL plugin should be.

Chapter 5 – WildAniMAL Implementation Approaches

This chapter discusses possible WildAniMAL implementation approaches. Section 5.1 discusses three implementation alternatives. Section 5.2 presents the chosen implementation approach: NuSMV Simulation Capabilities.

The NuSMV model checker provides an interactive shell where commands can be entered. The commands are grouped by the functionality they provide. There are eight main groups: Model Reading and Building, Simulation, Checking Specifications, Bounded Model Checking, Checking PSL Specifications, Execution, Traces, and Administration.

In the context of the present work, we are interested in those commands that help perform an interactive simulation of a NuSMV specification. Having that in mind, the groups of commands which are important to mention are: Model Reading and Building, and Simulation.

Sections 5.2.1 and 5.2.2 provide commands' descriptions that are focused on those aspects (options and environment variables) that are effectively used in this work. More detailed descriptions can be found in [12].

Section 5.2.3 provides a NuSMV simulation example where all the presented commands are used.

5.1. Implementation Approaches

In this Section, the main approaches to implementing the WildAniMAL plugin will be analysed. Three approaches are considered. Section 5.1.1 looks at the possibility of generating and using a Finite State Machine representation of the MAL interactors model to drive the animation. Section 5.1.2 looks at using the BDD representation of the MAL interactors model (created by NuSMV, the verification engine used by IVY Workbench) instead of creating our own finite state machine. Finally, Section 5.1.3 looks at the possibility of using NuSMV's simulation commands, available on its interactive mode, to perform the animation.

5.1.1 Generating a Finite State Machine

This approach can be described as transforming the MAL interactors model into a Finite State Machine (FSM) model. An introduction to the theory behind FSM is available in Section 2.3.

To use this approach an algorithm to translate MAL models into some FSM representation has to be developed and implemented. That work can be complex and time consuming and also tests of the algorithm implementation's correctness are needed. Due to these reasons this approach can be risky, and good results cannot be guaranteed beforehand.

The main advantage of this approach is that only the original MAL model is used, and the results from the simulation process are easily interpreted in the context of, and incorporated into, the MAL's model iterative creation process. Other advantage is that, if this approach can be efficiently implemented, then it will be as easy to perform an interactive simulation of the MAL's model (creating the FSM one step at a time) as it will the full generation of its FSM model. Because the algorithm will be custom made it will be easily adaptable to any need desired.

To face this approach's risks, NuSMV's flat model FSM capabilities can be used. These capabilities are supported by the following commands:

- build flat model - Compiles the flattened hierarchy into a Scalar FSM;
- build boolean model - Compiles the flattened hierarchy into boolean Scalar FSM;
- write flat model - Writes a flat model to a file;
- write boolean model - Writes a flat and boolean model to a file.

However, if the NuSMV FSM capabilities are used, then the main advantage stated above can be lost, due to the translation process between MAL model and the NuSMV generated FSM model. The simulation will no longer happen at the abstraction level of the MAL models, but at the level of the NuSMV specifications created from those models.

5.1.2 NuSMV Binary Decision Diagrams

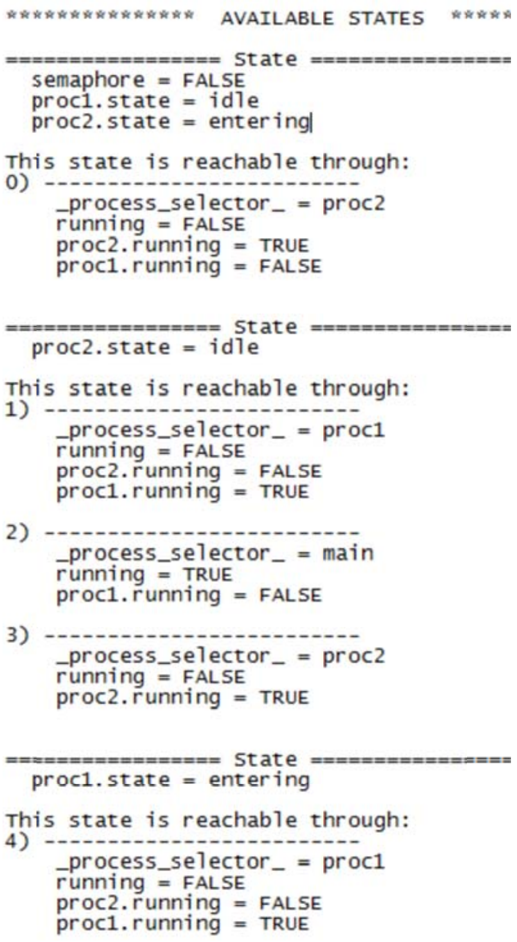
This approach can be described as using the BDD representation of the MAL interactors model, created by the NuSMV model checker, to perform the animation.

Binary Decision Diagrams (presented in Section 2.4) are used by the NuSMV model checker to perform model checking over the NuSMV model. These diagrams are not easily understandable and can be difficult to use for the purpose of implementing the WildAniMAL plugin.

This approach is not the best one because the initial MAL interactors model is translated to a NuSMV model that is read by NuSMV model checker and transformed into BDD. Because two translations steps are made, doing the analysis of the results obtained by animating the BDD, and using them to help the modeling process of a MAL interactors model, will be a daunting task. This is because several artificial variables can be added and transformations made between the two models and the BDD.

5.1.3 NuSMV Simulation Capabilities

The NuSMV model checker has simulations commands that can be used to help implement the proposed MAL interactors model animator plugin. An example of the NuSMV's simulation capabilities is presented in Figure 24.



```
***************  AVAILABLE STATES  *****

================= State ================
  semaphore = FALSE
  proc1.state = idle
  proc2.state = entering

This state is reachable through:
0) ------------------------
    _process_selector_ = proc2
    running = FALSE
    proc2.running = TRUE
    proc1.running = FALSE

================= State ================
  proc2.state = idle

This state is reachable through:
1) ------------------------
    _process_selector_ = proc1
    running = FALSE
    proc2.running = FALSE
    proc1.running = TRUE

2) ------------------------
    _process_selector_ = main
    running = TRUE
    proc1.running = FALSE

3) ------------------------
    _process_selector_ = proc2
    running = FALSE
    proc2.running = TRUE

================= State ================
  proc1.state = entering

This state is reachable through:
4) ------------------------
    _process_selector_ = proc1
    running = FALSE
    proc2.running = FALSE
    proc1.running = TRUE
```

Figure 24 NuSMV simulation example.

Figure 24 shows the available states at a given moment in the simulation. The concept of Available States is similar to the concept of Enabled Tasks in CTTE. Enabled Tasks are calculated when the CTTE's user interactively selects a task to perform and CTTE'S simulator shows what the next enabled (we can also say available) tasks are. Because the SMV Model is produced from the MAL interactors model, in the IVY Worbench tool, it can be used for simulation purposes with the NuSMV model checker. The NuSMV commands that can be used for that purpose are the following:

- **read_model** → Reads a NuSMV fille into NuSMV;
- **pick_state** → Picks a state from the set of initial states;
- **simulate** → Performs a simulation from the current selected state;

The difficulty of this approach is that these commands must be invoked from the proposed WildAniMAL plugin. However, the commands are only available in interactive mode, and as such are not well suited to be called from an external process.

Conceptually the main problem with this implementation approach is that the SMV Model is slightly different from the initial MAL interactors model (as stated in Section 5.1.2). Therefore a process of constant translation and interpretation of animation results from SMV model to MAL model has to be made and that can be problematic and inefficient. Nevertheless, this is still better than directly using BDDs (NuSMV uses the BDDs to run the simulation), were there would be two steps between the original model and the representation our tool would use to support the animation.

Considering the above, this approach was the chosen one for the implementation of the WildAniMAL plugin.

5.2. NuSMV Interactive Shell

The NuSMV Interactive Shell offers an interaction mode that initiates a read-eval-print loop, in which commands can be executed. The activation of the shell is done by invoking the model checker with the "-int" option:

system prompt> **NuSMV -int** <RET>

NuSMV>

When the default "NUSMV>" shell prompt is displayed, the system is ready to accept and execute user commands.

A NuSMV command is a sequence of words. The first word represents the command to be executed and the remaining words are its arguments. With the "set" command it is possible to assign values to environment variables, which in turn influence the behaviour of the commands.

5.2.1 Model Reading And Building

The commands in this group are used for the parsing and compilation of the model into a BDD and are the following:

read_model -i model-file. Reads a NuSMV file into NuSMV.

If the **-i** option is not specified, the command reads the file specified in the environment variable ***Input_File***. If the option is specified the command sets the environment variable ***input_file*** to ***model-file***, and reads the model from the specified file.

go - Initializes the system for verification.

This command is responsible for reading the model (unless it has already been read), and generating a BDD from it. The model is first flattened, which includes instantiating modules by substituting their actual parameters for the formal parameters, and then prefixing the result with each particular instance's name, scalar variables are encoded to create a boolean model, and then the BDD is generated.

5.2.2 Simulation Commands

The commands in this group allow simulating a NUSMV specification and are the following:

pick state [-i [-a]]

Chooses an element from the set of initial states, and makes it the current state (replacing the old one). The chosen state is stored as the first state of a new trace, which will grow in number of states, as simulation evolves. The state can be chosen according to different policies, which can be specified via command line options. By default the state is chosen in a deterministic way.

Options:

-i → enables the user to interactively pick up an initial state. The user is requested to choose one state from a list of possible states. If the number of possible states is too high, then the user has to specify some further constraints on the values of the variables in the current state;

-a → by default, states only show those variables that have changed from the previous state. With this option, NuSMV displays all state and frozen variables regardless of whether they have are changed and unchanged with respect to the previous state. This option works only if the *-i* option has been specified.

simulate [-i [-a]] [-k steps]

Performs a simulation from the current selected state. The command generates a sequence of at most **steps** states (representing a possible execution of the model), starting from the **current state**. The current state can be set via the **pick_state** command**.**

Options:

-i → enables the user to interactively choose every state of the trace, step by step. As with **pick_state**, if the number of possible states is too high, then the user has to specify constraints on the state attributes. These constraints are used only for a single simulation step and are forgotten in the following ones.

-a → again, this makes NuSMV display all the state and frozen variables (changed and unchanged) during every step of an interactive session (which is not done by default).

-k steps → this option defines the maximum length of the path to be generated. The default value is determined by the default simulation steps environment variable **shown_states** (ranges between 1 and 100, and default is 25).

5.2.3 Simulation Example

To illustrate the use of the NuSMV simulation commands a model of a garage gate will be used. This model will be specified in the interactors language mentioned earlier in section 2.5. This specification can be seen in Figure 25.

```
/*
# positions
#  main=(100,100)
*/

types
  States = {opening, closing, opened, closed}

interactor main
 attributes
  [vis] currentState: States
 actions
  [vis] Ac # User Remote Controller used to control the gate
  Ia # Sensor that detects when gate is fully open
  If # Sensor that detects when gate is fully closed
 axioms
  [] currentState = closed
  currentState = closed -> [Ac] (currentState' = opening)
  currentState = opening -> [Ac] (currentState' = closing)
  currentState = closing -> [Ac] (currentState' = opening)
  currentState = closing -> [If] (currentState' = closed)
  currentState = opening -> [Ia] (currentState' = opened)
  currentState = opened -> [Ac] (currentState' = closing)
  per(Ia) -> currentState = opening
  per(If) -> currentState = closing
```

Figure 25 Interactors model of a garage gate.

To understand what this model represents we can see the state diagram in Figure 26.

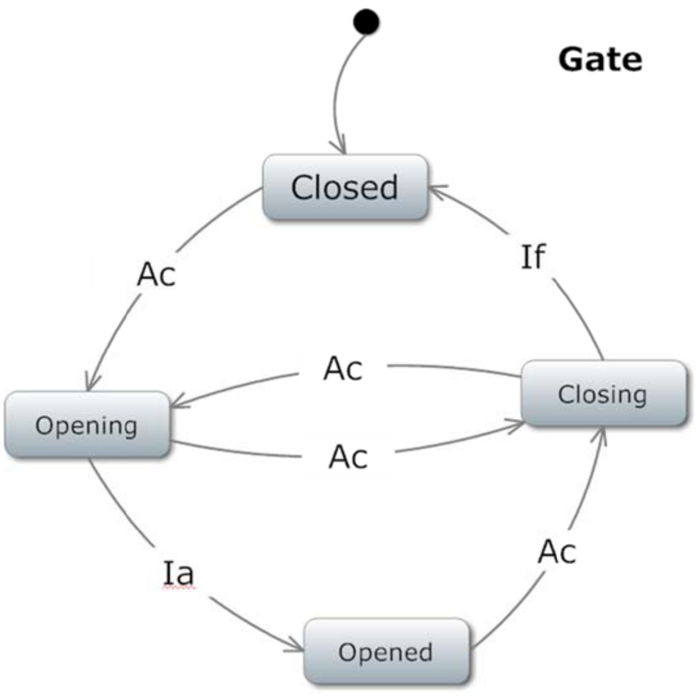
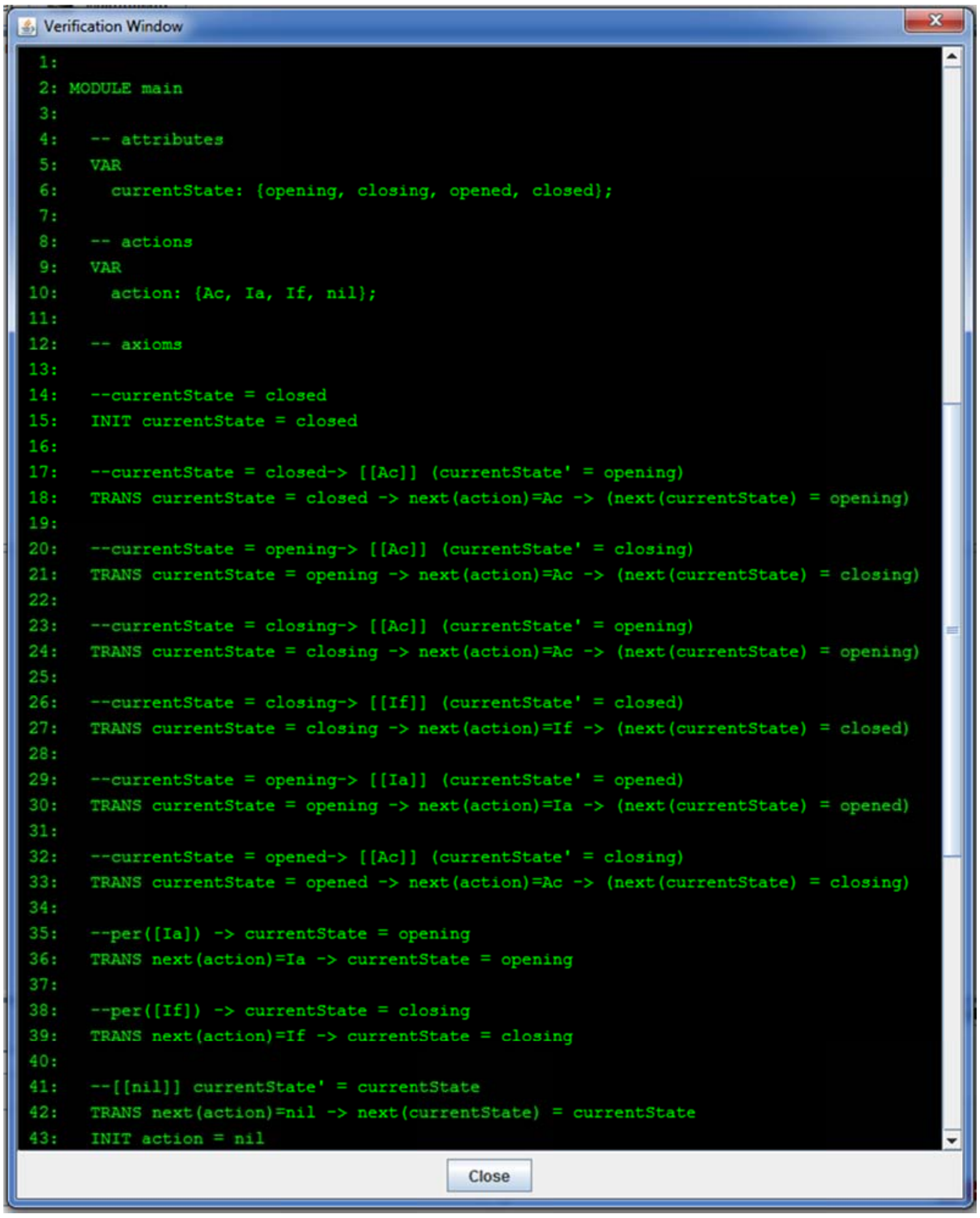


Figure 26 State Diagram of gate model.

With the IVY Workbench tool we can compile the interactors model, in Figure 27, to a NuSMV specification.



```

MODULE main

  -- attributes
  VAR
    currentState: {opening, closing, opened, closed};

  -- actions
  VAR
    action: {Ac, Ia, If, nil};

  -- axioms

  --currentState = closed
  INIT currentState = closed

  --currentState = closed-> [[Ac]] (currentState' = opening)
  TRANS currentState = closed -> next(action)=Ac -> (next(currentState) = opening)

  --currentState = opening-> [[Ac]] (currentState' = closing)
  TRANS currentState = opening -> next(action)=Ac -> (next(currentState) = closing)

  --currentState = closing-> [[Ac]] (currentState' = opening)
  TRANS currentState = closing -> next(action)=Ac -> (next(currentState) = opening)

  --currentState = closing-> [[If]] (currentState' = closed)
  TRANS currentState = closing -> next(action)=If -> (next(currentState) = closed)

  --currentState = opening-> [[Ia]] (currentState' = opened)
  TRANS currentState = opening -> next(action)=Ia -> (next(currentState) = opened)

  --currentState = opened-> [[Ac]] (currentState' = closing)
  TRANS currentState = opened -> next(action)=Ac -> (next(currentState) = closing)

  --per([Ia]) -> currentState = opening
  TRANS next(action)=Ia -> currentState = opening

  --per([If]) -> currentState = closing
  TRANS next(action)=If -> currentState = closing

  --[[nil]] currentState' = currentState
  TRANS next(action)=nil -> next(currentState) = currentState
  INIT action = nil
```

Figure 27 NuSMV specification of the gate model.

Having this NuSMV specification it is possible to simulate it using the NuSMV interactive mode. To start the simulation we have to do the following:

system prompt> NuSMV -int gate.smv

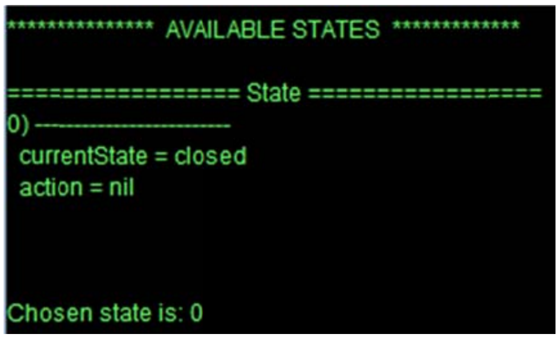
NuSMV> go

NuSMV>

The previous sequence of programs reads the model to the NuSMV system. After doing that we have to choose an initial state from the possible initial states of the model. In our case we will use the interactive approach, in which the user is able to interactively choose the states of the trace he wants to build. So we have to use the following command:

NuSMV> pick state –i -a

This command has the following result that is shown in Figure 28.



```
**************** AVAILABLE STATES *************

================= State =================
0) -------------------------
  currentState = closed
  action = nil


Chosen state is: 0
```

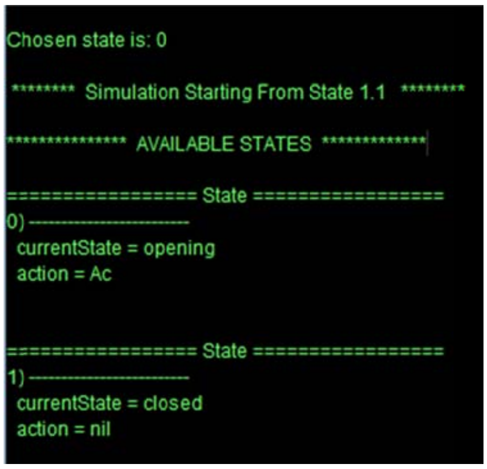
Figure 28 The result of pick_state –i –a command.

This result means that this model has only one initial state, and because of that the state is automatically chosen as the initial state.

To proceed with the simulation we have to use the simulate command with a parameter k with value 1, which will make the simulation advance one step. The command is:

NuSMV> simulate -i –a -k 1

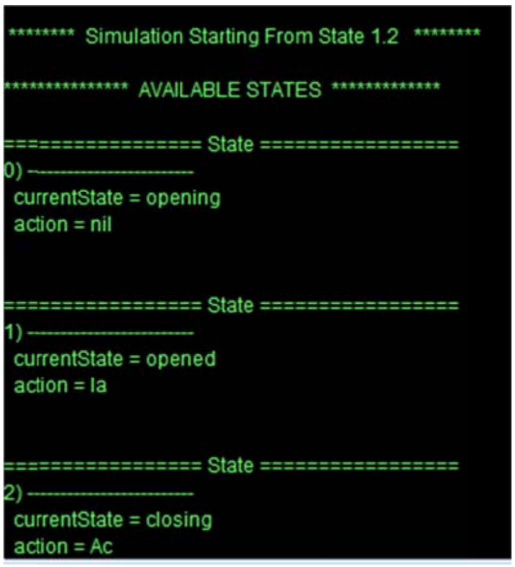
and NuSMV returns the available states (see Figure 29).



```
Chosen state is: 0

******** Simulation Starting From State 1.1 ********

*************** AVAILABLE STATES *************

================= State =================
0) -------------------------
  currentState = opening
  action = Ac


================= State =================
1) -------------------------
  currentState = closed
  action = nil
```

Figure 29 The result of simulate –i –a –k 1 command.

Now we have to choose one of the available states. If we choose 0 and use the simulate command again, we end up the result shown in Figure 30.



```
******** Simulation Starting From State 1.2 ********

*************** AVAILABLE STATES *************

================ State ================
0) -------------------------
 currentState = opening
 action = nil


================ State ================
1) -------------------------
 currentState = opened
 action = Ia


================ State ================
2) -------------------------
 currentState = closing
 action = Ac
```

Figure 30 The result of choosing state 0.

We can continue the simulation using the simulate command until no more states are reachable.

In this example we showed that the path, illustrated in the State Based diagram of Figure 31, is possible to be demonstrated using NuSMV's Interactive mode and a small set of its available commands.

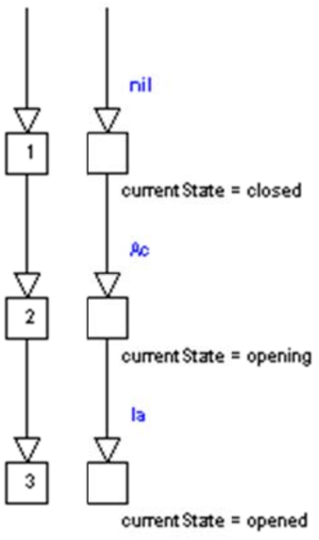


Figure 31 State Based diagram showing the simulation path.

5.3. Conclusion

In this chapter we described the NuSMV Interactive Mode and its available commands. To more effectively illustrate it we presented a real example of a model: a garage gate. The model was specified in the MAL Interactors language, compiled to a NuSMV specification, and finally a simulation was carried out. That simulation used the commands that were previous presented. We can conclude that NuSMV simulation commands can be useful to implement a MAL Interactors model animator because the needed output and general mechanism is easily available and ready to use.

Chapter 6 – WildAniMAL Implementation

This chapter describes the implementation of WildAniMAL as a plugin for the IVY Workbench tool. An architectural view with UML diagrams is provided. To provide more detail on the implementation, an explanation of the main methods is presented.

6.1. WildAniMAL's Architecture

Because the JAVA programming language was used, the architecture of the WildAniMAL plugin can be easily explained by using UML diagrams for each of the Java packages created. This scheme for presenting the architecture is well suited to provide the "main picture" of the implementation.

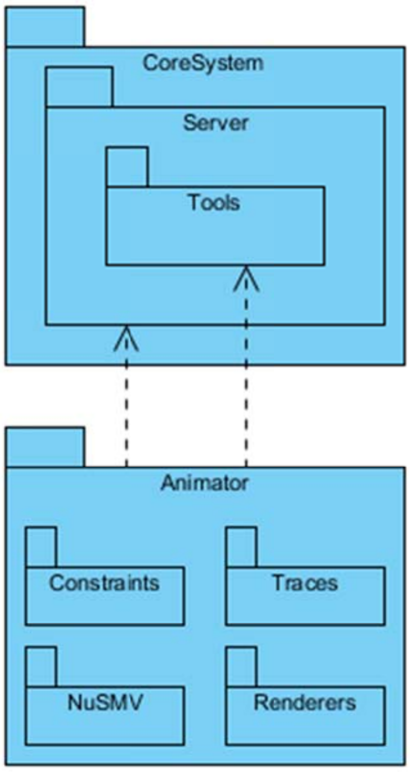


Figure 32 WildAniMAL plugin architecture as a package diagram.

The architecture of the plugin has five packages (see Figure 32): Animator (the root package), Traces, NuSMV, Constraints and Renderers. The plugin also depends on the Tools and Server packages of the IVY Workbench CoreSystem main package.

The Animator root package contains the following classes: **Gui**, **Main**, **TreeActions and Parser.** These classes interact with the inner packages of Animator package, as it will be shown next.

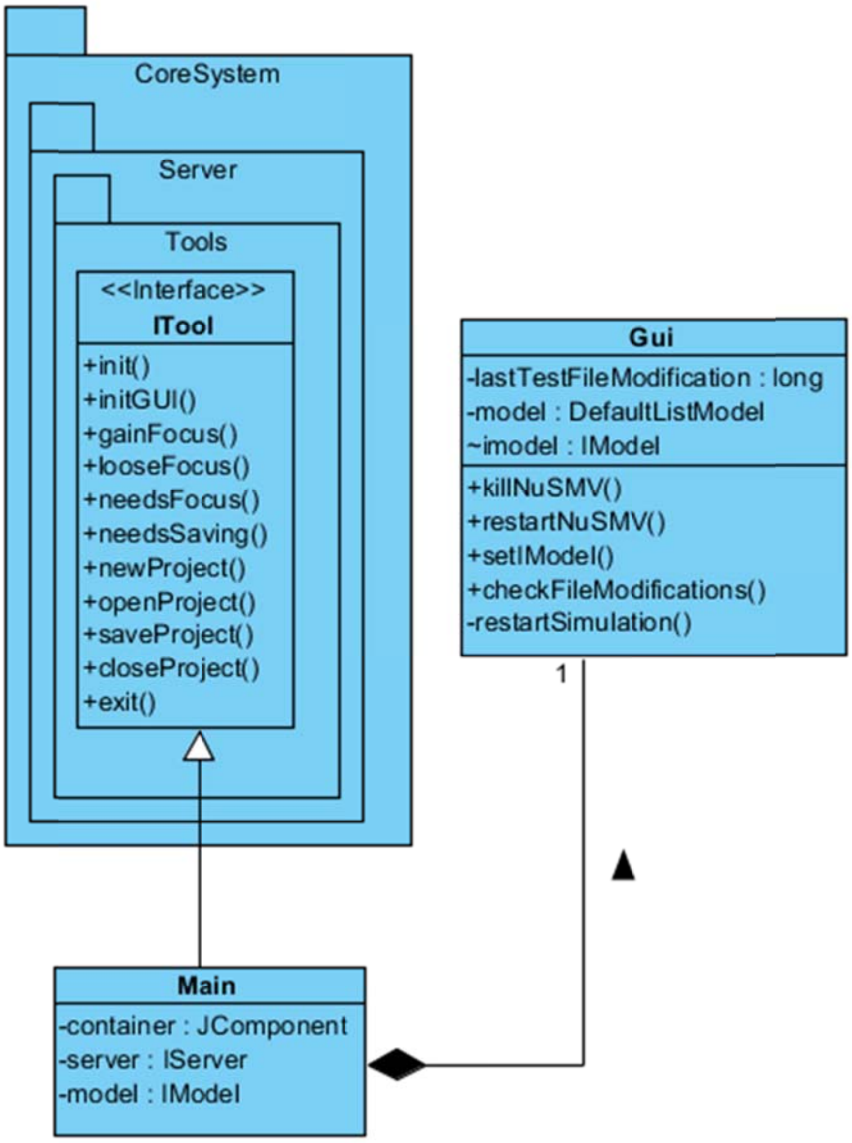


Figure 33 Main package class diagram.

The **Main** class is responsible for implementing the interface needed to create a plugin for the IVY Workbench tool as explained in Section 3.2. In particular, it initializes the **Gui** class, in the **initGui** method of the interface.

The **Gui** class (see Figure 33), as the name may give a clue, handles the graphical user interface of the plugin. It has the code for displaying the buttons, panels and tables, used in the interface. It also handles the events for the buttons presses.

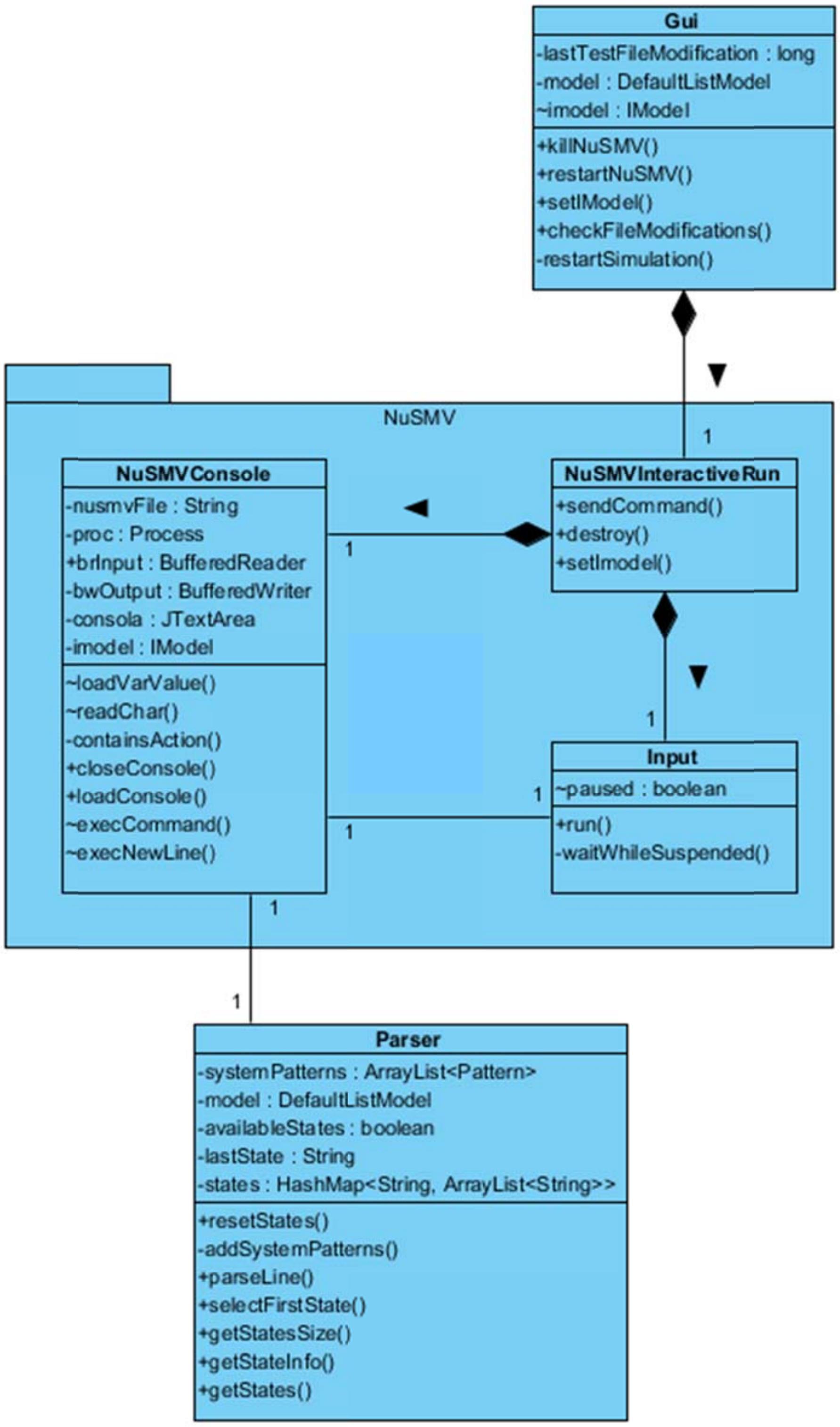
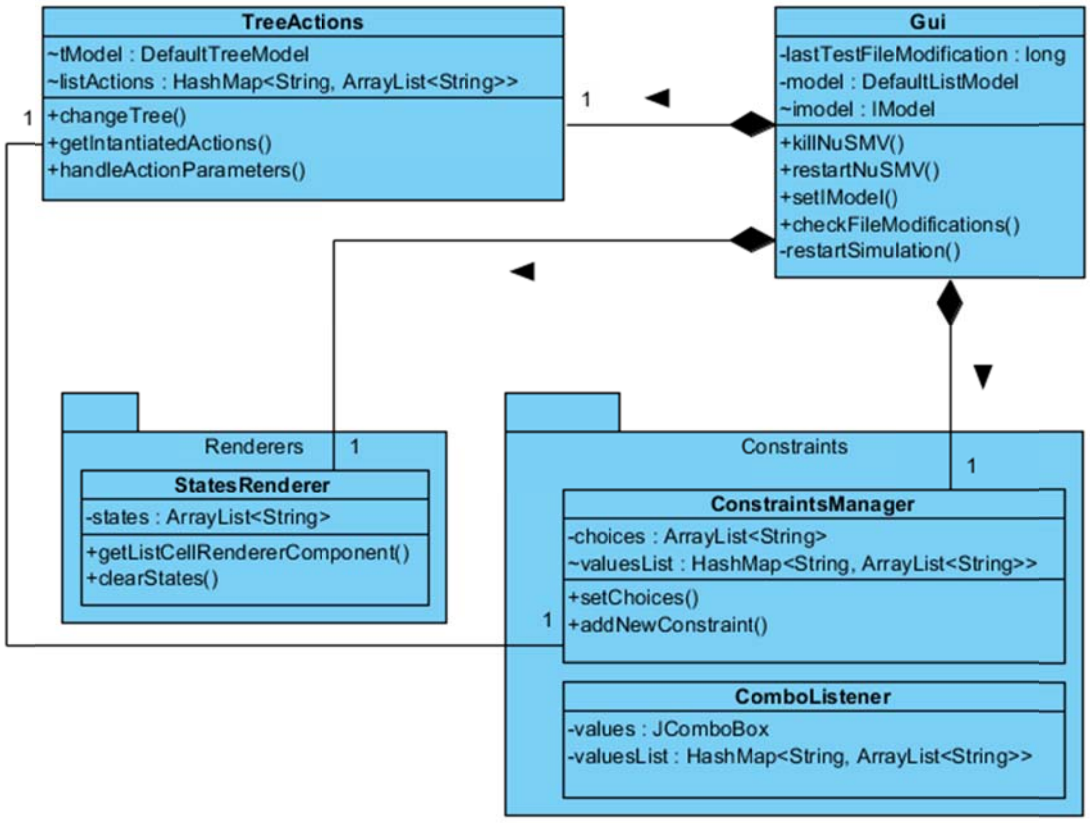


Figure 34 NuSMV package class diagram.

The **NuSMV** package (see Figure 34) is responsible for handling the communication between the graphical user interface, in which the user can select simulation commands, and the external NuSMV model checker process (that works in interactive mode as explained in Section

5.2). It uses the **Parser** class to parse the states contained in the results obtained from the NuSMV model checker process. These states, obtained in each simulation step from NuSMV, feed a JList. When the states are parsed it is possible to see the information associated with each of them in another JList (StateInfo).



TreeActions
~tModel : DefaultTreeModel
~listActions : HashMap<String, ArrayList<String>>
+changeTree()
+getIntantiatedActions()
+handleActionParameters()
Gui
-lastTestFileModification : long
-model : DefaultListModel
~imodel : IModel
+killNuSMV()
+restartNuSMV()
+setIModel()
+checkFileModifications()
-restartSimulation()
Renderers
StatesRenderer
-states : ArrayList<String>
+getListCellRendererComponent()
+clearStates()
Constraints
ConstraintsManager
-choices : ArrayList<String>
~valuesList : HashMap<String, ArrayList<String>>
+setChoices()
+addNewConstraint()
ComboListener
-values : JComboBox
-valuesList : HashMap<String, ArrayList<String>>
1
1
1
1
1

Figure 35 Constraints package class diagram.

The **Constraints** package (see Figure 35) has the function of enabling WildAniMAL to filter the states obtained from NuSMV based on the values of their attributes. In each of the simulations steps, the user has to choose a current state from the set of all possible states at that point in the simulation. This set can become large. Hence, filtering the states with a conjunction of conditions on the values of their attributes, helps the user focus on the states that matter at each particular moment, and also helps him choose the right state to proceed with the simulation.

Package **Renderers** is responsible for rendering the states that meet the constraints of the filter. Currently, that is done by changing the background color.

When the future states of a simulation are more than one hundred, NuSMV does not produce the list of possible states. In this case, **"Too many States"** appears in **stateList**.

Then, constraints have to be entered to filter the states (here at the level of the NuSMV model checker process) in order to obtain the states' list needed to proceed with the simulation.

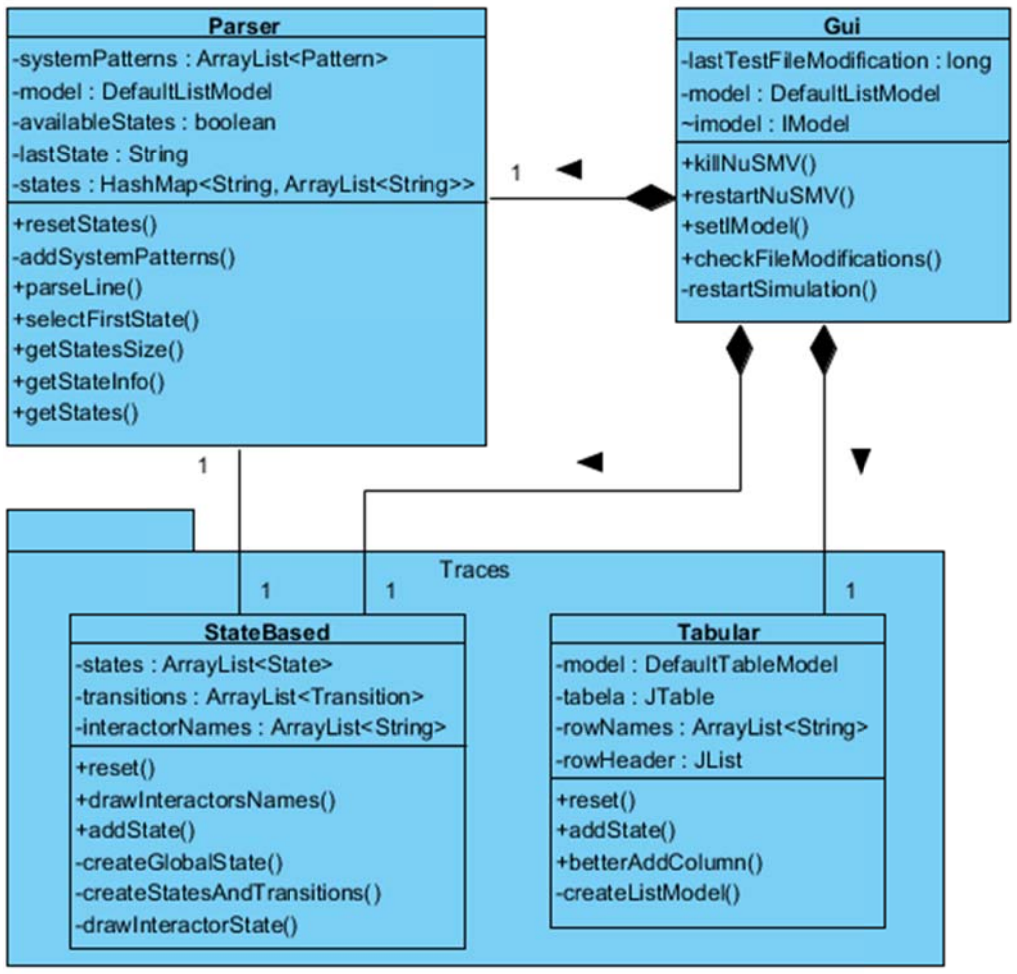


Figure 36 Traces package class diagram.

The **Traces** package (see Figure 36) is responsible for showing the states resulting from the simulation. This is achieved with the help of visual representations. These visual representations are two of the ones already available in the Traces Visualizer plugin of the IVY Workbench: ***StateBased*** and ***Tabular***, and their implementation is described in [24]. They were adapted to receive states one by one, because the trace is created step by step as the user is entering his choices in the interactive simulation. In the Traces Visualizer plugin the trace is fully formed with all states and is displayed promptly.

When a new state is parsed in the **Parser** class, a method (***addState***) is called in each visual representation that adds the state info and does what is needed to update the drawn visual representation, so that it reflects the newly added state information. In the case of the

StatedBased representation**,** the update is done by calling **drawInteractorState** (which performs a repaint)**.**

It's easy to add more visual representations because the main class of a representation will only have to implement the **addState** method, and the graphical (or textually, if wanted) representation. This feature makes the plugin extensible regarding the visual representations available.

6.2. WildAniMAL's Source Code Description

This section presents a description of the most relevant aspects of the implementation's source code. That description will be grouped by the packages described in the previous section.

6.2.1 Animator Package

Class Main

As already mentioned, the **Main** class implements the plugin interface of the IVY Workbench tool.

```
public class Main implements ITool {
      /** container for application. */
      private JComponent container = null ;

      /** application core server. */
      private IServer server = null;

      private Gui frame;
      private IModel model;

      public Main() {
            frame= new Gui();
            frame.saveLastFileModified();
      }
```

The previous code shows that the **Main** class implements the ITool interface, that is, the plugin interface of the IVY Workbench tool. The variables ***container*** and ***server*** relate to the **CoreSystem** of the IVY Workbench tool, and enable the plugin to communicate with it. In particular, they enable the WildAniMAL plugin to retrieve information from the shared data structure used by all the plugins of the tool. It is through this shared information that the plugin integrates its own functionalities (in this case, the simulation of the interactors model - using the NuSMV specification as an intermediate representation) with the rest of the tool. The **model** variable will hold all data from the interactors model and is used to retrieve information needed to construct constraints and also to help the NuSMV package classes perform their function.

The **Main()** constructor initializes the **GUI** class which, has its name indicates, is the Graphical User Interface of the plugin. The **saveLastFileModified** method is used to store in a variable the last time when **test.smv** (the SMV Specification file) was modified. That information will be used to test when a new model was compiled in the Properties Editor. Whenever a new interactors model is compiled, the WildAniMAL simulation has to be restarted.

Another method that is used during the initialisation of the plugin is **initGUI**.

```
public void initGUI(JFrame main, final JComponent rootContainer) {
    this.container = rootContainer;

    container.setLayout(new BorderLayout());
    container.add(frame,BorderLayout.CENTER);
}
```

The method simply adds WildAniMAL's graphical user interface (given by the GUI class as **frame** variable) to the JComponent (rootContainer) that has been assigned to it by the Core System. Each plugin is graphically located in a tab.

Next the handling of focus must be provided.

```
public void gainFocus() {
    frame.checkFileModifications();

    CServer i=(CServer)server;

    model=i.getModel();
    frame.setIModel(model);
}
```

The **gainFocus** method is executed whenever the user chooses the plugin WildAniMAL in the IVY Workbench tool (by clicking in the respective tab). In this method, a check is made to determine if a new interactors model was compiled, in which case the simulation will be restarted. That is done by using the **checkFileModifications** method of the **Gui** class. Also, the reference to the interactors model data structure is retrieved for future use during the simulation process.

The **loseFocus** method simply stores the last time when the current interactors's model was compiled. That is done to enable the verification made in the **gainFocus** method.

```
public void loseFocus() {
    frame.saveLastFileModified();
}
```

Finally the exit behaviour of the plugin must be provided.

```
public void exit() {
    frame.killNuSMV();
}
```

The **exit** method frees all resources used in WildAniMAL plugin, and is called when the user exits IVY Workbench application.

Class Gui

The **Gui** class implements WildAniMAL's graphical user interface (see Figure 37) and handles buttons events. It also coordinates all the functionalities implemented in this plugin: the simulation, constraints handling, drawing of traces visual representations and filters.

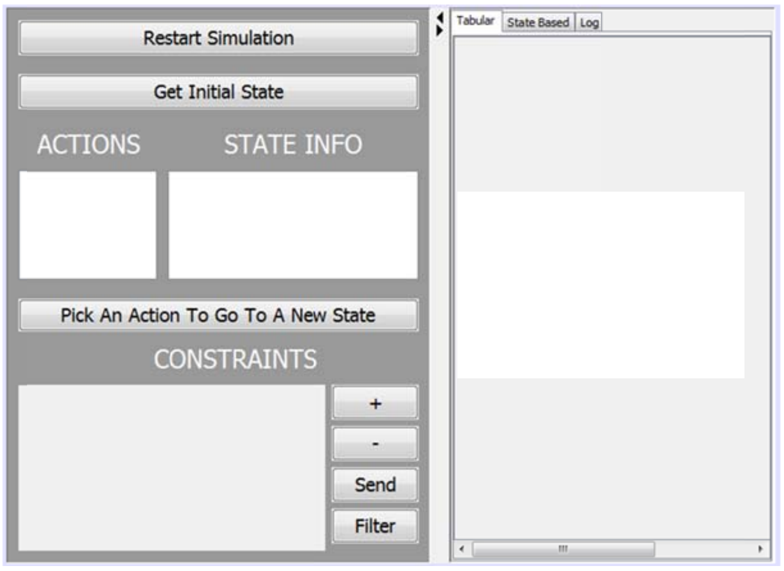


Figure 37 Graphical user interface implemented by GUI class.

The constructor of the **Gui** class initializes the graphical components and also the auxiliary classes that will handle WildAniMAL's functionalities.

```
public Gui() {
    initComponents();

    GridLayout gd = new GridLayout(0,1);

    cPanel.setLayout(gd);

    model = (DefaultListModel) statesList.getModel();
    parser = new Parser(model, statesList);

    treeActions = new TreeActions();
    nusmv = new NuSMVInteractiveRun("test.smv", Consola, parser);

    constraints = new ConstraintsManager(cPanel, treeActions);

    stRenderer = new StatesRenderer();
    statesList.setCellRenderer(stRenderer);

    stateBased = new StateBased(stateBasedPanel);
    tabular = new Tabular(tabela, scrollTabela);
}
```

The **statesList** variable (a JList) holds the current states returned on each step of the simulation. A reference to its **model** (data) and the component itself are passed on in the **Parser** class constructor, because in the simulation process, and in the associated parsing needed, this class updates the states list directly.

The **treeActions** variable (instance of **TreeActions** class) is also initialized here, and is responsible for storing locally the actions of the interactors model to help constraints' creation. For that reason, the **constraints** variable (an instance of the **ConstraintsManager** class) is initialized using a reference to it.

Another class that is instantiated in the Gui constructor, and the most important of them all, is **NuSMVInteractiveRun (**variable **nusmv**). It is the **nusmv** variable that will setup the actual interactive simulation, using an external NuSMV model checker process. The variable is initialized with a JTextarea (**Consola**) that will receive the textual output of the commands sent to the NuSMV process, with a reference to the **Parser** class **(**variable **parser)** that will be used to parse that same output, and with the name of the file holding the NuSMV specification. This file's name is hardcoded (by choice) because it is a temporary file generated in the **Properties Editor** plugin of IVY Workbench when the current interactors model is compiled.

The **StatesRenderer** class (variable **stRenderer**) is also instantiated in the **GUI** constructor. It is the **stRenderer** variable that will be responsible for showing the result of filtering the elements of **statesList** (by changing their background color) when some constraints are applied.

Finally, the **StateBased** **(**variable **stateBased)** and the **Tabular** **(**variable **tabular)** classes are also instantiated in the **Gui** constructor. The two corresponding variables will enable showing the progress of a simulation, through the visual representations they implement. Whenever the user chooses a state to proceed with the simulation, these two variables receive that information which will be shown with the corresponding graphical strategy. The **stateBased** variable provides a kind of state diagram and the **tabular** variable provides a normal table.

Next the method that shares the interactors model (**IModel** variable), between all the variables that need it, is provided. These variables are: **nusmv**, **constraints** and **treeActions**.

```
public void setIModel(IModel mod) {
        imodel = mod;
        nusmv.setImodel(mod);
        constraints.setChoices(imodel);
        treeActions.changeTree(imodel);
}
```

Next the methods that handle button events are presented.

```
private void btGetFirstStateActionPerformed(java.awt.event.ActionEvent evt) {
        nusmv.sendCommand("pick_state -i -a");
        btGetFirstState.setEnabled(false);
}
```

This method handles the click event on the **Get Initial State** button, and does that by sending the shown command to the NuSMV model checker (using the **nusmv** variable).

Next, another button's (**Pick State**) event handling is provided.

```
private void btPickStateActionPerformed(java.awt.event.ActionEvent evt) {
        int index = statesList.getSelectedIndex();
        if (index != -1) {
            //Add State to the Trace Visualizer
            stateBased.addState(""+index, parser.getStateInfo(""+index));
            tabular.addState(parser.getStateInfo(""+index));

            //SendCommand to show next states
            if (statesList.getModel().getSize() > 1)
                nusmv.sendCommand(""+index);

            model.clear();
            nusmv.sendCommand("simulate -i -a -k 1");
            stRenderer.clearStates();

            //Update Trace Visualization
            stateBasedPanel.repaint();
            tabela.repaint();
            interactorsNamesPanel.repaint();
        }
    }
```

This method picks up the current state choice, that is, the selected number in the **statesList** JList. If a choice exists (not null) then the respective state info is added to the traces visual representations (**stateBased** and **tabular** variables). After that, the choice (state number) is sent to NuSMV model checker and **statesList** is cleared. Then, the **simulate** command is sent to NuSMV which will return states to fill **statesList** again. Finally repaints are made in order to show the state update in the trace's visual representations.

Next the **Filter** button event handling is provided.

```
private void filterActionPerformed(java.awt.event.ActionEvent evt) {
        JPanel constPanel;
        JComboBox vars, op, vals;
        String var, opc, val;

        ArrayList<String> lista = new ArrayList<String>();

        for (int i = 0; i < cPanel.getComponentCount(); i++) {
            constPanel = (JPanel) cPanel.getComponent(i);

            vars = (JComboBox) constPanel.getComponent(0);
            op = (JComboBox) constPanel.getComponent(1);
            vals = (JComboBox) constPanel.getComponent(2);

            var = (String) vars.getSelectedItem();
```

```
                opc = (String) op.getSelectedItem();
                val = (String) vals.getSelectedItem();

                if (val.contains("(")) {
                    val = treeActions.handleActionParameters(val);
                }
                lista.add(var + " " + opc + " " + val);
            }
```

The **filterActionPerformed** method gets the constraint conditions from cPanel into a list (variable **lista)**. The constraints may have action with parameters and if so a special method **handleActionParameters** is used to replace the internal notation used in NuSMV (e.g. **doSomethingAction_a_b_c)** by the more user friendly notation used in MAL models (**doSomethingAction(a,b,c)**). After that **lista** is sent to the **parser**, which returns the states that match the constraints' conditions. These states are then passed on to the states renderer (**stRenderer)** that renders them differently (red background) on the **statesList.**

The event handler for the **Send button** (method **btSendActionPerformed**) is similar to the previous method. The difference is that the constraints are joined in a string as a conjunction (using the **&** operator) and are sent to the NuSMV model checker. Hence, instead of filtering the current list of states being displayed, this method sends constrains that will be used by NuSMV to generate a new (smaller) list. This is particularly useful when the list of possible states is too big (over 100 states) in which case NuSMV will not generate it.

Class Parser

The **Parser** class is responsible for parsing the output of the NuSMV model checker. Because the parsing process consists mainly in obtaining states and their info, this class has a data structure to store them and provides methods to query that information.

The patterns used in the parsing process are initialised with the **addSystemPatterns** method.

```
    private void addSystemPatterns() {
        String ident="([a-zA-Z][a-zA-Z0-9_$~.><\\[\\]\\-]*)";
        String value="([a-zA-Z0-9_$~.><\\[\\]\\-]*)";
        systemPatterns.add(Pattern.compile("(\\d+)\\)")); // State filter
        systemPatterns.add(Pattern.compile("(\\s*)"+ident
        + "(\\s*)=(\\s)*(\\d)*" + value)); // Action\Atribute Filter
        systemPatterns.add(Pattern.compile("\\s*Set of future states is EMPTY: "
        + "constraints too strong\\? Try again.\\s*")); // Set of Future States Empty
        systemPatterns.add(Pattern.compile("\\s*Too many \\([0-9]+e\\+[0-9]+\\) "
        + "future states to visualize. Please specify further constraints: \\s*"));
        // To Many States After Constraint Send
    }
```

The **addSystemPatterns** method compiles the patterns (**regular expressions**) used in the parsing process of the NuSMV model checker's output. The first pattern matches a state number. The second pattern matches an action or attribute value (that is part of state info). The third pattern matches the indication that after applying constraints there aren't any states to proceed with simulation. The last pattern matches the indication that applied constraints aren't sufficient and more have to be specified.

The **parseLine** method is responsible for parsing a line of the NuSMV model checker's output. The parsing is done by identifying specific keywords and patterns in the text produced by NuSMV. The first part of this method checks if the line contains "AVAILABLE STATES", which means that a new simulation step has started (a state has been chosen). In that case the **states** structure is cleared to receive new states info.

```
        public void parseLine(String lineRead) {
                if (lineRead.contains("AVAILABLE STATES")) {
                    availableStates=true;
                    states = new HashMap<String,ArrayList<String>>();
                    Gui.setBtPicksState(true);
                }
```

Next the method checks if the line contains a new state number. If so, then it also tests if the model was on a situation where constraints insertion was needed. Then the state number is added to **states.** That state number is also stored in an auxiliary variable **lastState.**

```
    String aux;
        if (availableStates) {
            matcher = systemPatterns.get(0).matcher(lineRead);
            if (matcher.lookingAt()) {
                if (model.contains("Too Many States")) {
                    model.removeElement("Too Many States");
                     Gui.setConstraints(false);
                }

                aux = matcher.group(1);
                lastState = aux;

                if (!model.contains(aux) && !"".equals(aux)) {
                    model.add(Integer.parseInt(aux),aux);
                    states.put(aux, new ArrayList<String>());
                }
            }
```

After parsing a state number **(lastState),** all the actions and attributes values that make part of its info are parsed, and stored in its entry in **states**.

```
    else if (!"".equals(lastState)) {
                matcher = systemPatterns.get(1).matcher(lineRead);
                if (matcher.lookingAt()) {
                    aux = matcher.group(0);
                    if (states.containsKey(lastState) && !"".equals(aux)) {
                        states.get(lastState).add(aux.trim());
                        if (aux.contains("action")) {
                            aux = aux.replace("action = ","");
                            model.set(Integer.parseInt(lastState), aux);
                        }
                    }
                }
    }
```

This part of the method detects and signals a situation where constraints have to be entered. In this situation, the **Pick State** button is disabled, because there are not states to choose from.

```
                matcher = systemPatterns.get(2).matcher(lineRead);
                matcher1 = systemPatterns.get(3).matcher(lineRead);
                if (matcher.lookingAt() || matcher1.lookingAt()) {
                    model.clear();
                    model.addElement("Too Many States");
                    Gui.setConstraints(true);
                    Gui.setBtPicksState(false);
                }
```

Finally, this part of the method detects a situation where the output for a simulation step has ended and a state choice, to proceed with the simulation, is needed.

```
        if (lineRead.contains("Choose a state form the above") ) {
            availableStates = false;
            lastState = "";
            Gui.setBtPicksState(true);
        }
    }
```

6.2.2 Constraints Package

The **ConstraintsManager** class is responsible for managing constraints that can be sent to the NuSMV model checker or used to filter states. This class constructor is initialized with the constraints JPanel (see Figure 28), and with a reference to an instance of the **TreeActions** class that will help with retrieving information on the actions in the interactors model.

The **setChoices** method (some parts of the code are presented) fills the variables **choices** (all names of attributes and variables in the interactors model) and **valuesList (**the corresponding values for any variable in **choices)**.

```
for (int j=0; j<v.size();j++ ) {
            elem=v.get(j);

            aux = new ArrayList<String>();
            def=model.getDef(elem);

            if (!def.equals("")) {
               if (def.contains("array")) {
                   choices.remove(choices.size()-1);
                   aux=model.getAttributeValuesOnly(elem);
                   valuesList.put(elem,aux);
               }
                else if (defs.containsKey(def))
                   valuesList.put(elem, defs.get(def));
                else {
                   aux=model.getAttributeValuesOnly(elem);
                   valuesList.put(elem,aux);
                   defs.put(def,aux);
                }
               choices.add(elem);
            }
}
```

In the method, when an action variable has parameters, the method **getInstantiatedActions** of the **TreeActions** class is used, to unfold the types of these parameters, in order to obtain all the possible combinations of parameter values.

The **addNewConstraint** method is responsible for creating a new constraint condition. A constraint is graphically represented by three combo-boxes. The first holds all the elements of **choices**, the second holds the operators (“=” and “!=”), and the last is updated with all possible values (values from **valuesList**) for the element currently selected on the first combo-box.

```
public void addNewConstraint() {
    if (choices.size() > 0) {
        JPanel constraint = new JPanel();
        ArrayList<String> operators= new ArrayList<String>();
        ArrayList<String> values;
        JComboBox vars= new JComboBox(choices.toArray());
        constraint.add(vars);
        operators.add("=");
        operators.add("!=");
        JComboBox op= new JComboBox(operators.toArray());
        constraint.add(op);

        String key = choices.get(0);

        if (valuesList.containsKey(key))
            values = valuesList.get(key);
        else
            values = new ArrayList();

        JComboBox vals= new JComboBox(values.toArray());
        constraint.add(vals);
        ComboListener cl = new ComboListener(vals,valuesList);
        vars.setSelectedIndex(0);
        vars.addActionListener(cl);
        op.setSelectedIndex(0);

        if (vals.getItemCount()>0)
            vals.setSelectedIndex(0);

        painel.add(constraint);
        painel.updateUI();
    }
}
```

A textual representation of the constraint condition is what is sent to the NuSMV model checker, or used to filter states in **statesList**.

6.2.3 NuSMV Package

Class **NuSMVInteractiveRun** is responsible for handling the communication between the graphical user interface, in which the user can execute simulation commands, and the external NuSMV model checker process.

```
public NuSMVInteractiveRun(String nusmvFile,
      JTextArea consola1,Parser parser1){
   console = new NuSMVConsole(nusmvFile,consola1,parser1);
   console.loadConsole();

   input = new Input(console);
   console.setInput(input);

   input.start();
   console.execCommand("go");
}
```

NuSMVInteractiveRun’s constructor instantiates the **NuSMVConsole** class (variable **console**) with a reference to the file with the NuSMV specification, a reference to the JTextArea (**consola1**) that will receive the output from the model checker process, and also a reference (parameter **parser1)** to the Parser instance that will parse each line of the NuSMV model checker’s output. The **Input** class is also instantiated, which starts a thread (variable **input**) that will be continuously reading data from the input stream of the NuSMV model checker process. Finally, the command **go** is sent to the **NuSMVConsole,** and it initializes the simulation of the current NuSMV specification (as explained in section 5.2.1).

```
public void run() {
      while(true) {
         console.readChar();
         try {
            waitWhileSuspended();
         } catch (InterruptedException ex) { }
      }
}
```

```
public void setPaused(boolean p) {
    this.paused = p;
}

private void waitWhileSuspended()
        throws InterruptedException {
    while (paused) {
        Thread.sleep(200);
    }
}
```

The **Input** thread can be paused, if no data is available in the stream. That is done with a state variable (**paused**) that is constantly checked (with **waitWhileSuspended**), in the **run** method of this thread. Without this, the process would be kept active while waiting for input, which would create a big impact in CPU usage.

Next methods of the **NuSMVConsole** class are provided.

```
public void loadConsole() {
    String[] cmdarray = {"","-int", nusmvFile};
    cmdarray[0] = ""+System.getProperty("user.dir") + File.separator +
            "NuSMV" + File.separator + "bin" + File.separator + "NuSMV";

    ProcessBuilder pb = new ProcessBuilder(cmdarray);
    pb.redirectErrorStream(false);

    try {
        proc = pb.start();
    } catch (IOException ex) { }

    InputStream inputStream = proc.getInputStream();
    OutputStream outputStream = proc.getOutputStream();

    InputStreamReader inputStreamR = new InputStreamReader(inputStream);
    OutputStreamWriter outputStreamW = new OutputStreamWriter(outputStream);

    brInput = new BufferedReader(inputStreamR);
    bwOutput = new BufferedWriter(outputStreamW);
}
```

The **loadConsole** method sets NuSMV's command path, which has to be inside IVY Workbench application path and more specifically in NuSMV/bin/NuSMV, and starts a java process with it. Then its input (**brInput)** and output (**brOutput**) streams are retrieved.

```
void execCommand(String command) {
    input.setPaused(false);
    try {
        bwOutput.write(command);
        execNewLine();
    } catch (IOException ex) {

    }
}
```

The **execCommand** method is used to send a command to the NuSMV process. Each time it is executed it starts by activating the **Input** thread (awaking it).

```
void readChar() {
        int c = 0;
        char ch = 0;
        try {
            c = brInput.read();

            if (c == -1) {
                input.setPaused(true);
                parser.selectFirstState();
                return;
            }

            ch = (char) c;
        } catch (IOException ex) { return; }
```

The **readChar** method will read a character at a time from **brInput.** This has to be done in this way (the usual way is to read line by line) because sometimes the NuSMV process does not print the last line of result, for example when it is waiting for the user to choose a state and subsequently to press enter (newline). When **brInput** returns no char (a value equal to -1), then **Input** thread is paused and **readChar** methods returns immediately.

```
if (ch=='\n') {
            str = sb.toString();
            parser.parseLine(str);
```

```
                        sb = new StringBuffer();
                        consola.append(str+"\n");
                        consola.setCaretPosition(consola.getDocument().getLength());
        }
```

The characters consecutively read by the **readChar** method are accumulated in a StringBuffer (**sb),** until a newline (\n) is read. In that situation, the stringBuffer is transformated in a String (the read line), which is sent to the **Parser** to be parsed and printed in the JTextArea **Log.** The StringBuffer **sb** is also cleared to begin accumulating characters to form the next line.

```
        else {
            sb.append(ch);
            aux = sb.toString();

            if (aux.matches("  action = [a-zA-Z0-9]+") && containsAction(aux)) {
                parser.parseLine(aux);
                consola.append(aux);
                consola.setCaretPosition(consola.getDocument().getLength());
                sb = new StringBuffer();
            }
```

If **ch** is not newline then it is accumulated in **sb.** Then a test is made to determine if the string is an action (using the **containsAction** method). This is the strategy used to overcome the problem of NuSMV not printing the last line of command output, which originates the last action of a state's result of a command not being parsed. This strategy works because the action is the last line printed by NuSMV in any state attributes listing.

The **containsAction** method checks if a string passed as a parameter (**aux**) is a valid action in the interactors model.

```
            private boolean containsAction(String aux) {
                    try {
                    ArrayList<String> actions =
                            imodel.getActionsVariable("main.action");
                    aux = aux.replace("  action = ","");

                    boolean match = false;

                    String ac;
                    aux = aux.trim();
```

```
String encontrada = "";

for(String act: actions) {
    ac = act.trim();

    if ( ac.compareTo(aux)==0 )   {
            match = true;
            encontrada = ac;
            break;
    }
}
```

If an action is encountered (**match** is true) then an additional test is made. This handles the situation when one action has a name that starts with another action name. For example: setValueMCP and setValue. If this verification is not performed, then if the correct action to be matched is **setValueMCP**, what is (**wrongly**) matched is **setValue.**

```
if (match) {
            for(String act: actions) {
                ac = act.trim();

                if (ac.contains(encontrada) &&
                    ac.length() > encontrada.length())
                        return false;
            }
        }
                return match;
        }
        catch (Exception e) {
            return false;
        }
    }
```

6.3. Conclusion

This chapter described the implementation of WildAniMAL as a plugin for the IVY Workbench tool. Section 6.1 presented a high level view of how the implementation code was organized. As the work of implementation was a Java programming task, this was explained with UML class diagrams, showing how classes were organized into packages.

Section 6.2 makes an extensive explanation of the code. One reason to do that is to fully present some problems that were encountered during development, and how they were overcome. By explaining the implementation in detail, it becomes easier for anyone to understand the code and consequently improve it at a later occasion. This also made it possible to think about how some implementation choices were made, and in the task of writing the explanation to describe this code, some methods were implemented in a more efficient way.

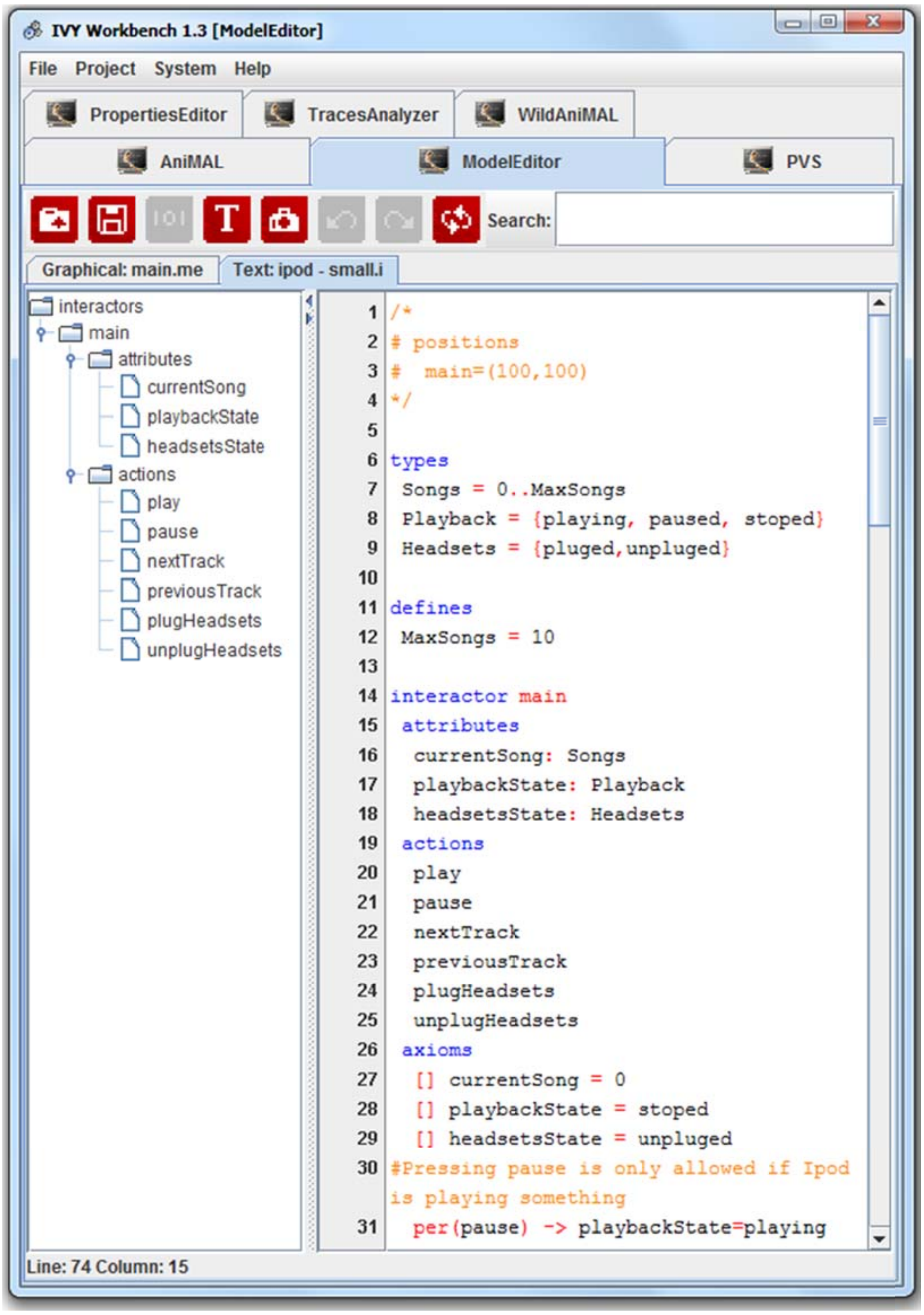
Chapter 7 – Using WildAniMAL

This chapter demonstrates the WildAniMAL plugin of the IVY Workbench tool. Section 7.1 presents a simulation example that makes extensive use of all the functionalities available in the plugin. Section 7.2 presents the conclusions of this chapter.

7.1. WildAniMAL's Usage Example

To explain how the WildAniMAL plugin can be used, a small example of an Ipod-like music player will be introduced. To be able to keep the explanation short and understandable, some aspects of the real device will be abstracted in order to work with a simpler model.

First an interactors model of the device will be created using the Model Editor plugin of the IVY Workbench (see Figure 38).

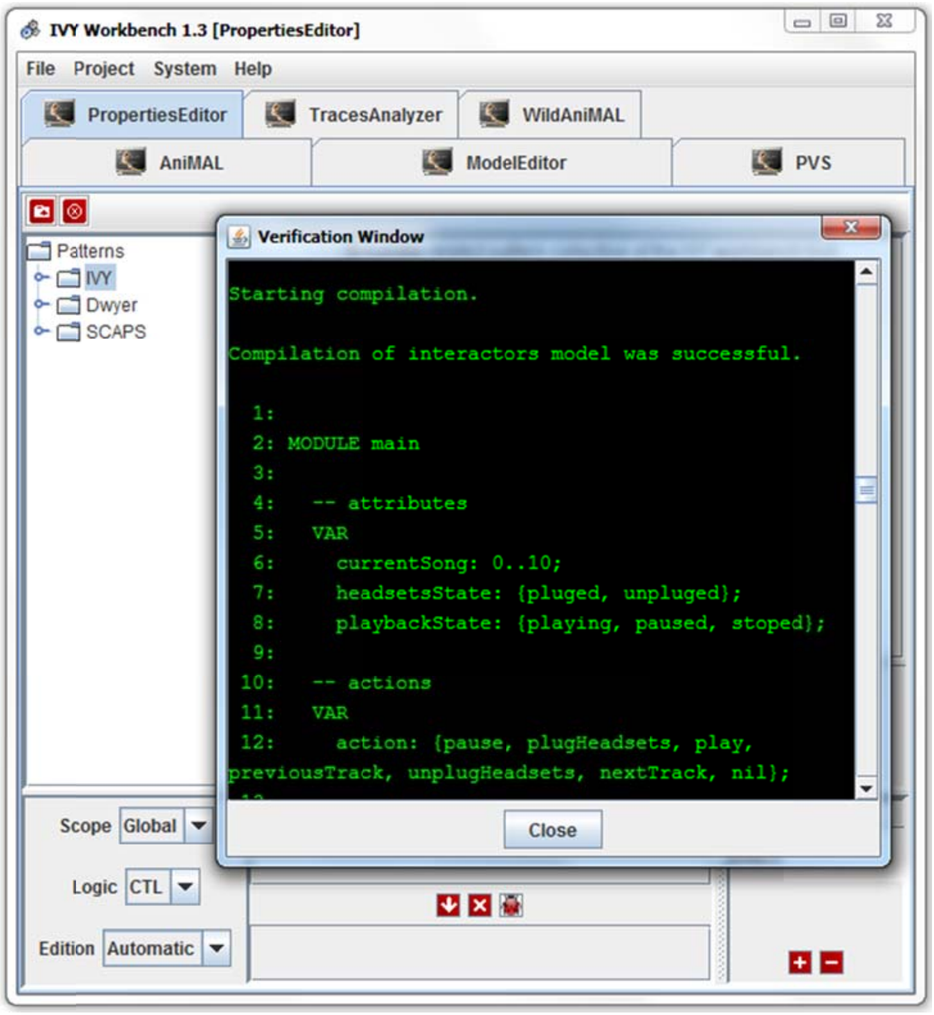


```
/*
# positions
#   main=(100,100)
*/

types
 Songs = 0..MaxSongs
 Playback = {playing, paused, stoped}
 Headsets = {pluged,unpluged}

defines
 MaxSongs = 10

interactor main
 attributes
  currentSong: Songs
  playbackState: Playback
  headsetsState: Headsets
 actions
  play
  pause
  nextTrack
  previousTrack
  plugHeadsets
  unplugHeadsets
 axioms
  [] currentSong = 0
  [] playbackState = stoped
  [] headsetsState = unpluged
#Pressing pause is only allowed if Ipod
is playing something
  per(pause) -> playbackState=playing
```

Figure 38 Ipod model creation with Model Editor plugin.

The full interactors model is presented in Appendix II.

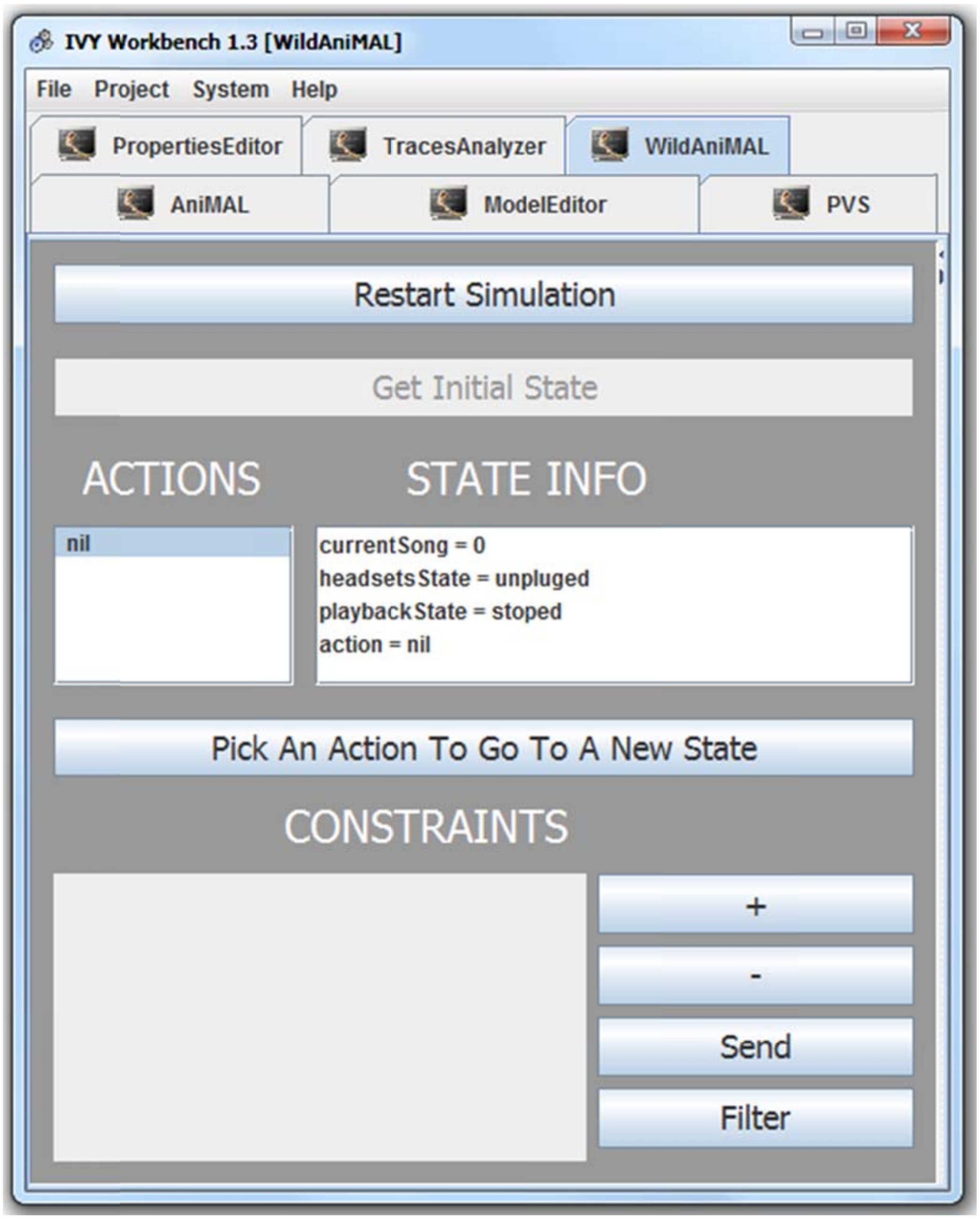
After the model's creation, it has to be compiled to a NuSMV specification. That is achieved in the Properties Editor plugin of the IVY Workbench. To perform that compilation we have to click in the "bug" button (near the "X" button) of the Properties Editor. If the compilation is successful then the respective NuSMV specification is shown (see Figure 39).



```
Starting compilation.

Compilation of interactors model was successful.


MODULE main

  -- attributes
  VAR
    currentSong: 0..10;
    headsetsState: {pluged, unpluged};
    playbackState: {playing, paused, stoped};

  -- actions
  VAR
    action: {pause, plugHeadsets, play,
previousTrack, unplugHeadsets, nextTrack, nil};
```

Figure 39 Ipod interactor model compilation with Properties Editor plugin.

Now we can really start simulating the NuSMV specification, which is the interactors model intermediate representation. When in WIdAniMAL, the first thing to do is click the "**Get Initial State"** button (see Figure 40). That operation results in a list of available initial states. These states can be accessed from the Actions list. Note that in this case the list will only have **nil** actions, one for each possible initial state, since no action is performed to reach the initial state (remember that the **nil** action represents a state transition without an associated action).



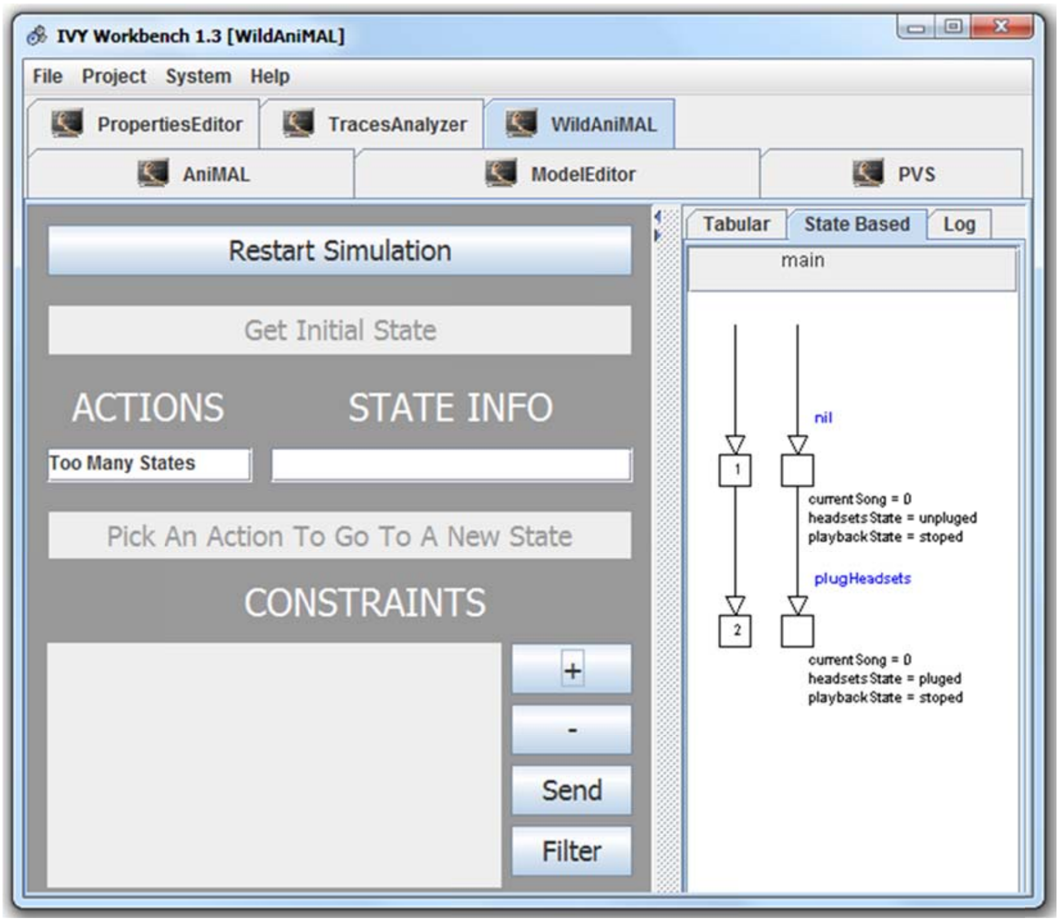
```
currentSong = 0
headsetsState = unpluged
playbackState = stoped
action = nil
```

Figure 40 Result of pressing “Get Initial State” in WildAniMAL.

In the current model, there is only one initial state (**nil** action) to choose. That state’s attributes can be looked up in the **State Info** list (see Figure 40) by selecting the action in the **Actions** list. To choose the state we have to click the “**Pick An Action To Go To A New State**” button.

To perform the interactive simulation, we simply have to continuously choose an action from the **Actions** list. Each time the "**Pick ...**" button is clicked, a new list of actions appears. Each one leading to a new state.

In this example, if the actions: **nil** and **plugHeadsets,** are chosen, in that order, then a situation is reached, in which it is not possible to continue with the simulation, as it can be seen in Figure 41. After **plugHeadsets**, the list of possible future states is too large (more than one hundred possible states). In that case, the actions list presents only one element, which indicates that there are "Too Many States".



currentSong = 0
headsetsState = unpluged
playbackState = stoped

currentSong = 0
headsetsState = pluged
playbackState = stoped

Figure 41 Simulation reached a "Too Many States" situation.

In this case we might want to look at the sequence of the actions executed so far. The sequence of actions and states of the interactive simulation can be shown as a State based diagram (see Figure 42), using a Tabular representation (see Figure 43), or as a textual Log (see Figure 44). The State Based diagram and the Table are visual representation of the trace generated in the simulation. The textual Log shows the details (output of NuSMV process) of the communication (commands and their results) between the plugin and the NuSMV model checker.

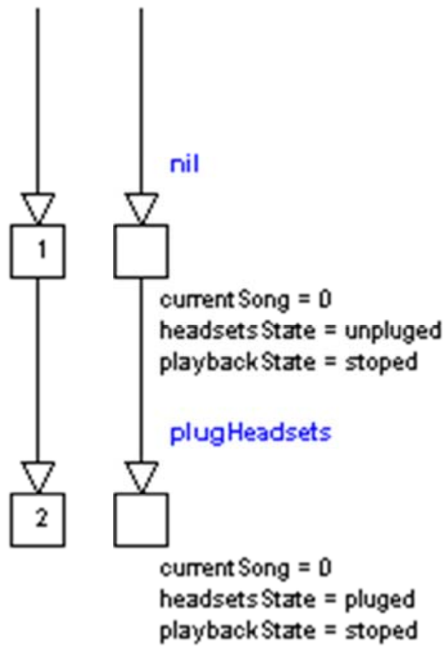
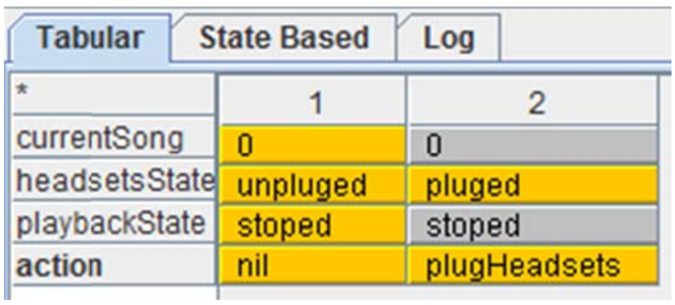
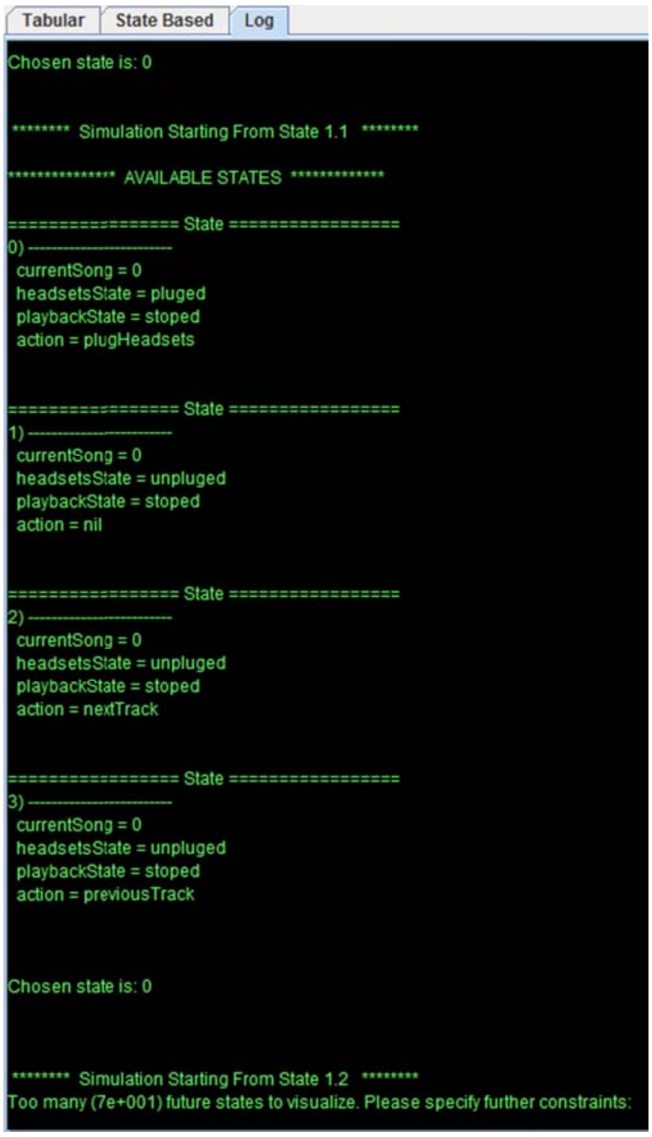


Figure 42 State Based representation of the trace created in the simulation.



*	1	2
currentSong	0	0
headsetsState	unpluged	pluged
playbackState	stoped	stoped
action	nil	plugHeadsets

Figure 43 Tabular representation of the trace created in the simulation.



```
Chosen state is: 0

******** Simulation Starting From State 1.1  ********

************** AVAILABLE STATES *************

================= State =================
0) -------------------------
 currentSong = 0
 headsetsState = pluged
 playbackState = stoped
 action = plugHeadsets


================= State =================
1) -------------------------
 currentSong = 0
 headsetsState = unpluged
 playbackState = stoped
 action = nil


================= State =================
2) -------------------------
 currentSong = 0
 headsetsState = unpluged
 playbackState = stoped
 action = nextTrack


================= State =================
3) -------------------------
 currentSong = 0
 headsetsState = unpluged
 playbackState = stoped
 action = previousTrack



Chosen state is: 0



******** Simulation Starting From State 1.2  ********
Too many (7e+001) future states to visualize. Please specify further constraints:
```

Figure 44 Log representation of the simulation.

In Figure 44, we can see that the current simulation reached a situation where there are too many futures states to choose from (more than one hundred, that is the maximum number of future states that NuSMV can handle). In this case, constrains have to be entered and that is done as shown in Figure 45.

The constraints can be created using the + and – buttons, to add and remove them, respectively. When all constraints are created, they are sent to the NuSMV model checker using the "**Send"** button**.** If the constraints are successful in the job of reducing the number of future states, then a new actions list is available to continue the simulation. That is the current case, as shown in Figure 46.

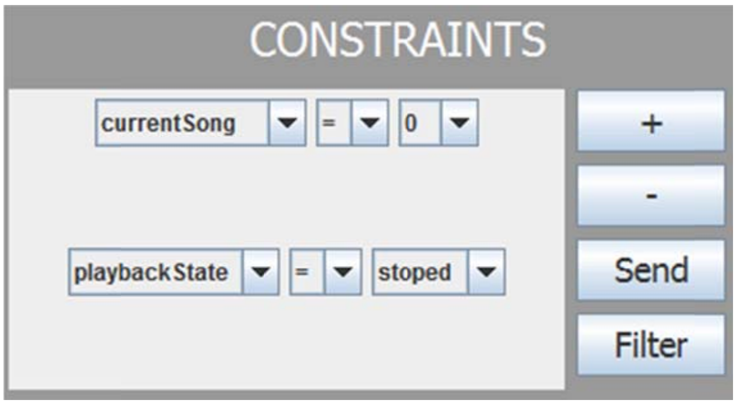
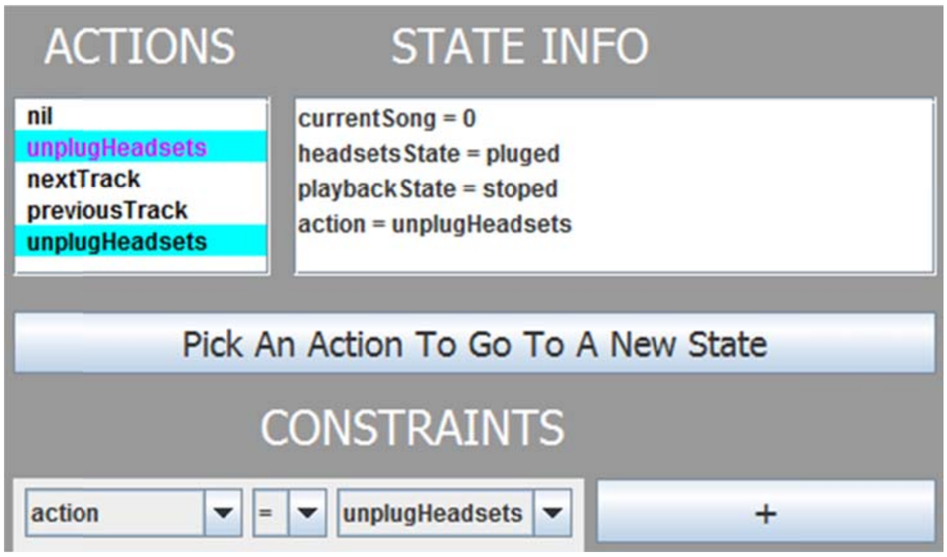


Figure 45 Constraints insertion.

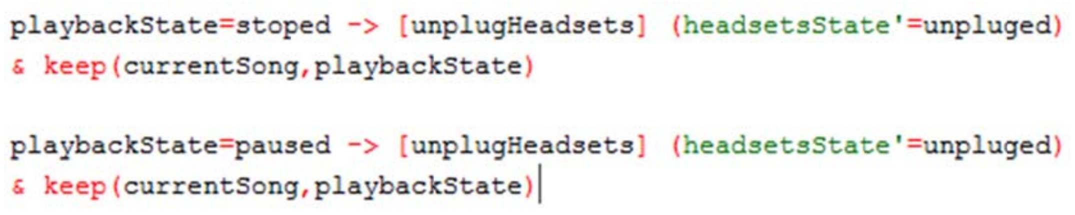
Once the constraints (see Figure 45) have been successfully sent, and a new actions list is returned (see Figure 46), an irregular situation occurs. When filtering the actions list with the constraint **action = unplugHeadsets**, using the **Filter** button, we can see that this action leads to two states. The problem is that one of these makes no sense, because it means that if the headsets are unpluged from the Ipod, then the new value of **headsetsState** is **pluged**. Basically it means that if headsets are unpluged, they remain pluged. Because that cannot happen in real world, the original interactors model probably has an error and has to be corrected.



currentSong = 0
headsetsState = pluged
playbackState = stoped
action = unplugHeadsets

Figure 46 Result of application of the constraints sent.

After an analysis of the interactors model, the conclusion is that under some circumstances the behaviour of the **currentSong** and **playbackState** variables was not being defined. To solve this two news axioms have to be added. These axioms are:



```
playbackState=stoped -> [unplugHeadsets] (headsetsState'=unpluged)
& keep(currentSong,playbackState)

playbackState=paused -> [unplugHeadsets] (headsetsState'=unpluged)
& keep(currentSong,playbackState)
```

These new axioms will guarantee that, when the Ipod is stoped (**playback = stoped**) or paused (**playback = paused**), and an action to unplug the headsets is carried on, then the headsets will be unpluged (**headsetsState**=**unplugged**) and the current song and playbackState states will be kept. Doing this prevents the variables **currentSong** and **playbackState** from non-deterministically assuming values.

After changing the interactors model in the Model Editor, and compiling it again in the Properties Editor, we can go back to WildAniMAL to perform a new simulation. Doing that simulation, we can see that now the situation "Too Many States" no longer appears and that the Ipod model has the expected behaviour. That can be verified, looking to the state based diagram of the new simulation, in Figure 47.

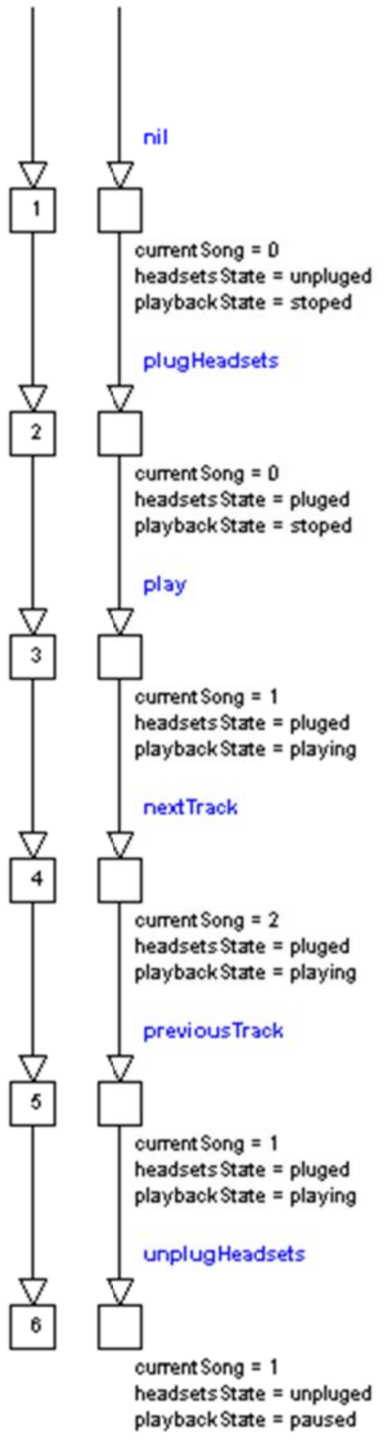


Figure 47 State Based representation of a trace originated in a simulation.

7.2. Conclusion

This chapter presented an interactors model of an Ipod-like device. It was shown how that model can be created, compiled to a NuSMV specification and simulated with WildAniMAL.

We demonstrated how WildAniMAL can be useful in the task of detecting bugs and errors in an interactive manner. The situation (error in the model) that was presented and corrected was representative of other similar situations that can occur in other models.

What is important to retain is that we can easily validate an interactors model and see if it behaves how we expect it too. If that does not happen, then we can use WildAniMAL functionalities to find out why. Doing this early validation is useful, because we can construct the interactors model incrementally by validating some steps at a time, instead of creating the big model and verify it as one.

Chapter 8 – Conclusions and Future Work

This chapter summarizes the work done and all results achieved. Some future work can be done in order to improve this WildAniMAL plugin, and therefore the aspects that can be worked on are also presented.

8.1. Goal

The goal of this work was to develop a plugin to help an IVY Workbench user while creating an interators model to interactively explore its behaviour: that is, enable the user to manually trigger events and observe how the model evolves. WildAniMAL (Watch It vaLiDation Animator for MAL) can perform this. It assists the modelling and analysis process, by providing functionalities to simulate and validate the model being created.

8.2. Results

In order to implement the plugin, three possibilities were studied (see Chapter 5):

a. representing a MAL interactors model as a Finite State Machine (FSM) and use that to drive the animation;
b. use the BDD (Binary Decision Diagrams) representation of the MAL interactors model, created by the NuSMV model checker, to perform the animation;
c. use the NuSMV model checker simulations commands, available on its interactive mode, to perform the animation.

After an analysis of the different alternatives it was decided to use the NuSMV's simulation capabilities in the implementation of WildAniMAL. The implementation is described in Chapter 6.

The implemented plugin supports the animation of models as intended. At each step the user can select one of the available actions and the animator presents the state (or possible states, in case of non-determinism) resulting from that action.

During the process of implementing the plugin, problems related to existence of non-determinism in the models arose. These related both to NuSMV not generating the list of possible future states, after a transition, if the number of states in the list exceeds one hundred, and also because even if the list of possible states is less than one hundred, it might be too large for a human user to analyse it. These issues were solved with the introduction of constraints to delimit the effect of actions in the state of the system, and thus reduce non-determinism in the simulation.

A first result of this dissertation is that the goal of this work, recalled in the previous Section, has been achieved as is demonstrated by the example presented in Chapter 7.

A parallel result of the work, that is not apparent in the thesis, but is nevertheless important for the IVY workbench development project, was the improvement of the existing plugins. When implementing the WildAniMAL plugin, some parts of the code of the CoreSystem and of its plugins were improved, and now more efficient data structures are used. The tool was developed in 2006 and since then many developments and changes were introduced in the Java language. Two examples of it, relating to data structures, are the use of ArrayLists instead of Vectors, and HashSets instead of HashMaps. Also, when detected, some parts of the code were rewritten, to be more easily understood or simply because minor bugs could happen as the code had some minor faults. Other times the improvement was to clear code, as some redundancy was present.

Another result achieved is that the IVY Workbench was extensively tested, because that was needed to test the WildAniMAL's implementation and usage. That enabled the detection of some situations when it did not work as expected, and demanded a need to correct the interoperability of all the plugins and the CoreSystem of the tool. That was done. Developing WildAniMAL also enabled us to think about how the functionalities were initially implemented and how they could be improved. This is specifically true in some aspects of usability.

Another important result is that WildAniMAL implementation was fully documented, because UML diagrams were created that describe its architecture and also because a detailed code explanation was carried out. This results leads the way to its future improvement, as enables any person to, relatively easily, understand its implementation and, if desired, improve its functionalities and source code.

8.3. Future Work

As future work, a more efficient (or automatic) integration of the WildAniMAL plugin in the IVY Worbench can be performed. Some steps of using it, require the use of other plugins of the IVY Workbench tool. The use of the Model Editor plugin is obviously a requirement to build the models, but using Properties Editor to compile the model created in the Model Editor should be avoided. The user has to go there only to push a button to compile the model. That task can be automated, but needs some changes in the CoreSystem, so that the compiler might be globally available in the system.

Additionally work can be carried out in testing WildAniMAL with more examples. One way to achieve this is to make it available to the Model Checking scientific community, so that different people might benefit from its capabilities, and also contribute with their feedback to improve tool. For example, suggesting improvements and new functionalities.

References

[1] E. M. Clarke, E. A. Emerson and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.,* vol. 8, pp. 244-263, 1986.

[2] F. Paternó, "Design, Specification and Verification of Interactive Systems'94," *Proceedings of the First International Eurographics Workshop,* 8-10 June 1994.

[3] J. Campos and M. Harrison, "Model checking interactor specifications," *Automated Software Engineering,* vol. 8, pp. 275-310, 2001.

[4] K. Loer e M. Harrison, A framework and supporting tool for the model-based analysis for dependable interactive systems in the context of industrial design, 2004.

[5] K. L. McMillan, Symbolic model checking, Kluwer Academic Publ, 1993.

[6] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri, "NuSMV: a new Symbolic Model Verifier," in *Proceedings Eleventh Conference on Computer-Aided Verfication (CAV'99)*, Trento, Italy, 1999.

[7] J. C. Campos and M. Harrison, "Systematic analysis of control panel interfaces using formal tools," *XVth International Workshop on the Design, Verification and Specfication of Interactive Systems (DSV-IS 2008),* pp. 72-85.

[8] J. Campos and M. Harrison, "Modelling and analysing the interactive behaviour of an infusion pump," in *Electronic Communications of the EASST 45: Fourth International Workshop on Formal Methods for Interactive Systems (FMIS 2011)*, 2011.

[9] E. M. Clarke, "The Birth of Model Checking," in *25 Years of Model Checking*, O. a. V. H. Grumberg, Ed., Berlin, Heidelberg, Springer-Verlag, 2008, pp. 1-26.

[10] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill and J. Hwang, "Symbolic model

checking: 1020 states and beyond," in *Proceedings 5th Annual Symposium on Logic in Computational Science*, 1990.

[11] A. Cimatti, E. Clarke, F. Giunchiglia and M. Roveri, "NuSMV: a new symbolic model checker," *International Journal on Software Tools for Technology Transfer (STTT),,* 2(4) March 2000.

[12] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, R. M. e A. Tchaltsev, NuSMV 2.5 User Manual, 2010.

[13] R. Cavada, A. Cimatti, C. A. Jochim, G. Keighren, E. Olivetti, M. Pistore, R. M. e A. Tchaltsev, NuSMV 2.5 Tutorial, 2010.

[14] D. R. Wright, "Finite State Machines," 2005. [Online]. Available: http://www4.ncsu.edu/~drwrigh3/docs/courses/csc216/fsm-notes.pdf. [Acedido em 25 October 2012].

[15] A. Gill, Introduction to the theory of finite-state machines, McGraw-Hill, 1962.

[16] S. B. Akers, "Binary Decision Diagrams," *IEEE Trans. Computers 27,* vol. 6, n.º 509-516, 1978.

[17] H. R. Andersen, An Introduction to Binary Decision Diagrams, IT University of Copenhagen, 1999.

[18] R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary Decision Diagrams," *ACM Computing Surveys,* vol. 24, pp. 293-318, September 1992.

[19] D. J. a. H. M. D. Duke, "Abstract interaction objects," *Comput. Graph. Forum 12,* vol. 3, pp. 25-36, 1993.

[20] M. Ryan, J. Fiadeiro e T. Maibaum, "Sharing actions and attributes in modal action logic," *Theoretical Aspects of Computer Software,* pp. 569-593, 1991.

[21] E. Clarke, O. Grumberg e D. Peled, Model Checking, MIT Press, 1999.

[22] Group, Object Management, Unified Modeling Language: Superstructure version 2.0, 2005.

[23] J. C. Campos, J. Machado e E. Seabra, "Property patterns for the formal verification of automated production systems," pp. 5107-5112, 2008.

[24] N. M. E. Sousa and J. C. Campos, "Um visualizador de tracos de comportamento para a

ferramenta ivy. IVY Technical Report IVY-TR-5-03," October 2006.

[25] G. Mori, F. Paternò and C. Santoro, "CTTE: support for developing and analyzing task models for interactive system design," *Transactions on Software Engineering archive,* vol. 28 Issue 8, August 2002.

[26] F. Paternó, Model-Based Design and Evaluation of Interactive Applications, Springer, 2000.

[27] F. Paternó, "Task models in interactive software systems," *Handbook of Software Engineering and Knowledge Engineering,* 2001.

[28] D. Paquette, Simulating task models using concrete user interface components, 2004.

[29] N. Guerreiro, S. Mendes, V. Pinheiro e J. C. Campos, "Animal - a user interface prototyper and animator for mal interactor models," *Interação 2008 - Actas da 3a. Conferência Nacional em Interação Pessoa-Máquina,* pp. 93-102, 2008.

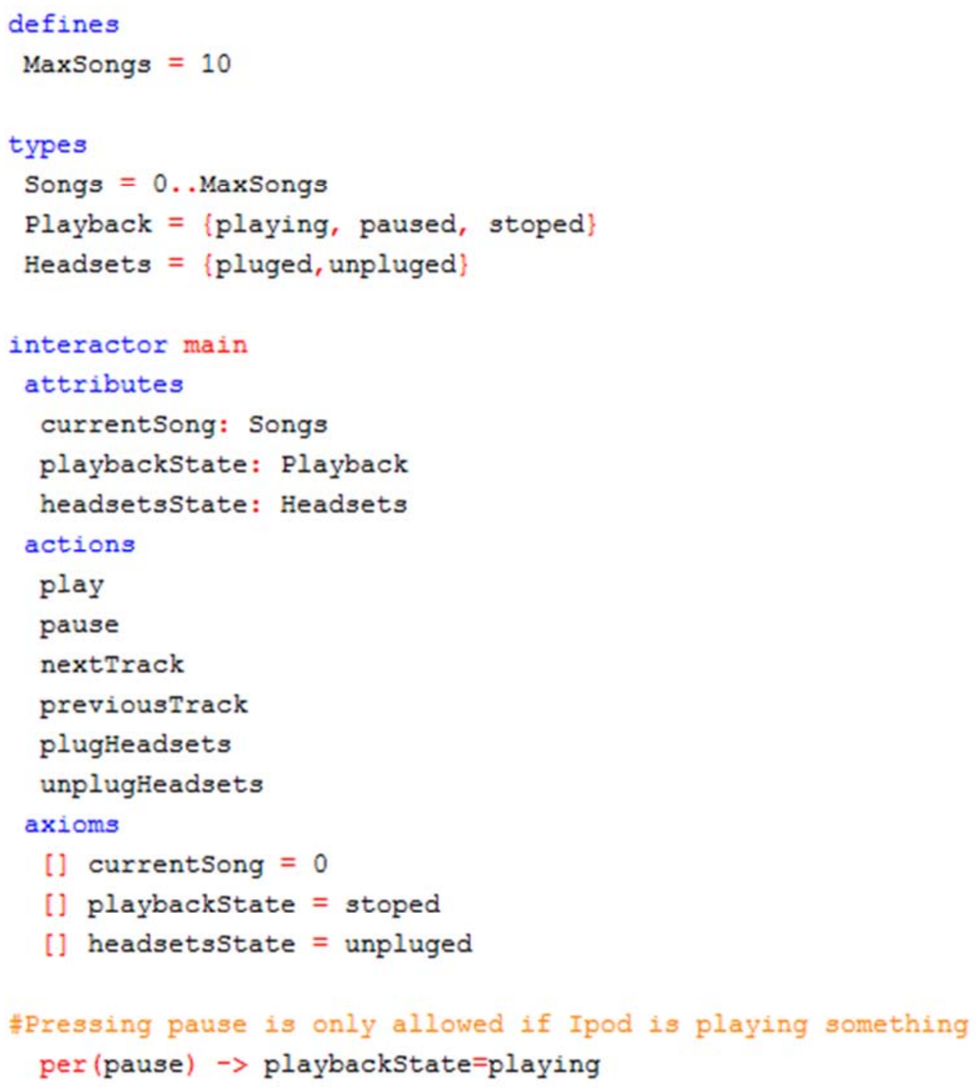
Appendix I – Build.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="Ivy WorkBench" default="help" basedir=".">

  <!– Properties :
________________________________________________________________________

       app.name - Name of application.
       app.version - Version of application.
       build.home - The directory where the built application is to be put.
       build.plugin.dev - The directory where to put the jars that are needed
                         for plug-in development.
       ipf.system - Name of jar file to generate when targeting the jars for
                   plug-in development.
  –>

  <property name="app.name" value="ipf"/>
  <property name="app.version" value="0.1"/>
  <property name="build.home" value="${basedir}/build"/>
  <property name="build.plugin.dev" value="${basedir}/dev-plugin"/>
  <property name="ipf.system" value="${app.name}-${app.version}-system.zip"/>

  <!– Paths :
____________________________________________________________________________

       classpath - The class path to use when compiling the application.

  –>

  <path id="classpath">
    <fileset dir="lib" includes="*.jar"/>
  </path>

  <typedef resource="org/java/plugin/tools/ant/jpf-tasks.properties">
    <classpath refid="classpath"/>
  </typedef>

  <!– Targets :
______________________________________________________________

        help - Show some help on building the application.
        clean - Clean the proect build folder.
        build - Compile the aplication classes.
        docs - Generate Javadocs.

  –>

  <!– Help _______________________________________________________________
–>

  <target name="help">
    <echo>
      <![CDATA[
${app.name} build file:
clean - cleans up the project build folder
build - builds entire project
run   - runs application
check - checks plug-ins integrity
docs  - generates plug-ins documentation
dist  - creates binary and source distribution packages
test  - runs some tests
]]>
    </echo>
  </target>

  <!– Clean ______________________________________________________________
–>

  <target name="clean" description="Cleans up the project build folder">
    <tstamp>
      <format property="dt-stamp" pattern="yyyy-MM-dd-HH-mm" />
      <format property="d-stamp" pattern="yyyy-MM-dd" />
    </tstamp>

    <delete dir="${build.home}" quiet="true" />
    <delete dir="${build.plugin.dev}" quiet="true" />

    <delete dir="${basedir}/plugins/CoreSystem/classes" quiet="true" />
    <delete dir="${basedir}/plugins/ModelEditor/classes" quiet="true"/>
    <delete dir="${basedir}/plugins/PropertiesEditor/classes" quiet="true"/>
    <delete dir="${basedir}/plugins/TracesAnalyzer/classes" quiet="true" />
    <delete dir="${basedir}/plugins/AniMAL/classes" quiet="true" />
    <delete dir="${basedir}/plugins/WildAniMAL/classes" quiet="true" />
    <delete dir="${basedir}/plugins/PVS/classes" quiet="true" />

    <mkdir dir="${build.home}/plugins/CoreSystem"/>
    <mkdir dir="${build.home}/plugins/ModelEditor"/>
    <mkdir dir="${build.home}/plugins/PropertiesEditor"/>
    <mkdir dir="${build.home}/plugins/TracesAnalyzer"/>
    <mkdir dir="${build.home}/plugins/AniMAL"/>
    <mkdir dir="${build.home}/plugins/WildAniMAL"/>
    <mkdir dir="${build.home}/plugins/PVS"/>

  </target>

  <!– Init _____________________________________________________________________ -
->

  <target name="-init">
    <mkdir dir="${build.home}" />
  </target>

  <!– Build PlugIns ____________________________________________________________ –>

  <target name="-build-plugins">
    <ant dir="plugins/CoreSystem" target="${target}"/>
    <ant dir="plugins/ModelEditor" target="${target}"/>
    <ant dir="plugins/PropertiesEditor" target="${target}"/>
    <ant dir="plugins/TracesAnalyzer" target="${target}"/>
    <ant dir="plugins/AniMAL" target="${target}"/>
    <ant dir="plugins/WildAniMAL" target="${target}"/>
    <ant dir="plugins/PVS" target="${target}"/>

</target>

  <!– Build the Application ____________________________________________________ –>

  <target name="build" depends="-init" description="Builds entire project">
    <antcall target="-build-plugins">
      <param name="target" value="build"/>
    </antcall>

    <copy todir="${build.home}/lib">
      <fileset dir="lib" includes="*.jar" />
    </copy>
    <copy todir="${build.home}">
      <fileset dir="." includes="*.*,**/*"
              excludes="nbproject/,todo*,build*,build/,plugins/" />
    </copy>

  </target>

  <!-- Run the Application ___________________________________________ -->

  <target name="run" description="Runs application">
    <antcall target="-build-plugins">
      <param name="target" value="build"/>
    </antcall>

    <java jar="${build.home}/lib/jpf-boot.jar"
          dir="${build.home}"
          fork="true"/>
  </target>

  <!-- Check Plugin Integrity _________________________________________ -->

  <target name="check"
          depends="build"
          description="Checks plug-ins integrity">
    <jpf-check basedir="${basedir}/plugins"
              includes="*/plugin.xml,*/plugin-fragment.xml"
              verbose="true"
              usepathresolver="true" />
  </target>

  <!-- Generate Javadocs ______________________________________________ -->

  <target name="docs"
          depends="build"
          description="Generates plug-ins documentation">
    <antcall target="-build-plugins">
      <param name="target" value="docs" />
    </antcall>
    <jpf-doc basedir="${build.home}/plugins"
            includes="*/plugin.xml,*/plugin-fragment.xml"
            destdir="${build.home}/docs"/>
  </target>

  <!-- Distribution for Plug-in Development ___________________________ -->
```

```
<target name="plugin-dev"
        depends="clean,build"
        description="Prepares Jars for Plug-in development">
  <mkdir dir="${build.plugin.dev}"/>
  <copy todir="${build.plugin.dev}" includeemptydirs="false">
    <fileset dir="${build.home}/lib"
             includes="*.jar" />
    <fileset dir="${build.home}/plugins/CoreSystem"
             includes="*.jar" />
  </copy>
  <zip jarfile="${build.plugin.dev}/${ipf.system}" compress="${jar.compress}">
    <fileset dir="${build.plugin.dev}"/>
  </zip>
  <delete dir="${build.plugin.dev}" excludes="${ipf.system}"/>
</target>

<!-- Distribution ________________________________________________ -->

<target name="dist"
        depends="clean,build,docs"
        description="Prepares distribution packages">
  <jpf-zip basedir="${build.home}/plugins"
           includes="*/plugin.xml,*/plugin-fragment.xml"
           destdir="${build.home}/plugins"/>

  <delete includeemptydirs="true">
    <fileset dir="${build.home}/plugins">
      <include name="**/*"/>
      <exclude name="*.zip"/>
    </fileset>
  </delete>

  <zip destfile="${build.home}/${app.name}-bin-${app.version}.zip"
       duplicate="fail"
       update="false">
    <fileset dir="${build.home}" includes="**/*"/>
  </zip>

  <zip destfile="${build.home}/${app.name}-src-${app.version}.zip"
       duplicate="fail"
       update="false">
    <fileset dir="${basedir}"
```

```
excludes="build,**/classes/**,.check*,.fb*,nbproject/private/**,build/**,logs/**,data/**,temp/*
*,*.zip,todo.txt,plugins/org.jpf.demo.toolbox.ftpmonitor/**"/>
    </zip>

    <delete includeemptydirs="true">
      <fileset dir="${build.home}">
        <include name="**/*" />
        <exclude name="${app.name}-???-${app.version}.zip" />
      </fileset>
    </delete>
  </target>

  <!– Run Tests. ______________________________________________________ –
>

  <target name="test" depends="build" description="Some tests">
    <jpf-pack basedir="${build.home}/plugins"
              includes="*/plugin.xml,*/plugin-fragment.xml"
              destfile="${build.home}/all-plugins.jpa" />
    <mkdir dir="${build.home}/all-plugins-extracted" />
    <jpf-unpack srcfile="${build.home}/all-plugins.jpa"
                destdir="${build.home}/all-plugins-extracted" />
  </target>

</project>
```

Appendix II – Ipod interactors model



```
defines
 MaxSongs = 10

types
 Songs = 0..MaxSongs
 Playback = {playing, paused, stoped}
 Headsets = {pluged,unpluged}

interactor main
 attributes
  currentSong: Songs
  playbackState: Playback
  headsetsState: Headsets
 actions
  play
  pause
  nextTrack
  previousTrack
  plugHeadsets
  unplugHeadsets
 axioms
  [] currentSong = 0
  [] playbackState = stoped
  [] headsetsState = unpluged

#Pressing pause is only allowed if Ipod is playing something
  per(pause) -> playbackState=playing
```

```
#Pressing play is only allowed if Ipod stoped or paused playback
  per(play) -> (playbackState=stoped | playbackState=paused)
  & headsetsState=pluged

#Unpluging headsets is only allowed if they are plugged
  per(unplugHeadsets) -> headsetsState = pluged

#Pluging headsets is only allowed if they are unplugged
  per(plugHeadsets) -> headsetsState = unpluged

#If play button is pressed, then the first song starts playing
  playbackState=stoped & headsetsState=pluged -> [play]
  (currentSong'=1) & (playbackState'=playing) & keep(headsetsState)

#Pressing NextTrack or PreviousTrack buttons preserves the current
#state, if Ipod stoped playback
  playbackState=stoped -> [nextTrack]
  keep(currentSong,playbackState,headsetsState)
  playbackState=stoped -> [ previousTrack]
  keep(currentSong,playbackState,headsetsState)

#When Ipod is playing songs and NextTrack button is pressed, and if
#the played song is the last one of the playlist, playback is stoped
  playbackState!=stoped & currentSong=MaxSongs -> [nextTrack]
  (playbackState'=stoped) & (currentSong'=0) & keep(headsetsState)

#When Ipod is playing songs and NextTrack button is pressed, and if
#the played song isn't the last one of the playlist, the next song
#in the playlist is played
  playbackState!=stoped & currentSong<MaxSongs -> [nextTrack]
  (currentSong'=currentSong+1) & keep(playbackState,headsetsState)

#When Ipod is playing songs and PreviousTrack button is pressed, and
#if played song is the first one of the playlist, playback is stoped
  playbackState!=stoped & currentSong=1 -> [previousTrack]
  (playbackState'=stoped) & (currentSong'=0) & keep(headsetsState)

#When Ipod is playing songs and PreviousTrack button is pressed, and
#if the played song isn't the first one of the playlist, the previous
#playlist song in is played
  playbackState!=stoped & currentSong>1 -> [previousTrack]
  (currentSong'=currentSong - 1) & keep(playbackState,headsetsState)
```

```
#If Ipod is playing something and pause button is pressed, it stays
#in pause
  playbackState=playing  -> [pause] (playbackState'=paused) &
  keep(currentSong,headsetsState)

#When headsets are unpluged and Ipod playbackstate is paused, Ipod
#starts playing current song
  playbackState=paused  -> [plugHeadsets] (headsetsState'=pluged) &
   (playbackState'=playing) & keep(currentSong)

#When headsets are unpluged and Ipod playbackstate is stoped, Ipod
#keeps that state
  playbackState=stoped -> [plugHeadsets] (headsetsState'=pluged)
  & keep(currentSong,playbackState)

#If Ipod is playing and headsets are unpluged, then playbackState
#changes to paused
  playbackState=playing -> [unplugHeadsets] (playbackState'=paused)
  & (headsetsState'=unpluged) & keep(currentSong)
```