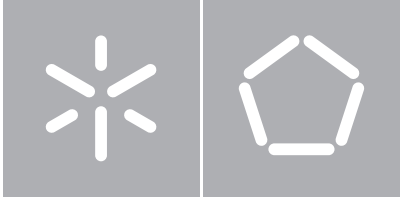
Universidade do Minho
Escola de Engenharia

Manuel António Freitas de Sousa

Safety Critical Interactive Computing Systems' Modelling

Novembro de 2012



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Manuel António Freitas de Sousa

Safety Critical Interactive Computing Systems' Modelling

Dissertação de Mestrado
Mestrado de Informática

Trabalho realizado sob orientação de

Professor José Creissac Campos
Dra. Miriam C. Bergue Alves

Novembro de 2012

Acknowledgements

I would like to thank everyone who helped me during this time.

I would like to thank to my supervisor, Professor José Creissac Campos (Universidade do Minho) and my co-supervisor Miriam C. Bergue Alves (Institute of Aeronautics and Space · Brazil).

Special thanks to my mother, my father and my brother Bruno, for supporting me and giving motivation, especially when it tends to disappear.

Resumo

Normalmente, testar um sistema interativo envolve testar manualmente as possíveis interações com ele. Uma vez que esta é uma abordagem manual, torna-se muito caro verificar todas as interacções possíveis. Em sistemas interactivos de segurança crítica esta tarefa é essencial. Uma forma de ultrapassar este problema é a utilização de ferramentas de análise sistemática. IVY Workbench é uma dessas ferramentas. Pretendemos aplicá-la para realizar a verificação de um Sistema Interactivo de Segurança Crítica. Os objetivos para esta dissertação são: desenvolver um conjunto de modelos de sistemas interactivos de segurança crítica, verificar propriedades relevantes dos modelos, fazer uma avaliação crítica do processo de modelação e sugerir melhorias para a ferramenta e linguagem.

Abstract

Typically, testing an interactive system involves manually testing their possible interactions. Since this is a manual process, it becomes very costly to check all possible interactions. In safety critical interactive systems this task is essential. One way to overcome this problem is to use tools for systematic analysis. IVY Workbench is one of these tools. We plan to apply it to perform verification of Safety Critical Interactive Systems. The objectives for this dissertation are: development of a set of models of safety critical interactive systems; verification of relevant properties of the models; critical assessment of the modelling process and suggestion of improvements to the tool and language.

Contents

List of Figures

List of Tables

Glossary

B

BDD Binary Decision Diagram, p. 10.

BNF Backus Normal Form or Backus-Naur Form, p. 7.

C

CR Electric Control Network, p. xi.

CTL Computation Tree Logic, p. 5.

E

EV Flight Events Sequence Network, p. xi.

G

GUI Graphical User Interface, p. 7.

I

IAE Institute of Aeronautics and Space - Brazil, p. 3.

ICO Interactive Cooperative Object, p. 5.

L

LTL Linear Temporal Logic, p. 5.

M

MAL Modal Action Logic, p. 4.

MSI Inertial Sensor Module, p. 55.

O

ObCS Object Control Structure, p. 6.

S

SAL Symbolic Analysis Laboratory, p. 9.

SMT Satisfiability Modulo Theories, p. 10.

SOAB-CDB Onboard Flight Control Software-VLS Onboard Computer, p. 55.

T

TPGS Testing and Preparation Ground System, p. 3.

V

VEG Visual Event Grammars, p. 7.

X

XML Extensible Markup Language, p. 9.

Chapter 1

Introduction

Nowadays interactive systems are everywhere, so problems that resulted from interactions with these systems appear more frequently. As systems becomes more complex, and with more features, interaction with them becomes more complex too. Like any other piece of software, interactive systems need to be analysed to guarantee their quality. In this kind of systems, evaluation can be performed, for example, with respect to how easy the system is to learn, how perceivable the presented information is, or how predictable the system is. However, it must also be considered whether the system prevents errors, and always behaves correctly.

We are interested in techniques that systematically verify Interactive Systems with respect to safety. By safety we mean that the system is free from errors and we want to establish that all possible interactions do not trigger any error or erroneous behaviour. To achieve that, we focused in the exhaustive analysis of all possible system behaviour.

Several techniques are available to perform evaluation of interactive systems ranging from empirical studies with users to inspections performed by usability experts (see [11] for a good coverage of this topic). Typically the techniques are not systematic and they consume too many resources (usually time and financial cost) to achieve quality results. Besides, the studies results cannot guarantee safety, since the techniques are not exhaustive in their analysis. To achieve exhaustive verification of Interactive Systems, we will analyse formal methods that allow it applicability, for systematic verification. This will guarantee that if a system's property is proven/demonstrated, then we can be sure that it always holds for that system. Such warranty is required since, we will apply this kind of analysis

to Safety Critical Interactive Systems.

1.1 Motivation

Evaluation of interactive systems should be done during the development cycle, and not at its end. In chapter 9 of their book on Human Computer Interaction, Dix et al. [11] discuss several approaches to the evaluation of Interactive Systems. They distinguish evaluation techniques according to who is involved: the designer/usability expert or the user. Some techniques are carried out only by designers/usability experts, others need common users to be performed.

Evaluation of an Interactive System may have several objectives, which can be evaluate the reachability of system functionalities, evaluate the users' interaction experience, or identify any specific problem with the system. The book describes several techniques like cognitive walkthrough, which main goal is to establish how easy a system is to learn, heuristic evaluation, experimental evaluation, observation, queries and evaluation through monitoring physiological responses. Some of them are performed by experts. Others, analysis of data obtained through user experiences with the system. In an experimental evaluation, an evaluator chooses one or more hypothesis to be tested, and experiments are performed with a significant number of potential target users and the data from the experiments is collected for later analysis. This technique can be seen as bug finding, in the sense that, errors are "caught" in the collected data of the experiences. However, bug finding is just not enough to perform verification in Safety Critical Interactive Systems, as the user could miss some critical interaction with the System.

An Interactive System is safety critical when its failures can lead to loss of human lives, or loss of a large amount of money, or even both. Examples of such systems can be found in aeronautics and space, aviation, banking, health or even sports. Sport might not be an obvious area, but safety-critical interactions exists mostly in motorsport systems. For example, a steering wheel of a Formula 1 (F1) car, or a telemetric monitoring system, are Interactive Systems whose failure could cause loss of lives (pilot or even audience) and loss of quite large amount of money (a top team F1 car may cost 2-3 million euros).

Another problem with "traditional" techniques is that they require a considerable amount of usability expertise (knowledge of the principles and guidelines),

time (to observe user interactions) and resources (users to perform experiences), to get accurate results, and that they cannot be performed automatically. All the techniques presented above cannot verify Interactive System, they just give indications at the level of usability and proper operation of what was tested, but not in an exhaustive manner, which is not enough for Safety Critical Interactive Systems.

One way to achieve systematic analysis is by using formal i.e., mathematically rigorous, methods. Formal analysis of Safety Critical Interactive Systems has been applied to a number of interactive systems in aviation and health. These analysis were able to find problems with either security [2, 16] or disrespect for good practice in interface design [18]. Some concepts like graph theory were also applied [19]. The IVY Workbench has been applied to some realistic case studies [6, 7, 8], and there is interest in exploring its applicability to Safety Critical Interactive Computing Systems in the space domain.

The system under consideration is Brazilian Aeronautics and Space Institute's (IAE) current satellite launcher Testing and Preparation Ground System (TPGS). TPGS includes several operators' interactions that enable control over the telemetry and other aspects of the preparation and launching of the rocket. Each interface screen enables interaction between the operator and the system via a graphical user interface.

1.2 Research Methodology and Goals

The main objective of this research is to apply IVY Workbench to the IAE's TPGS, which is a critical system supporting space mission preparation. The adopted research methodology follows the activities below:

- review the recent and state-of-the-art publications;
- study the system to model based on the user manuals and remote interaction with the IAE' researchers;
- develop a set of models of the IAE' TPGS;
- test critical properties over the developed models.

Although the IVY Workbench had been chosen from the outset as the tool to model TPGS, the analysis of similar tools was required in order to understand how better

and appropriate the IVY Workbench is when compared to them. Another particularity is that the system was modelled based on the user manual and not on the actual system.

The objectives of the research then are:

- to develop a set of models of the system addressing different modelling approaches;
- perform properties verification over the models;
- propose improvements and new features in both the language and the tool based on the modelling process results.

1.3 Structure of the Document

This document is organized as follows. Chapter 2 contains an analysis of some tools used to model and verify Interactive Systems. The main characteristics of the tools are described, and a comparison between them is made.

In Chapter 3, the language used to model the system under study, the MAL Interactors language, is described. The structure of an Interactor is introduced, alongside with what can be expressed with the language.

Chapter 4 introduces the system to be modelled. What it is used for, its architecture and all its subsystems. Moreover, issues to be addressed in the modelling process are described.

In Chapters 5 and 6 the modelling of two TPGS's subsystems, Flight Events Sequence Network (EV) and Electric Control Network (CR) respectively, is described. The obtained models are described step by step and analysed.

In Chapter 7 an analysis of both the models and the tool is made. Concerning the analysis of the models, some verified properties are shown. Data on the verification times obtained in the two machines that performed the tests is analysed. The tool analysis is related to the language improvements and tool changes that can be made in order to reduce the modelling and verification times, and to make the tool more versatile.

The document ends in Chapter 8 with a synthesis of the work done, the conclusions obtained, and the future work to be carried out.

Chapter 2

State-of-the-art

In the previous chapter we have mentioned that typical user interface evaluation techniques are usually not automated and can take a considerable amount of time and resources to apply. Now, we will consider some techniques/formalisms that use model checking to perform automated and exhaustive verification of Interactive Systems.

Model checking is a technique for formally verifying systems specified as finite-state machines [9]. Desirable properties of the behaviour of the system are expressed as temporal logic formulas [10] (e.g. LTL, CTL), and a traversal of the finite-state machine is performed in order to check if certain defined property holds or not.

Four tools will be considered: PetShop, which uses Petri nets; the VEG toolkit, which uses grammars; SAL (Symbolic Analysis laboratory), a framework that combines different verification tools; and the IVY workbench, which uses Modal Action Logic to specify the models.

2.1 *PetShop*

PetShop[1] is an environment to edit, verify, and execute Interactive Cooperative Objects (ICO) models. Palanque proposed the ICO [15] formalism, which is intended to specify interactive systems. ICO is an object-oriented formalism, that can be seen as a class, divided in four different parts:

[1] http://www.irit.fr/recherches/ICS/softwares/petshop/

- a Data structure which is a set of typed attributes;
- a set of Operations that are called services (part of that set of operations are also the internal operations that can only be invoked by the object that defines them);
- an object Object Control Structure (ObCS) that defines the behaviour of the ICO;
- and a Presentation specified by a set of *widgets*, that connect the user interaction with the object's user services.

The ObCS, besides defining the behaviour of the ICO, defines all behaviour regarding to the services, such as service availability, service requests processing, services required as a client, and own operations. To represent all these features a *high-level Petri Net* is used. The *Places* of the *Petri Net*, are seen as the states of the system, and the *Transitions* as the services of the ICO. User interactions, called user services, are represented as ellipses, linked to the corresponding *Transition*.

Presentation as mentioned above is a set of *widgets*, and is the only place where the interaction between user and system is performed, so each *widget* could activate an user service. Each window of an interactive system, is modelled by one ICO and so, the synchronization between ICO's services are defined in the ObCS.

From this specification it is possible to automatically verify the model. Several tools are available to verify CTL/LTL formulas in *Petri Nets*, and some interesting properties can be proved (for example, the absence of deadlock). However, because the system is represented in more then one ICO, we need to verify more then one ICO at the same time. This formalism has been applied to various kinds of systems including a plane cockpit display system, an example of a Safety Critical System [1].

One problem in this tool/formalism, and generally in model checking, is the *state space explosion* problem. The number of states needed to represent a System can become massive, making it non viable to analyse all the states either due to processing time/power demands, or due to memory demands. For small systems this problem does not exist, but a real system may have a very large number of states, or even an infinite number. PetShop uses a model checker that represents the states explicitly, so the verification of properties can become too costly, or simply not viable.

2.2 *VEG toolkit*

To resolve the problem of state explosion, Berstel et al.[3] proposed VEG toolkit, which, instead of using *Petri Nets*, use an extension of BNF grammars to specify GUIs. The idea is to decompose the specification of a large GUI interface into a communicating automata, which reduces the state space [3]. A VEG object represents an automaton and is specified by a grammar representing a small part of the GUI. The specifications are independent of the layout of the GUI, so there is a separation between presentation and behaviour, which makes the reuse of the same behaviour in different presentations possible. The grammar of a VEG object specifies the behaviour, and communication between components by a set of rules, typically with the following format:

<current state> ::= <guard> <communication> <visual action>
<goto next state>

Each rule specifies the behaviour of a component in a state. The <*current state*> is the state we are specifying, the <*guard*> is the event or set of events that "trigger" the rule, <*communication*> is where we send events to other components, and <*visual action*> is where we specify the visual changes. These changes are programmed independently from the grammar rules, based on the particular presentation toolkit.

Besides the separation of presentation from behaviour, VEG has separation of data and control. As defined above, a specification in VEG is composed by a part that describes the event-driven behaviour of the GUI, and other part describing the manipulation of data. Sometimes these separations need to be bridged. Consider, a simple GUI that reads a number, and shows a mark which turns green or red depending on whether the number is even or odd, respectively. The routine for checking the number is independent from the GUI design, but the result influences the behaviour of the GUI. To model this kind of routines VEG uses *Semantic Functions* which are collected in the Semantic Library and are written in a programming language. The result of these functions can be passed to visible actions.

As with the ICO formalism, the verification of VEG specifications is done using model checking. Because a specification in VEG is not an automaton *per si*, or some structure to which model checking can be directly applied, a translation of the specification to a model that can be verified by a model checker is needed.

Besides that, VEG was designed in a way that makes it easy to translate the specification into an abstraction in order to reduce the state space. VEG toolkit[2], the tool used for specification, verification, design and implementation of graphical user interfaces in VEG, translates VEG specifications to Promela, the language of the SPIN model checker. With SPIN we can test consistency and correctness, detect deadlocks and unreachable states, and generate test cases for validation which is a new feature that we do not have in ICO. The properties to check can only be specified using LTL.

Because the Promela specification is an abstraction of the original VEG specification, we face new problems: *Soundness* and *Completeness*.
For a program P, an error location ε, and an algorithm $\mathcal{A}$ that performs abstraction of a system/program:

- $\mathcal{A}$ is sound iff $\mathcal{A}(P, \varepsilon) =$ "safe" then P is safe w.r.t. ε.
- $\mathcal{A}$ is complete iff P is safe w.r.t. ε then $\mathcal{A}(P, \varepsilon) =$ "safe".

For this particular case we say that, the algorithm that abstracts the VEG specification to a Promela specification is sound if and only if when some property is true (or false) in the Promela specification then the same property is true (or false) in the VEG specification. The algorithm is complete if and only if when some property is true (or false) in the VEG specification then that property is true (or false) in the Promela specification.

From the undecidability of the halting problem [4], we know that there is no sound and complete $\mathcal{A}$. So there are two approaches: *falsification* and *verification*. Falsification is the exploration of a subset of states of the original specification, so this approach is like a bug finder in the sense that, if we find an error it exists in the original specification. On the other hand, if it is not possible to find the error in the abstraction, we can say nothing about the absence of that error, in the original specification. Verification consists on the exploration of a superset of states of the original problem. In this case if we do not find an error it actually do not exists in the original. On the other hand, if we find an error, that error could be actually a genuine error, that can be reproduced in the original specification, or could be spurious, i.e., the error is not feasible in the original specification.

[2] http://home.dei.polimi.it/campi/veg/

Apart from these problems model checking can be applied for verification, and VEG formalism was applied to several systems, like an editor similar to Notepad, a graph construction library and a large real medical software application. More than verifying properties, the VEG toolkit can generate Java code of the specified GUI. This is a major difference between the previous formalism, and makes the modelling useful to implement the GUI, which is a good way to attract the industry to use the tool.

2.3 *Symbolic Analysis Laboratory*

In many examples we might need to represent many numerical variables, and in some cases we might need to represent real numbers with no approximation. SAL is a formalism/tool developed at SRI International[3] that has a unique "infinite" bounded model checker, capable of performing verification in such systems. SALenv is a framework for combining different tools for abstraction, analysis, theorem proving and model checking of transition system. SALenv has an intermediate language called SAL to specify transition systems, which is similar to input languages used in other verification tools like SMV[4], or Murϕ[5]. The abstract syntax tree of SAL is specified in XML, which makes this language easy to translate to other modelling and programming languages.

A SAL model is a context, and a context is a container of types, constants, and modules (which represents a transition system). A type can be an enumeration or an infinite set like $\{x : REAL \mid x > 0 \quad AND \quad x < 10\}$. Constants can be seen as global variables of that context, that cannot be changed. A module is where the specification of the transition system is in. Inside a module we have one or more input, output, local, or global variables, an initialization "zone" where we can define initial values for the variables and a definition area where we can define invariants. Still inside a module, we have a transition "zone" where we define the transitions ("steps") of the system. In Fig. 2.1 we can see an example of a SAL context, that nondeterministically switches its state from ready to busy and vice-versa. Being this language intended to be platform independent, due to its

[3] http://sal.csl.sri.com/
[4] http://www.cs.cmu.edu/ modelcheck/smv.html
[5] http://www.cs.utah.edu/formal_verification/Murphi/

```
short: CONTEXT =
BEGIN
  State: TYPE = {ready, busy};

  main: MODULE =
    BEGIN
      INPUT request : BOOLEAN
      OUTPUT state : State
      INITIALIZATION
        state = ready
      TRANSITION
        state' IN IF (state = ready) AND request THEN
                    {busy}
                  ELSE
                    {ready, busy}
                  ENDIF
    END;
END
```

Figure 2.1: SAL context example.

abstract syntax being specified in XML, it is easy to apply several model checkers to a SAL model. In fact, SALenv has several model checkers with many approaches with respect to how the model is abstracted and represented (i.e., BDD-based model checker, SAT Solving, SMT Solving, etc). While there is the possibility to check properties with LTL and CTL, the current version of SAL framework does not print counterexamples for CTL properties, and can only verify CTL properties that can be translated to LTL. In fact, the verification of CTL properties is still experimental.

Another feature present in this framework is the *Path Finder*, which is a random trace generator. Giving a context and a module of a SAL specification, the *Path Finder* randomly generates a trace, where the number of transitions of the generated trace can also be specified. Like the VEG formalism, SALenv has a automated test generator that generates test cases.

One good feature, that PetShop also has, is the possibility of performing simulation over the models. This is a good feature to test specific sequence of actions in order to perform a first validation of the model. Something that in traditional temporal logics, like LTL or CTL, is hard to achieve. The simulator available in SAL framework is a command line-like interface.

2.4 *IVY Workbench*

Campos et al. have proposed the IVY Workbench, a similar tool to the previous one, which supports a modelling, verification and analysis cycle of interactive systems. In this tool the *Interactor* is the main concept used to structure the models. The *Interactor* concept (Faconti and Paternò, 1990; Duke and Harrison, 1993) was proposed to help designers focus on key issues in interactive systems. An *Interactor* is a kind of an object capable of representing its state in any rendering environment. Several formal specification languages have been used to specify *Interactors*, like *Z* and *VDM*. As SAL, IVY Workbench has an intermediate language to specify a model, which is an adapted version of Modal Action Logic (MAL) [17].

The definition of an *Interactor* has three main components:

- State;
- Behaviour;
- Render.

The State is represented by a set of typed attributes. *Actions* are used to manipulate the State of the *Interactor*. The Behaviour is described using an extended version of Structured MAL. In addition to the usual propositional operators and actions the logic provides a modal operator, two deontic operators, and a special reference event to specify the initialization of the state variables. The modal operator defines a relation on the current and next values of attributes, given a particular action. The rendering relation of the *Interactor* is defined by using the annotation *[vis]* before an action or an attribute. The annotation before an attribute shows that the attribute is visibly perceivable, when attached to an action, that it can be invoked by the user. The composition of *Interactors* is done using aggregation. The actions and attributes of an aggregated *Interactor* are always accessible to the *Interactor* that aggregates it.

A compiler, *i2smv*, translates the MAL Interactors model to a Symbolic Model Verifier*SMV* specification, in order to perform model checking. IVY Workbench has some features to perform the verification and analysis. The tool can be divided in three main components (features):

- *ModelEditor*;

- *PropertiesEditor*;
- *TraceAnalyzer*;
- *WildAniMAL*.

The *ModelEditor* is the component where we write our models using MAL *Interactors*. Model editting can be done in plain text or using a GUI interface. The *PropertiesEditor*, as its name suggests, is the section where we can define the formulas. Formulas can be written in text, as in the *ModelEditor*, or they can be constructed, using a *GUI Wizard*, from a set of patterns available in the tool. For the formulas that have been checked false, the model checker generates a trace of a counterexample. The *TraceAnalyzer* is the component that helps the designer understand why the formulas are checked false. From a formula checked as false, the *TraceAnalyzer* gets the trace of the counterexample, and then shows it using a number of different representations. The *WildAniMAL* is the component where the model can be simulated. It helps to test the behaviour of the model by experimentation, which is useful to test simple interactions.

2.5 Conclusion

From the analysis above we have seen that all the tools can verify LTL formulas, and only PetShop and IVY can verify CTL formulas. SAL can verify CTL formulas too, but with a particularity, only CTL formulas that can be translated to LTL. Another difference between the tools relates to the verification techniques that they provide. All the tools verify properties using the model checking technique. The PetShop performs model checking over Petri Nets, the VEG toolkit over a Promela specification and IVY over a SMV specification. The SAL env performs model checking through SAT and BDD-based model checkers. In SAL env is also possible to verify properties through theorem proving with SMT solvers. That makes the SAL env more versatile, specially to verify bigger systems, because theorem provers are faster than model checkers [14] (although it must be noted that this is done at the expense of the fully automated nature of model checking, in particular in the case of liveness). Table 2.1 summarizes the above discussion.

Formalism/Tool	Specification	Abstract Representation	Verification
PetShop	Petri Net	Petri Net	LTL, CTL
VEG toolkit	Grammar	Promela Specification	LTL
SAL env	SAL Language	SMT, SAT, BDD	LTL
IVY Workbench	MAL Interactors	SMV Specification	LTL, CTL

Table 2.1: Summary of the analysis.

We have seen a number of approaches but the same methodology for verifying an interacting system: design a model of the system and perform verification, either by checking properties (LTL, CTL, etc) or by simulation. One problem with checking properties is that they can be hard to specify. The problem with simulation is that it can take a long time, and we can not be assured that we have gone through all the states. What if we could check a property and simulate at the same time? For example, follow a path of execution, and verify the validity of a certain property. Or check the validity of a certain property, where the user can only use some predefined actions. That kind of analysis and modelling is a task analysis-related, and was used with several formalisms and tools, like ICO, SAL and IVY Workbench. In this approach we restrict/guide the analysis trough a predefined behaviour of the user. This analysis is hard to be done just using temporal logic, and in some cases is just impossible, due to the limited expressiveness of the logics. In ICO and IVY Workbench the "starting point" is the model of the system. Now we need to create a user task model and then perform the typical analysis with the temporal logic. Good examples of this kind of modelling and analysis in ICO and IVY Workbench can be found in [15, 5], respectively. In ICO and IVY Workbench the separation between system model and user model is clear. This enables us to make both analysis without overcomplicating the system model. In IVY we have both models as different *Interactors*, and only need to create an *interactor* that will aggregate them and perform the orchestration between them. In ICO the user model only needs to link the user action to the actions in the system.

Chapter 3

Modelling in the IVY wokbench

Section 2.4 provided a brief introduction to the IVY workbench. This is the tool that will be used in the remaining of this dissertation. As such, this chapter provides more details regarding the modelling languages supported by the tool. Two languages are considered: on one side, MAL interactors, the language used to describe the interactive systems; on the other, CTL, the language used to express properties for verification.

3.1 The MAL Interactors language

As described in section 2.4 the main concept used to structure MAL Interactor models is the Interactor, which has three main components: state, behaviour and rendering. Figure 3.1 shows two of these components (state and behaviour) in the *SIN* Interactor.

The state is represented by typed attributes, as in the example. MAL Interactors have two base types only, the *boolean* and enumeration types. In figure 3.1 *Colour*, *Characteristic*, *Id* and *Int* are enumerated types. Enumerations, as is to be expected, are defined by explicitly listing all the possible values of the type. When the values are sequential (integer) numbers, the definitions can be simplified by using ranges that indicate the first and last number in the sequence only. Hence, type *Int* was defined as *0..5*, which is is equivalent to explicitly defining the enumeration {*0,1,2,3,4,5*}. Besides declaring its own attributes, an interactor can aggregate other interactors, thus allowing structuring the model (see the aggregates clause in the figure).

```
types
 Colour = {yellow, red, green}
 Characteristic = {Fixed, Blink}
 Id = {BD1A, tensaoBD1A}
 Int = 0..5

interactor SIN
 aggregates
  sinVariable via BD1_A
  sinVariable via tensaoBD1 A
 attributes
  automatic: boolean
  colour: Colour
  characteristic: Characteristic
 actions
  setValue2(Id,Int)
  setAutomatic
  setManual
  recAlarAler(Id)
 axioms
  #init
  [] automatic = true

  #BD1A
  effect(BD1_A.setValue(_v)) -> effect(setValue2(BD1A,_v))
  per(setValue2(BD1A,_v)) -> effect(BD1_A.setValue(_v))
  #tensaoBD1A
  effect(tensaoBD1_A.setValue(_v)) -> effect(setValue2(tensaoBD1A,_v))
  per(setValue2(tensaoBD1A,_v)) -> effect(tensaoBD1_A.setValue(_v))

  [recAlarAler(BD1A)] keep(automatic)
  [recAlarAler(tensaoBD1A)] keep(automatic)
  [setValue2(BD1A,_v)] BD1_A.value' = _v & keep(automatic)
  ..
  ..
```

State
Behaviour

Figure 3.1: Interactor example.

Behaviour is defined by a set of actions, in the example they are declared below the attributes. The behaviour of the actions is defined using axioms in MAL logic. These axioms define the semantic in terms of their effect on the state of the Interactor. In addition to the usual propositional operators, the logic provides:

- a modal operator [_] _, which means that if [*ac*] *expr* then *expr* is *true* in all states reached by the action *ac*;
- a special reference event [], that means, if [] *expr* then *expr* is *true* in the inicial state(s);
- a deontic operator *per*, means that if *per*(*ac*) then action *ac* is permitted to happen next;
- a deontic operator *obl*, means that if *obl*(*ac*) then action *ac* is obliged to happen some time in the future;

- operator *effect*, means that if *effect*(*ac*) then action *ac* happen at the exact moment

The special reference event [] is used to define the initial values of the attributes, as we can see in line 59 of figure 3.1. In this case, it is being stated that attribute *automatic* has the value *true* in the initial state.

The modal operator is used to define the behaviour of an action, i.e. the effect that the action has on the attributes of the Interactor(s). In the *expr* part of the modal operator, the next state of an attribute is represented by a prime, as seen in line 70 of the figure 3.1.

Moreover when we want to express that certain attribute remains unchanged the *keep* operation is used, as seen in line 68 of the figure 3.1. In this case the axiom states that when action *recAlarAler* happens with value *BD1A* as parameter, the attribute *automatic* remains unchanged.

The expressions in lines 68 and 69 are particular cases of the generic expression:

[*Action*] *Consequence*

This means that when the *Action* occurs the *Consequence* happens. From this expression and making use of an implication another generic expression can be defined:

Condition –> [*Action*] *Consequence*

Similar to the previous but now the *Consequence* only happens, if the *Action* occurs and the *Condition* is true.

In some situations more than one action have the same consequence. To reduce duplications in the model it is possible to use the following “syntactic sugar”:

[*ac1*|*ac2*|*ac3*] *Consequence*
[!(*ac1*|*ac2*|*ac3*)] *Consequence*

The first line means that when the actions *ac1* or *ac2* or *ac3* occurs the *Consequence* happens. The other line means that when any action in the Interactor with the exception of *ac1*, *ac2*,*ac3* occurs the *Consequence* happens.

An action can have parameters as is the case of actions *setValue2* and *setAlarAler* in figure 3.1. Sometimes the definition of an action depends on the values that are received in the parameters.

[*ac*(_v)] *Condition_on_v –> Consequence*

This previous expression means that when action *ac* occurs with values on the parameters that obey *Condition_on_v*, then *Consequence* happens.

The deontic operators are used to express permissions.

per(*Action*) –> *Condition*

This previous expression means that the *Action* is only allowed to occur if the *Condition* holds. An example of this kind of expression is in line 63 of figure 3.1. That expression means that the action *setValue2(BD1A,_v)* (where *_v* is implicitly universally quantified) is only allowed, when the action *BD1_A.setValue(_v)* is happening.

The *obl* operator is used in similar expressions.

obl(*Action*) –> *Condition*

This means that if the *Condition* is true, then the *Action* must happen in the future. With all these expressions it is now possible to create an entire model.

3.2 CTL – Computational Tree Logic

The language to specify the properties for verification is CTL [10], a branching time logic. The behaviour of a system over time is seen as a tree-like structure (figure 3.2) in which at each moment in time, represented as a node in the tree, there are several possible futures (the branches sprouting from the node). Hence, a concrete behaviour of the system, i.e., a computation path will be a possibly infinite path going down the tree from its root.

Besides the classical logic operators the CTL language has path quantifiers and temporal operators. Temporal operators are:

- *A p* - meaning that p holds for all computation paths;
- *E p* - meaning that p holds for some computation paths.

These operators are used to describe the branching structure in the computation tree. The temporal operators are:

- *X p* - meaning that p holds in the next state;

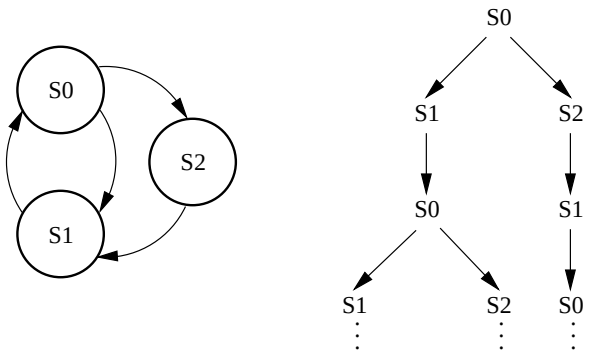


Figure 3.2: Branching Time.

- *F p* - meaning that eventually (i.e., in the future) p holds;
- *G p* - meaning that p always holds;
- *p U q* - meaning that q eventually holds and until then p always holds;

These operators are used to describe properties of a path through the tree.

With all these operators we can specify a wide range of properties, including liveness and safety properties. A liveness property asserts that something good does happen eventually. A safety property asserts that something bad does not happen.

In our study case, because most of the properties are defined to check that what was modelled complies with the manual, we will want to specify properties that can give us counter examples to check if the system is behaving as expected.

3.3 Conclusion

This section has introduced the languages used for modelling (MAL interactors) and expressing properties for verification (CTL) in the IVY workbench, the tool which will be used in the remainder of this dissertation. More details about IVY and its modelling language can be found in [6, 8]. For more details about CTL, see, for example, the book by Clarke et al. on model checking [9].

Chapter 4

TPGS – a Case Study

The Testing and Preparation Ground System (TPGS) is a space ground legacy system that has been in use to support space mission preparation for more than 15 years in the Brazilian Space Launcher Program. TPGS' software components are often updated due to new missions requirements, different rocket configurations, and new operating systems releases. The hardware is occasionally upgraded due to the obsolescence of the TPGS' equipment.

As main contractors are in charge of the upgrade activities, there is a need to provide a set of user interface requirements for correct mission-safety system operation as the system evolves due to new updates. The goal then, is to study the feasibility of modelling and analysing the user interface of the system with the IVY tool. This will allow the requirements to be formally modelled and verified against critical properties, and used as a requirement baseline (an oracle) for the TPGS during the acceptance phase of new versions of the system.

4.1 Overall architecture of the System

TPGS is a safety-critical system, composed of very specific and customized hardware and software components, responsible for the ground control, testing and preparation of a satellite launch rocket before it is launched. TPGS includes six different subsystems that provide specific functionality during the rocket's testing and preparation for launching:

PW subsystem: rocket's power supply, charging and discharging of the rocket's

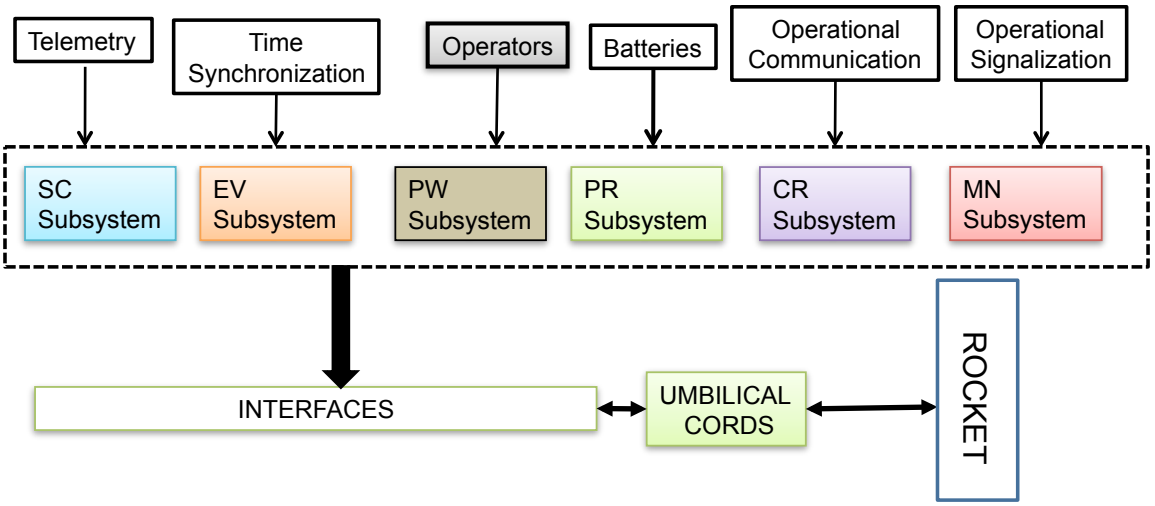


Figure 4.1: The PTGS's macro architecture.

batteries.

PR subsystem: rocket's electro-pneumatic equipment pressurization, activation of the pyrotechnic valves and unplugging of its umbilical cords.

CR subsystem: testing and preparation of the rocket's control sub-network, including the initiation of its Inertial Measurement Unit (IMU) and its on-board computer.

SC subsystem: security and safety testing of the rocket's equipment and the activation of its security valves during the automatic sequencing for launching.

EV subsystem: flight events sequence testing and the activation of the rocket's security mechanical devices (SMD) during the automatic sequencing for launching.

MN subsystem: visualization of the information collected during the rocket's testing, preparation and automatic sequencing for launching.

Figure 4.1 shows the macro architecture of TPGS, showing its main subsystems and its external interfaces.

4.2 User Interfaces

The functional requirements for the TPGS were established separately for each subsystem in a 250-page document. The rocket's preparation process for the flight involves thoroughly checking specific parts of rocket's electrical network. The operators of each subsystem illustrated in Figure 4.1 must follow a comprehensive preparation checklist that includes procedures and interactions with the rocket's hardware and software in real-time. The preparation and testing process is mainly based on the user interfaces inputs and outputs. As each TPGS's subsystem has a specific set of critical procedures to be accomplished, the user interfaces must provide an acceptable level of trustworthiness to deal with this situation.

Each user interface is composed of a number of panels. Some panels present messages to the operator about the state of the system/process, others allow him to interfere in the system/process, for instance running tests. In some cases, graphical representations (i.e. synoptics) are used, while in other case the panels consist mostly of buttons and textual fields. Due to confidentiality constraints, we are not able to describe here the user interfaces in any detail. In any case, it is relevant to note that each panel can contain several tens of user interface components, and that they are interconnected so that events in one panel will affect others. This will become clearer in the description of the models presented in the following chapters.

4.3 Selecting components for analysis

Considering the TPGS' user interfaces complexity, we decided to select two of the subsystems to apply our approach, the EV subsystem and the CR subsystem. By modeling them, we intended to address two issues:

- determine whether the expressiveness of the language was enough to model the type of user interfaces used in the aerospace field (and study the best approach to model them);
- determine the viability of analysing systems as complex as these (and study the level of detail that could be achieved while maintaining the analysis feasible).

The EV Subsystem is responsible for the testing and preparation of one of the rocket's electrical sub-networks, where the flight events and the rocket's security mechanical devices (SMD) are activated. During the rocket's flight, this sub-network is responsible for the sequential activation and control of the pyrotechnic systems in real-time according to a pre-programmed flight strategy.

The CR subsystem is responsible for testing, simulating, and analysing the automatic launch sequence, and monitoring a number of devices in the rocket after the launch.

4.4 Modelling Strategies

Considering the size of the subsystems being modelled, two main modelling strategies were considered. The first makes use of the Object-oriented features of the MAL Interactor language. Each model becomes a tree-like composition of Interactors (organised through aggregation relations), where each Interactor represents an entity in the interface (e.g. a panel or a control representing the state of a variable in one of the panels). The second strategy uses only one Interactor and models all the interface in that single Interactor. Visually we can see this strategy as a "flat" hierarchy.

It is envisaged that the first strategy will be better in terms of the organisation, maintainability and understandability of the model. One potential problem with this first approach relates to expressing communication between the different Interactors. Communication is achieved by explicit synchronisation on actions or variables' values between Interactors. Because of the tree-like compositional nature of the language, this can only be done top down in the composition tree. For example, in Figure 4.2 Interactor *main* knows Interactors *TMT* and *ALAR*, but these have no knowledge of each other or of Interactor *main*. Hence, communication between any two Interactors must be expressed in the Interactor at the root of the subtree they belong to (in the example, *main*).

One advantage of the first approach is the "reusing" of code. As the modelled system has several variables that record telemetry values, we can simply write a Interactor that captures the behaviour of a variable. Then in the screen that has these variables, we can instantiate as many Interactors as we want to represent the variables. The only code we need to add, besides the aggregation, is the

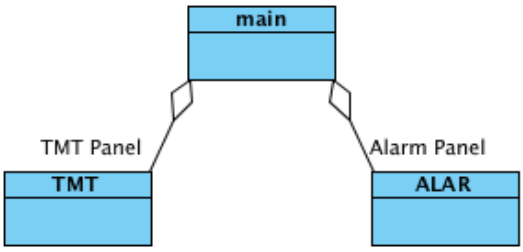


Figure 4.2: An hierarchy of interactors.

synchronization properties.

Besides the strategies to structure the model, strategies for specifying dependencies between variables must also be considered. In many cases the value of a variable will depend on the values of other variables in the model. One possibility is to express these dependencies in the definition of actions. Actions will affect these dependent variables differently according to specific conditions expressed as guards in the modal axioms. This will result in more complex modal axioms, and the guards must be repeated for each axioms.

When a variable's values depends exclusively on other variables (and not on specific actions) an alternative is to use invariants. By their nature, invariants can be useful for more than one action. They are a global property that we assure that is always true.

In Chapter 7 we will further explore the invariants *vs.* guards approach to expressing dependencies between variables, in particular with respect to verification performance.

4.5 Conclusion

This chapter has briefly described the system under analysis, the selected subsystems, and the modelling strategies that were considered for the analysis. The next chapters will present the modelling of the selected subsystems.

Chapter 5

EV Subsystem

EV subsystem is the Flight Events Sequence Network. It is a subsystem of TPGS, responsible to test and prepare one of the rocket's electrical sub-network. In the next section the modelling strategy adopted is explained. In the following sections the modelling of EV subsystem is presented.

5.1 Structure of the model

Figure 5.1 shows the architecture of the model. A *main* Interactor coordinates a number of other Interactors, each corresponding to a different panel in the user interface. In this case four panels are being considered: SIN, a Synoptic panel, shows the EV's electrical network; TMT, the telemetry panel, show information obtained from the telemetry system; ALAR, the alarms panel, shows messages regarding values that are in the alarm state; ALER, the alerts panel, show messages regarding values that are in the alert state. Each of these panels aggregates variables representing the information they display.

It must be noted that panels *Alarm* and *Alert* are special cases. They display the variables which are in an alarm or alert condition in the other panels. Hence, the values of their attributes must be coordinated with the attributes in the other interactors.

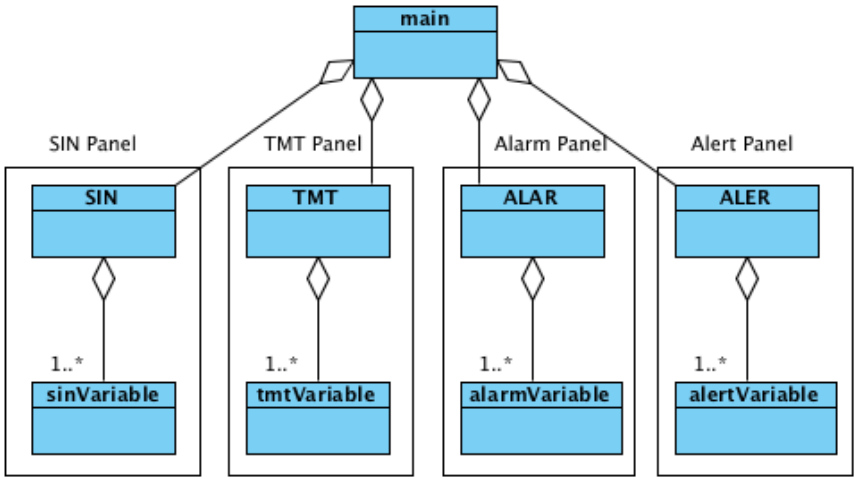


Figure 5.1: The models's macro architecture.

5.2 Navigation between screens

First, we started by modelling the navigation between the screens (one for each panel) of the system. That behaviour was represented in the *main* Interactor with the variable *display* with type *Screens* which is a set of the possible screens.

```
types
 Screens = {Principal, Sinotico, MonitTel, aler, alar, ...}
interactor main
 attributes
  [vis] display: Screens
 actions
  sin
  tmt
  alarmes
  ...
```

There is an initial panel for the operator to login to the system. Only after the operator have successfully logged in the main panel appears. To change between screens the user interface provides a set of buttons. Actions representing each of the buttons hence, the selection of each screen were defined. The actions are all defined in the same way: when an action is triggered, for example *sin*, the value

of the *display* changes to represent the appropriate screen (in the case bellow, the *Sinotico* screen):

[sin] display'=Sinotico

As some navigation buttons do not always appear in the screen, some restrictions have to be made in order to represent this behaviour. The action *sin* for example, is not available in the access screen, when the system is blocked, and in the exit screen. To represent that, we need to add a permission statement, saying when the action *sin* can be triggered.

per(sin) -> !(display in {acessoSis,acessoSev,Bloqueio,Escolha,Final})

Other actions are the opposite. They are only available in the access screens. The difference is only in the permission statement.

There is a particular action (*executar*) that appears in three of the screens only. The difference to the other actions is that it behaves differently according to the current screen. The permission statement is similar, but the definition of the action(s) needs to be different. When the screen is in *Escolha* the next screen after the action is *acessoSev*, when the screen is either *acessoSev* or *Bloqueio* then the next screen in *Principal*. This behaviour is captured by the following axioms:

per(executar) -> (display=Escolha | display=acessoSev | display=Bloqueio)
display=Escolha -> [executar] display'=acessoSev
(display=acessoSev | display=Bloqueio) -> [executar] display'=Principal

As we see above the permission statement is quite similar to the previous one. For the action, we wrote two modal axioms. The right side of the implications in the modal axioms (e.g. *[executar] display'=acessoSev*) is similar to the other actions, the left side (e.g. *display=Escolha*) is the guard that defines the conditions under which the action is defined (by that particular axiom).

To finalise the modelling of the navigation between screens we only need to define which is the first screen of the system. This is done by adding the initialization axiom bellow.

[] display=acessoSis

5.3 Modelling the variables and the screens

Modelling navigation between screens was relatively straightforward the next step was to model the screens. Since each of the screens presents information about a specific aspect of the system (particular cases are the alerts and warning screens that show lists of variables in alert/alarm), the first step was to model, in a general way, the behaviour of a single variable. Then, modelling a screen was the aggregation of a relevant number of variables, with any additional actions and control logic.

5.3.1 Variables

A variable has a value which can change over time, a colour, and an associated characteristic (fixed or blinking). The value of the variable changes according to some underlying system characteristic (e.g. temperature of some specific device).

The *TMT* screen has the more complex variables featuring characteristics such working limits, description, and measurement units. In fact, the variables in the other screens can be derived from *TMT* screen ones as they only have some of their attributes. To capture all that behaviour we designed the *tmtVariable* Interactor, which represents a variable of the *TMT* screen.

```
types
 Int = 0..5
 Unit = {YES,NO,VOLT}
 Colour = {yellow, red, green}
 Characteristic = {Fixed, Blink}
interactor tmtVariable
 attributes
  supAlertLim: Int
  infAlertLim: Int
  supAlarmLim: Int
  infAlarmLim: Int
  value: Int
  colour: Colour
  unit: Unit
```

```
 critical: boolean
 characteristic: Characteristic
actions
 setValue(Int)
```

The Int (integer) type is defined as the values from *0* to *5*. Initially was defined from *0* to *10*. For our study what is relevant is that we can set a certain integer value for the different states (alert, alarm), so the smaller range is enough. The types *Unit*, *Colour* and *Characteristic* are enumerations with the possible values for the corresponding attributes (*unit*, *colour* and *characteristic*).

As described in the user manual, there are four attributes pertaining to the working limits: *supAlertLim* (superior alert limit), *infAlertLim* (inferior alert limit), *supAlarmLim* (superior alarm limit) and *infAlarmLim* (inferior alarm limit). Then we have the attribute *value* which represents the value displayed by the variable at each moment, *colour* which is the colour being used to display the variable, *unit* represents the unit of the value, *critical* which states if the variable is critical, and the blinking characteristic defined by the *characteristic* attribute.

Regarding actions, the action *setValue* updates the value of the variable. This is the only action needed in this interactor. A first attempt to define it kept all attributes, other than the value, unchanged, as shown the above expression.

```
[setValue(_v)] value' = _v &
     keep(supAlertLim, infAlertLim, supAlarmLim,
          infAlarmLim, unit, colour, characteristic)
```

Although the EV allows to set the working limits, we did not model that feature for performance reasons in the verification. In Section 7 we explore those issues further.

The definition above, however, is too simplistic. Changes to the variable's value can cause alarms or alerts, which in turn can be acknowledged, or not, by the operator. A variable has the colour red and blinks, when its value is in alarm (i.e. the value falls inside the alarm limits), has not been acknowledged by the operator, and is critical. Naturally, since this condition must always hold, it can be said that it is an invariant of the system's behaviour. However, we did not use invariants for this condition, as will be explained in the next section. Instead, we used guards

on the modal axioms. Hence, for the condition just described the axiom above becomes:

```
[setValue(_v)]
  ((_v < infAlarmLim) | (_v > supAlarmLim) & critical)
   -> value' = _v & colour' = red & error' = Lim & alarmState' = AlaNRec
    & characteristic' = Blink & keep(supAlertLim,infAlertLim,supAlarmLim,
                                infAlarmLim,unit,critical,alertState)
```

The difference to the previous definition is that now we have specified changes to *colour* and *characteristic*, and we have introduced new variables *error*, *alarmState* and *alertState*. The guard states under which conditions this particular axiom applies. When the guard is true, a new error is triggered, the colour and the characteristic become red and blink, respectively, and the alarm state is set to non-acknowledged (*AlaNRec*). Besides this specific axioms, there are four others, defined under the same principle. Together, the axioms cover all the conditions over the working limits and critical condition of the variable expressed in the users' manual.

The definition of the new variables mentioned above are added to the *tmtVariable* Interactor as:

```
types
 ...
 Error = {Lim,nil}
 State = {AleRec,AleNRec,AlaRec,AlaNRec,Good}
interactor tmtVariable
 attributes
  ...
  error: Error
  alarmState: State
  alertState: State
```

The variable *error* records what kind of error has been detected, and the variables *alarmState* and *alertState* the state of the alarms and alerts, respectively. Because the type *State* has all the possible states for alarms and alerts, we must constrain the possible value of the alarm and alert state variables, with invariants as seen below.

(alarmState!=AleNRec) & (alarmState!=AleRec)
(alertState!=AlaNRec) & (alertState!=AlaRec)

5.3.2 Screens

A model of the *TMT* screen can now be created by aggregating instances of the *tmtVariable* interactor. The *TMT* screen has a table with more than 30 variables. Its function is to allow the monitoring of the variables, and as such, there are no actions for the user to perform in the screen itself. However, similarly to variables, the button providing access to this screen (indeed all the buttons providing access to the different screens) has a colour and a blinking characteristic. Two attributes to represent them are added to the model, resulting in the following Interactor:

```
types
 Id = {BD1A}
 ...
interactor TMT
 aggregates
  tmtVariable via BD1_A
 attributes
  characteristic: Characteristic
  colour: Colour
 actions
  setValue2(Id,Int)
```

In this example only one variable (*BD1_A*) was aggregated to the Interactor.

Note that while the screen does not have any action for the user to use, the Interactor defines *setValue2* action. This is an internal action used to set the values of the aggregated variables. It was defined to support the coordination invariants in the *main* Interactor. Furthermore, the *Id* type was defined as the enumeration of all button names (in this simple illustrative case only one). This enables the definition of *setValue2* to be generic. By being parameterized on the name of the button, the action can be defined to set the value of any button.

The working principle of the colour and characteristic attributes for the screens' access buttons is similar to that of the variables. The button is red and blinks when

at least one critical variable of the screen is in non-acknowledged alarm. As with the variables, the buttons have four more conditions related to the alarm and alert states of the variables contained in the screen. The idea is that the most serious condition (alert or alarm) prevails.

In this case of a single variable we can write the following invariants which directly express each of the rules in the manual:

```
BD1_A.colour = red -> (colour = red & characteristic = BD1_A.characteristic)
(BD1_A.colour = yellow & BD1_A.characteristic = Blink) ->
      (colour = yellow & characteristic = Blink)
(BD1_A.colour = yellow & BD1_A.characteristic = Fixed) ->
      (colour = yellow & characteristic = Fixed)
(BD1_A.colour = green) -> (colour = green & characteristic = Fixed)
```

For a single variable, we are able to simply say that the colour of the button is the same as the variable. The same applied to the characteristic. With more than one variables the invariant becomes larger and more complex, but the principle is the same. Note that while we could simplify the axioms, leaving them thus makes for a clearer connection to the manual.

Regarding the *setValue2* action, we need to synchronize it with the *BD1_A.setValue* action. That is, if the variables changes value so should the access button. A first, tentative, definition is simple:

```
[setValue2(BD1A,_v)] BD1_A.value' = _v
```

With this definition we are actually stating that when the *set* in the screen is triggered, the *set* in the variable is triggered too (because the value of the variable is being updated). In fact, the system does not work in this way, the system sets the variable and the screen is updated accordingly. We can make this clear by using invariants and permission axioms to express, and force, that when a variable sets his value the screen performs the same action, as shown in the following expression:

```
effect(BD1_A.setValue(_v)) -> effect(setValue2(BD1A,_v))
per(setValue2(BD1A,_v)) -> effect(BD1_A.setValue(_v))
```

The first implication states that when the *BD1_A* variable sets its value a set for that variable happens in the screen too (i.e., *setValue2* happens when the set of the variable happens). The second implication (a permission axiom) states that *setValue2* for *BD1_A* is only permitted when *BD1_A* is set (i.e., *setValue2* cannot happen unless the set of the variable happens).

5.3.3 Alarms and Alerts panels

Aggregating the TMT Interactor to the *main* Interactor, we now have the navigation and one screen working. The acknowledgement of alarms and alerts is done in the alerts and alarms screens. Hence, we need to add those screens to the model.

The alarms and alerts screens have a list of alarms and errors, respectively, that can be acknowledged by the operator. The same approach used for the *TMT* screen was followed: defining alerts or alarms variables, and aggregating them to create the screens' models.

The simpler variable is the alert variable:

```
interactor alertVariable
 attributes
  state: State
  error: Error
 actions
  recAlert
  setState(State)
```

The alert variable has the attribute *error* that records the error, and the attribute *state* that records the state of the alert (acknowledged or not). The actions are *recAlert*, to acknowledge the alert, and *setState*, which is used when a new value is updated in the *TMT* screen. Because the type *State* has the states for alarms and alerts we need to add the following invariant:

```
(state != AlaNRec & state != AlaRec)
```

This invariant states that the state of a alert variable can never be in alarm, be it acknowledged or non-acknowledged.

Considering that, as described in the user manual, the only error in the variables is the out of limits condition, we can write the following invariants:

```
(state = AleRec | state = AleNRec) -> error = Lim
(state = Good) -> error = nil
```

These invariants state that when the variable is in either type of alert, the error is due to an out of limits condition. When a variable is not in alert the error is *nil*.

The definition of the action *recAlert*, is straightforward:

```
[recAlert] state' = AleRec & keep(error)
```

The action *recAlert* changes the state to an acknowledged alert, and keeps the error (because the value remains in alert). A constraint is added to the action to avoid recognizing an alert that is already recognized:

```
per(recAlert) -> state = AleNRec
```

This constraint means that the action can only happen when the variable is in unacknowledged alert.

The definition of *setState* is simple too:

```
[setState(_e)] state' = _e
```

The variable *error* is not kept because a error can be triggered. The error's invariant above maintain the variable correctly set.

As with the *TMT* variables, we created an Interactor to aggregate the alert variables.

```
interactor ALER
 aggregates
  alertVariable via BD1_A
 attributes
  characteristic: Characteristic
  colour: Colour
 actions
  recAlert(Id)
  setState(Id,State)
```

This Interactor represents the alert screen and has colour and characteristic (variables *colour* and *characteristic*, respectively) in its access button. The invariants that state the behaviour of the button are the following:

BD1_A.state = Good -> colour = green & characteristic = Fixed
BD1_A.state = AleRec -> (colour = yellow & characteristic = Fixed)
BD1_A.state = AleNRec -> (colour = yellow & characteristic = Blink)

When there is no alert in the screen, the access button becomes green and does not blink. When there is at least one acknowledge alert the button becomes yellow and does not blink. Yellow and blinking happens when there is at least one unacknowledged alert. The invariants are only defined for a variable, the first and last implication are easily extended to consider more variables. The middle implication is not so easily extended because of the precedence of alerts. The unacknowledged alert has precedence over the acknowledged alert i.e., if there is an acknowledged alert and an unacknowledged alert, the button becomes yellow and blinks.

The actions' definitions are similar to those of the *TMT* screen Interactor.

effect(recAlert(BD1A)) -> effect(BD1_A.recAlert)
per(recAlert(BD1A)) -> effect(BD1_A.recAlert)
[recAlert(BD1A)] BD1_A.state' = AleRec & characteristic' = Fixed & keep(colour)
effect(BD1_A.setState(_e)) -> effect(setState(BD1A,_e))
[setState(BD1A,_e)] BD1_A.state' = _e

Note that action *setState*, as *setValue2* in the *TMT* screen, do not keep the variables *colour* and *characteristic*. The invariant above guarantees that the values of these variables will be updated as needed.

The alarm screen is similar to this alert screen. The only difference is that instead of yellow, the possible color of the access button is red.

5.3.4 Coordination between the screens

Aggregating the *ALER* and *ALAR* Interactors to the *main* Interactor, we now have three screens. Previously, in the *TMT* Interactor, we had not added an action to acknowledge alarms and alerts. Because the variables in the alert and alarm

screens are those in the *TMT* screen, we have to coordinate their values. First, we added the actions *recAlar* and *recAler* to the *tmtVariable* Interactor.

```
[recAlar(_e)] alarmState' = _e & keep(colour,value,supAlertLim,infAlertLim,
 supAlarmLim,infAlarmLim,unit,critical,error,alertState)
[recAler(_e)] alertState' = _e & keep(colour,value,supAlertLim,infAlertLim,
 supAlarmLim,infAlarmLim,unit,critical,error,alarmState)
```

We also added the corresponding actions in the *TMT* Interactor.

```
effect(BD1_A.recAlar(_e)) -> effect(recAlar(BD1A,_e))
per(recAlar(BD1A,_e)) -> effect(BD1_A.recAlar(_e))
[recAlar(BD1A,_e)] BD1_A.alarmState' = _e

effect(BD1_A.recAler(_e)) -> effect(recAler(BD1A,_e))
per(recAler(BD1A,_e)) -> effect(BD1_A.recAler(_e))
[recAler(BD1A,_e)] BD1_A.alertState' = _e
```

In order to coordinate the variables in the *TMT* screen with those in the alerts and alarms screens, we had to add coordination invariants in the *main* Interactor. First, we defined when some action can take effect.

```
per(monitALAR.setState(_b,_e)) ->
 (effect(monitTMT.setValue2(_b,0))
 | effect(monitTMT.setValue2(_b,1))
 | effect(monitTMT.setValue2(_b,2))
 | effect(monitTMT.setValue2(_b,3))
 | effect(monitTMT.setValue2(_b,4))
 | effect(monitTMT.setValue2(_b,5)))
```

The action *setState*, in the alarms screen, can only take effect when the value of the respective variable is updated. For the alerts screen the same situation occurs with the acknowledgement actions.

```
per(monitTMT.recAlar(_v,_e)) -> effect(monitALAR.recAlarm(_v))
per(monitTMT.recAler(_v,_e)) -> effect(monitALER.recAlert(_v))
```

The acknowledgement is made only in the alarm and alert panels, it is then propagated to the other screens as shown in the previous expressions. The alarm and alert states of the variables need to be synchronized, as indicated in the following expressions:

monitTMT.BD1_A.alertState = monitALER.BD1_A.state
monitTMT.BD1_A.alarmState = monitALAR.BD1_A.state

The actions to acknowledge alarms and alerts are done in their respective screens, so we have to restrict when they can be made (when the relevant screen is being displayed).

per(monitALAR.recAlarm(_v)) -> display=alar
per(monitALER.recAlert(_v)) -> display=aler

The invariants above assert that, the acknowledgement action can be made only in their respective screen.

5.3.5 Adding the SIN screen

The model of a new screen, called *SIN*, was aggregated to the *main* Interactor. It is a screen that contains a large number of variables. Each variable has only the value, colour and characteristic. The concept is the same, an Interactor representing a variable, simpler than the others, and an Interactor representing the *SIN* screen with the variables.

```
interactor sinVariable
 attributes
  colour: Colour
  characteristic: Characteristic
  value: Int
 actions
  setValue(Int)
  recAlarAler
 axioms
  [setValue(_v)] value' = _v
  [recAlarAler] keep(value)
```

The *SIN* Interactor is similar to the others but, besides variables, the screen has safety relays. An indicator in each relay indicates whether it can be released or not. These relays were not modelled with dedicated Interactors, as the variables were. Because they can be in one of two states only, and do not have any associated colour or characteristic, they were modelled as booleans directly in the *SIN* Interactor. Additionally, the SIN screen can be set to automatic mode.

```
interactor SIN
 aggregates
  sinVariable via BD1_A
 attributes
  RS_A_4C: boolean
  automatic: boolean
  colour: Colour
  characteristic: Characteristic
  lib_RS_A_4C: boolean
 actions
  setValue2(Id,Int)
  setRS_A_4C(boolean)
  setAutomatic
  setManual
  recAlarAler(Id)
  libReles
  inibReles
```

The actions *setValue2* and *recAlarAler* help simplify the synchronization invariants. The *setRS_A_4C* action changes the safety relay's state. The actions *setAutomatic* and *setManual* set and unset the automatic mode. The *libReles* and *inibReles* actions activate and deactivate the release indication of the safety relays. We can note bellow that the safety relay *RS_A_4C*, can only be set to true if the release sign (*lib_RS_A_4C*) is true.

```
per(setRS_A_4C(true)) -> lib_RS_A_4C
[setRS_A_4C(_b)] RS_A_4C' = _b & keep(automatic,lib_RS_A_4C)
```

The action definition is done with a typical model axiom, to arm or disarm the safety relay, but arming is only possible if the release sign is true, as stated in the

permission axiom in the first line.

To finish our modelling we need to aggregate the *SIN* Interactor to the *main* Interactor, and add the coordination invariants. The invariants are similar to those above.

5.4 Adding another variable

Until now only one variable was modelled in each of the screens. Now we will see the modelling effort needed to add a new variable. First we need to add the variables to each of the screens. In the Interactor *SIN* we need to aggregate a new *sinVariable* Interactor, and the respective coordination invariants and actions.

```
types
 Id = {BD1A,tensaoBD1A}
 ...
interactor SIN
 aggregates
  sinVariable via BD1_A
  sinVariable via tensaoBD1_A
  ...
 axioms
  ...
  effect(tensaoBD1_A.setValue(_v)) -> effect(setValue2(tensaoBD1A,_v))
  per(setValue2(tensaoBD1A,_v)) -> effect(tensaoBD1_A.setValue(_v))
  effect(tensaoBD1_A.recAlarAler) -> effect(recAlarAler(tensaoBD1A))
  per(recAlarAler(tensaoBD1A)) -> effect(tensaoBD1_A.recAlarAler)
  [recAlarAler(tensaoBD1A)] keep(RS_A_4C,automatic,lib_RS_A_4C)
  [setValue2(tensaoBD1A,_v)] tensaoBD1_A.value' = _v
      & keep(automatic,RS_A_4C,lib_RS_A_4C)
```

Recall that the actions *setValue2* and *recAlarAler* receive an *Id* as parameter. It is then necessary to add the new id to the type. The invariants and the actions are copies of the existing ones for the *BD1_A* variable. The part that grows more, as more variables are added, is the invariants that changes the colour of the screen access button.

```
((BD1_A.colour = red & BD1_A.characteristic = Blink) |
 (tensaoBD1_A.colour = red & tensaoBD1_A.characteristic = Blink))
 -> (colour = red & characteristic = Blink)
```

The button will be red and blinking if it exists at least one variable which is red and blinking.

```
(((BD1_A.colour = red & BD1_A.characteristic != Blink) &
  (tensaoBD1_A.colour != red | tensaoBD1_A.characteristic != Blink)) |
 ((tensaoBD1_A.colour = red & tensaoBD1_A.characteristic != Blink) &
  (BD1_A.colour != red | BD1_A.characteristic != Blink)))
 -> (colour = red & characteristic = Fixed)
```

The button will be red and fixed, if it exists at least one variable which is red and fixed and, and no variable is red and blinking.

```
((BD1_A.colour = yellow & BD1_A.characteristic = Blink &
  tensaoBD1_A.colour != red) |
 (tensaoBD1_A.colour = yellow & tensaoBD1_A.characteristic = Blink &
  BD1_A.colour != red))
 -> (colour = yellow & characteristic = Blink)
```

The button will be yellow and blinking, if exists at least one variable yellow and blinking and, no one red and blinking or fixed.

```
(((BD1_A.colour = yellow & BD1_A.characteristic != Blink) &
  (tensaoBD1_A.colour != red | (tensaoBD1_A.colour = yellow &
  tensaoBD1_A.characteristic != Blink))) |
 ((tensaoBD1_A.colour = yellow & tensaoBD1_A.characteristic != Blink) &
  (BD1_A.colour != red | (BD1_A.colour = yellow &
  BD1_A.characteristic != Blink))))
 -> (colour = yellow & characteristic = Fixed)
```

The button will be yellow and fixed, if exists at least one variable yellow and fixed and, no one red and blinking or fixed, and yellow and blinking.

```
(BD1_A.colour = green & tensaoBD1_A.colour = green)
 -> (colour = green & characteristic = Fixed)
```

The button will be green and fixed, if all the variables green.

The new variable *tensaoBD1_A* has a big impact in these invariants, in terms of the number of lines and model complexity. It is quite hard to not get lost in such a large invariant.

For the *TMT* screen the effort is the same.

```
interactor TMT
 aggregates
  tmtVariable via BD1_A
  tmtVariable via tensaoBD1_A
  ...
 axioms
  ...
  effect(tensaoBD1_A.setValue(_v)) -> effect(setValue2(tensaoBD1A,_v))
  per(setValue2(tensaoBD1A,_v)) -> effect(tensaoBD1_A.setValue(_v))
  [setValue2(tensaoBD1A,_v)] tensaoBD1_A.value' = _v
  effect(tensaoBD1_A.recAlar(_e)) -> effect(recAlar(tensaoBD1A,_e))
  per(recAlar(tensaoBD1A,_e)) -> effect(tensaoBD1_A.recAlar(_e))
  [recAlar(tensaoBD1A,_e)] tensaoBD1_A.alarmState' = _e
  effect(tensaoBD1_A.recAler(_e)) -> effect(recAler(tensaoBD1A,_e))
  per(recAler(tensaoBD1A,_e)) -> effect(tensaoBD1_A.recAler(_e))
  [recAler(tensaoBD1A,_e)] tensaoBD1_A.alertState' = _e
```

The invariants follow the same scheme of the previous screen.

The screens Alerts is somewhat similar, but a bit simpler.

```
interactor ALER
 aggregates
  alertVariable via BD1_A
  alertVariable via tensaoBD1_A
  ...
 axioms
  effect(recAlert(tensaoBD1A)) -> effect(tensaoBD1_A.recAlert)
  per(recAlert(tensaoBD1A)) -> effect(tensaoBD1_A.recAlert)
  [recAlert(tensaoBD1A)] tensaoBD1_A.state' = AleRec &
    characteristic' = Fixed & keep(colour)
```

```
[setState(tensaoBD1A,_e)] tensaoBD1_A.state' = _e
effect(tensaoBD1_A.setState(_e)) -> effect(setState(tensaoBD1A,_e))
```

As in the previous screens, the invariants and actions definitions for the *tensaoBD1_A* variable are copies of the existing ones for the *BD1_A* variable. The invariants that changes the colour of the screen button are simpler too. That is because in this screens, the button can only take two colours. *Green* and *yellow* in the case of the alerts screen, and *green* and *red* in the case of the alarms screen. For the alerts screen the axioms are:

```
(BD1_A.state = Good & tensaoBD1_A.state = Good)
 -> (colour = green & characteristic = Fixed)

((BD1_A.state = AleRec & tensaoBD1_A.state != AleNRec)|
 (BD1_A.state != AleNRec & tensaoBD1_A.state = AleRec))
 -> (colour = yellow & characteristic = Fixed)

(BD1_A.state = AleNRec | tensaoBD1_A.state = AleNRec)
 -> (colour = yellow & characteristic = Blink)
```

In this screen the effort to change the invariant to consider two variables is not so complex. In fact it is straightforward to do it. For the alarms screen the new MAL axioms are analogous.

Lastly, it is necessary to add the coordination invariants in the *main* Interactor. Most of these invariants are defined in such a way that there is no need to change anything. The ones that need to be added are the following:

```
monitTMT.tensaoBD1_A.value = monitSIN.tensaoBD1_A.value
monitTMT.tensaoBD1_A.colour = monitSIN.tensaoBD1_A.colour
monitTMT.tensaoBD1_A.characteristic = monitSIN.tensaoBD1_A.characteristic
monitALAR.tensaoBD1_A.state=AlaRec
 -> monitTMT.tensaoBD1_A.characteristic = Fixed
monitSIN.RS_A_4C
 -> monitTMT.tensaoBD1_A.alertState = monitALER.tensaoBD1_A.state
!monitSIN.RS_A_4C -> monitALER.tensaoBD1_A.state = Good
monitSIN.RS_A_4C
-> monitTMT.tensaoBD1_A.alarmState = monitALAR.tensaoBD1_A.state
```

!monitSIN.RS_A_4C -> monitALAR.tensaoBD1_A.state = Good

Most of these invariants are copies of the existing ones for the *BD1_A* variable. The ones who are different are the last six lines. The alarms and alerts of the new variable *tensaoBD1_A* only appear on the alarms and alerts screens, respectively, if the *RS_A_4C* relay is enabled.

In conclusion most of the axioms needed to add a new variable to the system are almost copies the existing axioms. The only aspect that becomes somehow complex, is the modelling of how the colour of the screen buttons changes, in screens *TMT* and *SIN*.

5.5 Variables description

The previous sections focused on describing the resulting models. In order to provide some insights on how the models were derived from the user manuals, an example of modelling the constraints related to the colour and characteristics of the variables is presented next.

In the manual we have the following statements:

> "*Blinking yellow: For a critical variable, when the current value of the variable is in non recognized alert (value within the range of alerts), there is no recognized alarm in the same variable and do not exists the previous criterion. If over the same variable a recognized critical alarm exists, then the fixed red prevail. For a non critical variable, when the current value of the variable is in non recognized alarm (value within the range of alarms).*"

These statements indicate when some variable becomes yellow and blinking. Firstly, and because the manual says that when "*a recognized critical alarm exists, fixed red prevail*", we need to split the formalization into two parts. One for when a recognized critical alarm does not exist, and the variable becomes blinking yellow. Another for when a recognized critical alarm does exist, in which case the color and blinking state of the variable should be kept as they are.

Using the attributes of the Interactor *tmtVariable* we can translate the statements above one by one to MAL. The statement "*For a critical variable, when the current value of the variable is in non recognized alert (value within the range of alerts), there*

is no recognized alarm in the same variable and do not exists the previous criterion." is translated to:

```
critical &
((_v >= infAlarmLim & _v < infAlertLim) |
(_v <= supAlarmLim & _v > supAlertLim)) &
(alarmState != AlaNRec & alarmState != AlaRec)
```

The part of the statement "*do not exists the previous criterion*" means being in unrecognized alarm, which is defined in the second and third line of the above *MAL* specification. The variable *_v* is the current value.

The second part of the statement is "*For a non critical variable, when the current value of the variable is in non recognized alarm (value within the range of alarms).*". This is translated to:

```
((_v < infAlarmLim) | (_v > supAlarmLim)) & !critical)
```

Rearranging the *MAL* specification produced and incorporating to the action *setValue(_v)* the specification becomes:

```
[setValue(_v)]
  (((((_v >= infAlarmLim & _v < infAlertLim) |
  (_v <= supAlarmLim & _v > supAlertLim)) & critical)
  | (((_v < infAlarmLim) | (_v > supAlarmLim)) & !critical))
  & (alarmState != AlaNRec & alarmState != AlaRec))
  -> value' = _v & colour' = yellow & error' = Lim & alertState' = AleNRec
  & characteristic' = Blink & keep(supAlertLim,infAlertLim,supAlarmLim,
                        infAlarmLim,unity,critical,alarmState)
```

The statement "*If over the same variable exists a recognized critical alarm, then the fixed red prevail.*" produces a specification similar to the previous. The only difference is in *(alarmState != AlaNRec & alarmState != AlaRec)* which becomes *(alarmState = AlaNRec | alarmState = AlaRec)*. Obviously *colour* and *characteristic* are not changed, only the *alertState* is updated to *AleNRec*.

5.6 Conclusion

The most interesting aspects of the EV subsystem, such as the alarms and alerts, was modelled. During the modelling some simplifications were made, in

Interactor	# attributes	# actions	# lines
main	1	25	130
sinVariable	3	2	11
SIN	5	7	51
tmtVariable	12	3	42
TMT	2	3	45
alertVariable	2	2	14
ALER	2	2	27
alarmVariable	3	2	15
ALAR	2	2	27
Total	**32**	**48**	**362**

Table 5.1: EV model figures.

order to reduce the verification time. One of them was the range of the *Int* type. Initially the type was defined from 0 to 10 but, to be able to set alarm and alert limits, from 0 to 5 was sufficient. The language was expressive enough to model the EV subsystem. In some cases the language could provide some "syntactic sugar" to reduce model size and to be easier to maintain.

Adding new variables proved to be not so difficult, with the exception of the invariants that changes the colour of the screen buttons. Those invariants escalated quickly in size, and because of that they become quite difficult to maintain or change.

Table 5.1 compiles some figures (number of attributes, actions and lines) about the model, considering the version with two variables. The Interactors *main* and *tmtVariables* are the bigger ones. *main* because of the navigation and constraints to synchronize all the Interactors, and *tmtVariables* because of the values of variables and control of the alarms and alerts triggered. The Interactor with more actions is *main* with 25, because of all the buttons to change between screens. The Interactor with more attributes is *tmtVariable* with 12, because of the number of characteristics that its variables have.

Chapter 6

CR Subsystem

CR subsystem is the Electric Control Network. It is a subsystem of TPGS responsible for the testing, simulation and analysis of the automatic launch sequence, and of a number of devices in the rocket, before its launch. In the following sections the modelling of CR subsystem is presented.

6.1 ‘‘Business layer’’

For modelling this subsystem we had access to a state machine that specifies when some action can take effect, or not. We could notice that the state machine works as the “business layer” of the model. Modelling in MAL being similar to defining a state machine, its translation to MAL is somehow direct. The state machine has nine states. A variable (*state*) is used to represent the current state. Each state has one or more labelled transition (actions) to other states or to itself. The actions in the model are the transition labels of the state machine. Some transitions have conditions that need to be met in order to be fired. The boolean variables needed to manage these conditions were identified and represented as attributes in the Interactor.

```
types
 States = {operationMode,microDisable,microEnable,testRec,simul,
 inicSAREC,automaticSeqRec,secureStateRec,inconsistentStateRec}
interactor main
 attributes
```

state : States
automaticMode : boolean
tstRecPer : boolean
simulPer : boolean
prepPer : boolean
saPer : boolean
saReady : boolean
actions
automatic
manual
criticFailure
inicSimul
endSimul
inicTestRec
endTestRec
recCondSatisfied
enableSa
disableSa
nowSa
restartOp
endSa
prepSaFailure
enableSo

We have represented the state machine in Interactor *main* in order to reduce the synchronization effort. As said above, the actual state is represented by the attribute *state*, of type *States*, which is an enumeration with the nine possible states. The rest of the attributes captured the state of some conditions:

- *automaticMode* - captures the mode of the system (*true* if automatic, *false* if manual);
- *tstRecPer* - captures if the tests of CR are enabled or not;
- *simulPer* - captures if the simulations of CR are enabled or not;
- *prepPer* - captures if the CR preparation is enabled or not;

- *saPer* - captures if the automatic sequence is enabled or not;
- *saReady* - captures if the TPGS is ready for the automatic sequence or not.

All this variables are typed as *boolean*.

In the actions defined below the attributes are the transitions of the state machine. Each transition (action) can only happen in a particular state or set of states, for example the action *criticFailure* can only happen in the states *microEnable*, *inicSAREC* and *automaticSeqRec*. Such condition is expressed as follows:

```
per(criticFailure) -> state in {microEnable,inicSAREC,automaticSeqRec}
```

This expression means that the action *criticFailure* is permitted only if *state* (i.e. current state) is equal to *microEnable*, *inicSAREC* or *automaticSeqRec*. For the other actions the process is similar.

We have defined the states from where the transitions "leave", now we need to define to what state they "go", and what they "change". Most of the transitions (actions) have different behaviour according to the actual state. The way to represent that is using guards. Consider, for example, the *criticalFailure* transition (action):

```
state=microEnable ->
 [criticFailure] tstRecPer'=true & simulPer'=true & saPer'=false
 & keep(automaticMode,state,prepPer,saReady)
state=inicSAREC ->
 [criticFailure] state'=microEnable & saPer'=false & saReady'=false
 & simulPer'=true & tstRecPer'=true & keep(automaticMode,prepPer)
state=automaticSeqRec ->
 [criticFailure] state'=secureStateRec & saPer'=false & prepPer'=false
 & keep(automaticMode,tstRecPer,simulPer,saReady)
```

Above we have three expressions, one for each state (*microEnable*, *inicSAREC* and *automaticSeqRec*, respectively). The first expression states that when the current state is *microEnable*, and the transition (action) *criticFailure* happens, the tests and simulations of CR are enabled, the automatic sequence is disabled, and the state and the remaining variables do not change. Visually the state machine evolves to the same state.

The second expression asserts that when the current state is *inicSAREC*, and the transition (action) *criticFailure* happens, the tests and simulations of CR are enabled, the automatic sequence is disabled, the TPGS is not ready for the automatic sequence, the state machine evolves to the state *microEnable* and the remaining variables do not change.

The third expression states that when the current state is *automaticSeqRec*, and the transition (action) *criticFailure* happens, the automatic sequence is disabled, the CR preparation is disabled, the state machine evolves to the state *secureStateRec* and the remaining variables do not change.

The remaining transitions (actions) were modelled following the same principle. To finish the "business layer" we defined the initial state and the values of the attributes:

```
[] state=microEnable & automaticMode = false & tstRecPer = false
 & simulPer = false & prepPer = false & saPer = false & saReady = false
```

The initial state of the state machine is *microEnable*, the CR is in manual mode and all the attributes are disabled. With this expression we have the "business layer" modelled.

6.2 Navigation between screens

Because the CR and the EV are subsystems of the TPGS, the navigation is uniform in all subsystems. The modelling of the navigation between screens was similar to the one modelled in the EV subsystem. We added the attribute *display* to the *main* Interactor, typed as *Screens*, and the actions to navigate between the panels, as seen below:

```
types
 ...
 Screens = {Voo, TestSim, DiagEsq, MonitTel, MonitUmb, ...}
interactor main
 attributes
  ...
  [vis] display: Screens
 actions
```

```
...
voo
tst
esq
tmt
tmu
...
```

Some of the panels are the same as in the EV, like the *TMT*, *TMU*, *Alarms* and *Alerts* panels. The modelling of the actions to navigate between the screens was similar:

```
per(voo) -> !(display in {acessoSis,acessoRec,Bloqueio,Escolha,Final})
[voo] display'=Voo & keep(state,automaticMode,tstRecPer,simulPer,prepPer,
                          saPer,saReady)
```

The first line is a permission statement and the second line is the definition of the behaviour of action *voo*. Like in EV the actions of navigation only change the *display*. The other navigation action were modelled analogously.

6.3 Variables

As in EV, CR has a set of variables that are monitored. We can reuse the *tmtVariable* Interactor, because the behaviour of the variables in the CR subsystem is the same as the EV. Although we had modelled the variables in the EV model, we did not reuse the Interactor. For this model, only the state of the variables was relevant (i.e. if in alarm, alert or in normal state). The *tmtVariable* Interactor has that characteristic modelled, but it also has many other unnecessary features. We have modelled a simplified version to reduce verification time.

```
types
 VarState = {Good,Alarm,Alert}
interactor VOOVariable
 attributes
  state : VarState
 actions
```

```
 setState(VarState)
axioms
 [] state = Good

 per(setState(_s)) -> state != _s
 [setState(_e)] state' = _e
```

For the EV model a variable has only the attribute state that records if the variable is in alarm, alert or normal (*Good*) states. It has the action *setState(VarState)*, which is the equivalent to the *setValue* in the *tmtVariable* Interactor of the EV model. The variable "begins" in the state *Good*, as the first axiom asserts. The line above the action definition are the condition to not change the state of the variable to the same state. It is defined for all the variable's possible states (Good, Alarm and Alert) by the variable *_s*. The definition of the *setState* action is straightforward.

6.4 Voo Screen

In this screen the operator can perform actions of preparation for the launch, and monitor the variables and results of the automatic sequence, whether in testing or during actual launch setting. This screen has some variables, so we need to aggregate instances of the *VOOVariable* Interactor to the Interactor that models the screen (*Voo*).

```
types
 ...
 Id = {IRECP}
 VarState = {Good,Alarm,Alert}
 MSIState = {nil,INICIO,PRONTO}
 INFOState = {nil,INIC,COMPL}
interactor VOO
 aggregates
  VOOVariable via irecP
 attributes
  aquec : MSIState
```

```
 state : States
 automaticMode : boolean
 stateInfo: INFOState
actions
 stateSync(States)
 aquecMSI
 doneAquecMSI
 sendInfo
 doneSendInfo
 automaticSync(boolean)
```

Some attributes on Interactor *VOO* are copies of attributes in the main Interactor, due to the problem of an included Interactor not having access to attributes and actions of the including Interactor. The attributes *state* and *automaticMode* are examples of that, and the action *stateSync* and *automaticSync* are the respective actions to synchronize the values between the two Interactors. The definition of them is similar to the ones in the EV.

```
[stateSync(_s)] state' = _s & keep(aquec,stateInfo,automaticMode)
[automaticSync(_b)] automaticMode' = _b & keep(aquec,stateInfo,state)
```

They update the respective values and do not change all the others. Then in the root Interactor (*main*) is necessary to add the invariants to maintain the synchronization.

The attributes *aquec* and *stateInfo* are used to record the state of two actions that can be done in this screen, heating the MSI and sending the PASS archive data to SOAB-CDB, respectively. The first action (heating the MSI) is modelled by the actions *aquecMSI* and *doneAquecMSI*.

The semantics for the actions was derived from the user manual as already illustrated. The manual states that it is only possible to heat the MSI when there is no variable in alarm or alert and the system is in automatic mode. Additionally, if the MSI is already heating when the command to heat the MSI is issued, the heating stops.

```
per(aquecMSI) -> (irecP.state = Good
 & !(state in {operationMode,microDisable,inconsistentStateRec}))
```

aquec = nil -> [aquecMSI] aquec' = INICIO & keep(state,stateInfo,automaticMode)
aquec != nil -> [aquecMSI] aquec' = nil & keep(state,stateInfo,automaticMode)

The first statement asserts that is only permitted to perform the action if the variable is not in alarm or alert and the system is in the states where the automatic mode is not manual. The second line is the definition of the action *aquecMSI* when the "display" that show the state of heating the MSI (*aquec*) is empty. In that case the attribute *aquec* changes to *INICIO*, meaning that the heating began. The third line is the same action but when *aquec* is not empty, it stops the heating and clears the *aquec* attribute.

The other action was defined to model the end of heating the MSI:

per(doneAquecMSI) -> (irecP.state = Good & aquec = INICIO)
[doneAquecMSI] aquec' = PRONTO & keep(state,stateInfo,automaticMode)

The permission statement in this case is that the variables are still not in alarm or alert and the heating has already started (*aquec* = *INICIO*). The definition of the action is to change the *aquec* attribute to *PRONTO*, and keep all the other attributes. Note that this action in the system is not triggered by the user. It can be seen as an event.

The second action, sending the PASS archive data, is somewhat similar. It is modelled by the *sendInfo* action, and as in the heating of the MSI, an action that models the completion of sending the PASS archive is defined.

per(sendInfo) -> (stateInfo in {nil,COMPL}
& !(state in {operationMode,microDisable,inconsistentStateRec}))
[sendInfo] stateInfo' = INIC & keep(state,aquec,automaticMode)
per(doneSendInfo) -> stateInfo = INIC
[doneSendInfo] stateInfo' = COMPL & keep(state,aquec,automaticMode)

The first line asserts that the action *sendInfo* can only take effect if it is not already sending the archive, and the system is in automatic mode. The third line is a straightforward definition that changes the "display" of this action (*stateInfo* attribute) to *INIC*, stating that the sending began. The fourth and fifth lines are related to the completion of sending the archive. The fourth line states that the action *doneSendInfo* can only take effect if the sending has already started. The fifth line is the definition of the action, similar to the previous.

When these two actions are completed with success other action is trigged in the business layer (*recCondSatisfied*). In order to trigger the action *recCondSatisfied*, the following invariant was added in the *main* Interactor:

saPer = true -> (monitVOO.aquec = PRONTO & monitVOO.stateInfo = COMPL)

The action *recCondSatisfied* is triggered when the system is ready for the automatic sequence, and the attribute *saPer* states that. The invariant assures that the attribute *saPer* is *true*, only when the heating of the MSI and the sending the PASS archive are completed with success.

6.5 Automatic Sequence

The automatic sequence is monitored in this screen by six rectangles, that display information according to the evolution of the events of the automatic sequence. Hence, six attributes were added to the *VOO* Interactor in order to model the automatic sequence.

```
types
 ...
 Int = 0..11
 FGState = {nil,COMAND}
 ALGORITState = {nil,INIC}
 ...
attributes
 ...
 saState : Int
 prepMSI : MSIState
 habilFG : FGState
 inicBR3 : FGState
 com1PSR : FGState
 navon : FGState
 algo1E : ALGORITState
actions
 ...
 inicPrep
```

inicPrepMSI
habilFontGir
inicAlg
verFontGir
comPSR1
verPSR1
comMSIReady
verMSI
comInicNav
inicNavMSI

The automatic sequence was modelled as a ordered sequence of events (in this case actions). Beside the attributes to record the state of the six rectangles of the screen, the attribute *saState*, an integer used to define the sequence, was also added. *saState* starts as zero. The first event of the sequence was modelled as follow:

```
per(inicPrep) -> (aquec=PRONTO & stateInfo=COMPL & saState=0
            & state=automaticSeqRec)
[inicPrep] saState' = 1 & keep(aquec,state,stateInfo,automaticMode,
                  prepMSI,habilFG,inicBR3,com1PSR,navon,algo1E)
```

The automatic sequence can only start when the MSI is heated and the PASS archive is sent. The first line asserts that the *inicPrep* action can only take effect if the MSI is heated, the PASS archive is sent, the state of the system (*state*) is in the automatic sequence and the attribute of the automatic sequence order is equal to zero. The action simply increments the attribute *saState*, so that the model might be able to accept the next event in the automatic sequence. Notice that there are now more attributes. The new attributes were added to the *keep* operation of the previously defined actions of this Interactor, so that they would not randomly change their value.

The next event is beginning the preparation of the MSI (*inicPrepMSI*). For this event the permission statement is simpler, it is only necessary to check that the automatic sequence order variable is equal to one. That statement is in the first line below.

```
per(inicPrepMSI) -> (saState = 1)
```

```
[inicPrepMSI] prepMSI' = INICIO & saState' = 2 & keep(aquec,state,
   stateInfo,automaticMode,habilFG,inicBR3,com1PSR,navon,algo1E)
```

As in the previous action the attribute *saState* is incremented, besides that the "rectangle" *prepMSI* changes to *INICIO*, stating that the preparation of the MSI has began.

The remaining events were modelled in an analogous manner.

Because the *VOOVariable* was added to this Interactor there is the need to define the action that sets the state of the aggregated variable.

```
effect(irecP.setState(_e)) -> effect(setState(IRECP,_e))
per(setState(IRECP,_e)) -> effect(irecP.setState(_e))
[setState(IRECP,_e)] irecP.state = _e & keep(aquec,state,...)
```

This definition is similar to ones defined in the EV model. The first two lines defines when this action happens and how it synchronizes. The last line is the definition of the action, which updates the state of the variable. Note that it is necessary to add the action *setState(Id,VarState)* to the list of actions in this Interactor.

To finalize, we aggregated this Interactor to the *main* and added the invariants *state = monitVOO.state* and *automaticMode = monitVOO.automaticMode* to synchronize the values of *state* and *automaticMode*. We now have the CR subsystem with the Voo screen.

6.6 Conclusion

The CR subsystem was modelled differently when compared to the EV. For this model we had access to a document that had a state machine, explaining when some action had permission to be happen, and how the subsystem evolved. Other difference is the abstraction made in the variables. Because several aspects concerning variables had been tested in the previous model, this model was reduced to the essential, the state of the variables (alarms and alerts), and the actual value of the variables was not considered. How to add new variables to the model was not discussed, but it is analogous to the EV model. In fact it is simpler because, due two the simplifications, the complex invariants that escalated quickly in terms of complexity are not needed in this case.

Interactor	# attributes	# actions	# lines
main	8	41	193
VOOVariable	1	1	11
VOO	11	18	92
Total	**20**	**60**	**296**

Table 6.1: CR model figures.

Table 6.1 compiles some figures (number of attributes, actions and lines) about the model, considering a single variable. The Interactores *main* and *VOO* are the bigger ones. The *main* because of the navigation and the "business layer" that controls when some actions can happen, and the *VOO* because of the action of the screen and the automatic sequence. The Interactor with more actions is *main* with 41, because of all the buttons to change between screens, and the "business layer" that has several actions too. The Interactor with more attributes is *VOO* with 11, because of the variables to control the automatic sequence and the actions of that screen.

Chapter 7

Analysis

In this Chapter the analysis performed on the models is described. Some properties that were tested and their results are presented, as well as verification times in the machines used. An analysis is also made of the IVY Workbench and some improvements to the tool is suggested.

7.1 Model Analysis

During modelling, we checked some properties in order to verify if the model was being properly designed, as described by the user manual, and more than that, to verify whether the system had errors or problems. As soon as we modelled the *TMT* panel with one variable, we started testing whether the model behaved as intended. To that end, we defined some CTL properties related to navigation. One example of this is the property:

!EF(display=Final)

This property states that there is no situation (or systems state) where the variable *display* is equal to *Final* — i.e., the system does not reach the exit panel. The result was a counterexample (see figure 7.1) illustrating the reaching of the exit panel, which is what we expected. The trace, in this case presented using the tabular representation, shows the sequence of actions [*enter*, *executar* (execute), *executar*, *fim* (end)] and the changes they produce in the system state.

We have tested properties related to the actions too. For example, verifying if a variable can be in alert. In order to get counterexamples we wrote the inverse

	1	2	3	4	5
main.action	bloq	enter	executar	executar	fim
display	acessoSis	Escolha	acessoSev	Principal	Final
sinalSonoro	FALSE	FALSE	FALSE	FALSE	FALSE
sinalSonoroActivo	FALSE	FALSE	FALSE	FALSE	FALSE

Figure 7.1: Trace of the counterexample (main Interactor only).

property i.e., for all states where the variable *BD1_A* is in normal state (its colour is green), there is no next state where the variable is in alarm (colour red).

AG(monitTMT.BD1_A.colour = green -> !EX (monitTMT.BD1_A.colour = red))

As expected, the property was model checked as false, and a counterexample was generated.

After analysing the counterexample, illustrated in figure 7.2 (in this case using a state machine-like representation), we thought it was spurious. It highlighted a situation where the variable colour was red (rightmost column, $colour = red$), but under an acknowledged alert condition ($state = AleRec$). According to our understanding of what had been modelled, such situation should not be possible. Another strange aspect of the counterexample was that the limit variables were all being changed to zero (when they should always keep their values).

After analysing the conditions in the operation manual we found that it did not define what happens to a non critical alert, which was the case in the counterexample. Because of that, the behaviour of the system under that particular condition was not being defined in the model. Consequently the model checker was considering any possible behaviour as legal, which meant that variables where in practice being changed randomly. With this property, we found a gap in the definition of the alerts and alarms in the operation manual of the system. After discussing this with an operator of the system, it was concluded that indeed the manual was not accurate in the treatment of these conditions. No other error was found in the other properties we tested up to date.

7.2 Performance considerations

In this phase, an additional concern was related to the performance of the verification, and how different modelling approaches impacted on it. We mentioned

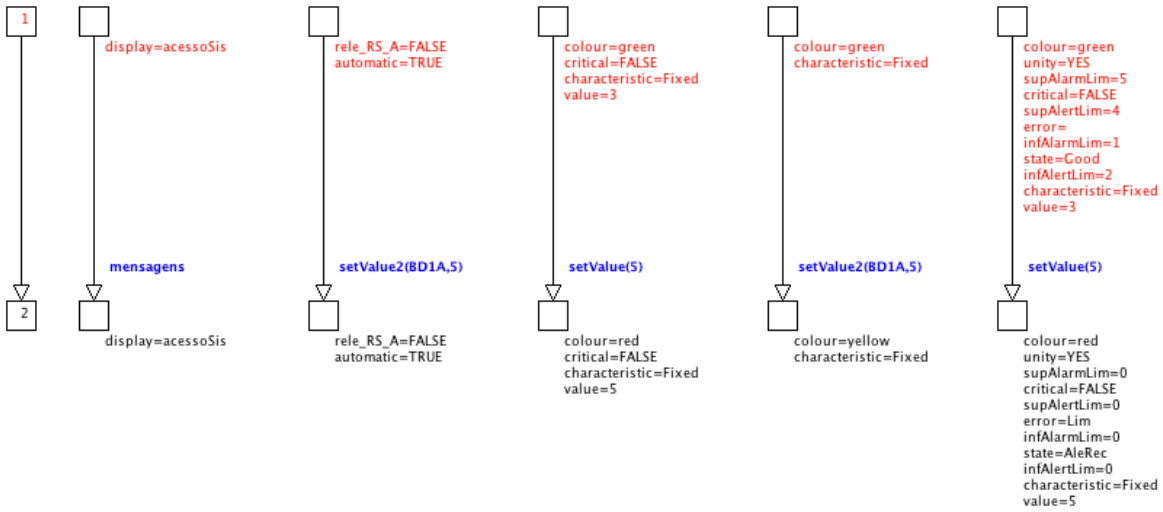


Figure 7.2: Trace of the counterexample.

in Section 4.4 the use of invariants *vs.* guards in the definition of how button colors and blinking characteristics change as the variables' values change. We experienced that invariants make the model checker consume more memory during the verification, as well as taking more time to perform it. Hence, we decided that, mainly in the action *setValue* of the *tmtVariable* Interactor, we should use guards instead of invariants.

We have described the simpler versions of the models (considering very few variables). Our experience adding more detail to the model shows that this is just a case of adding more variables and the corresponding axioms. Given the approach we have devised, this addition has not particular complexity in terms of writing the models. The CR model is relatively simpler than EV, mainly due to the simplification which has been done in the variables.

In order to explore the impact of adding variables to the CR model, multiple versions of this model (i.e. with different numbers of variables) were use to test the property:

AG(automaticMode=TRUE -> !EF(monitVOO.action = inicNavMSI))

The goal was to obtain a path, as counter example, that illustrates the steps of the automatic sequence (because of the size, the trace is not presented). This property means that for all paths where *automaticMode* is equal to *TRUE* (which

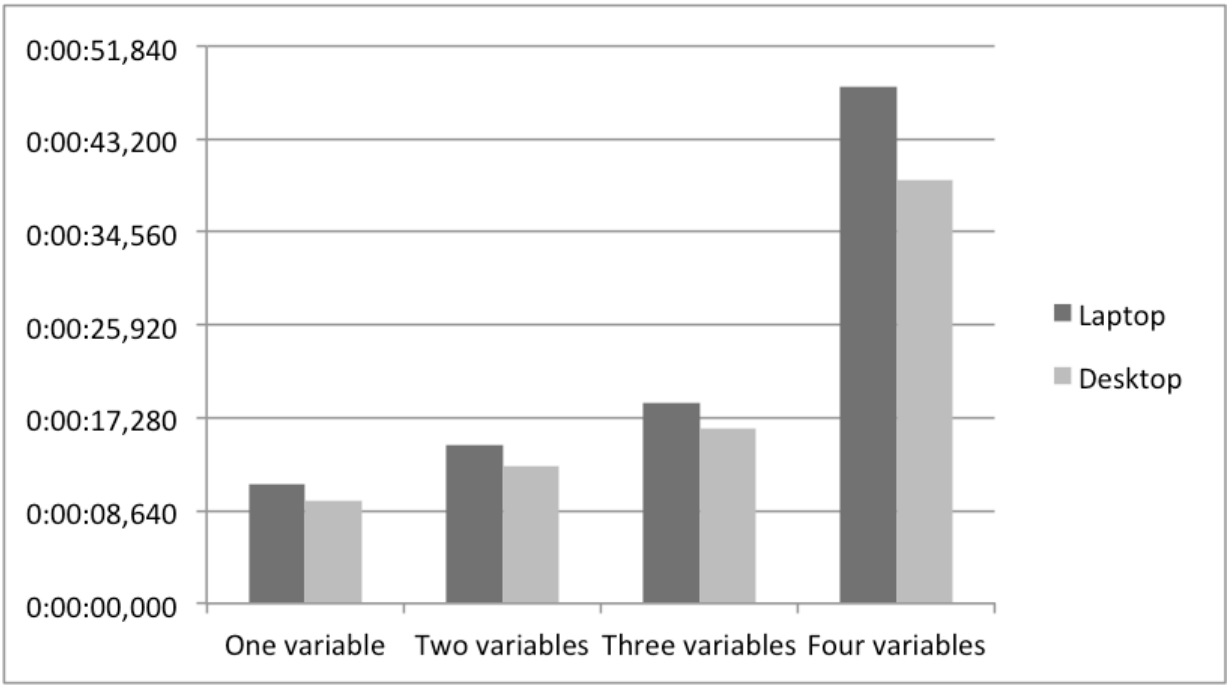


Figure 7.3: Verification times of CR model.

is always the case), there is no future state where the action of the *monitVOO* Interactor is equal to *inicNavMSI* (which is the last action of the automatic sequence). The verification times of this property in various model versions are illustrated in figure 7.3. The properties were tested on two machines. A Macbook Pro with an Intel Core 2 Duo P8800 at 2.66 GHz with 8Gb of ram, and a PC with an Intel Core i7 960 at 3.20GHz with 24Gb of ram. The machines have different operating system, Mac OS X and Windows Server 2008 R2 Standard respectively. The verification times were fairly quick, but we were expecting a higher difference between the machines. The biggest difference was almost ten seconds in the model with four variables. The results show that the addition of variables increases the verification time nonlinearly. In this model this is noticed especially from three to four variables, which increases verification time by almost thirty seconds in the Mac.

Three versions of the EV model were created, with two, three and four variables, respectively. First the property above was tested, in the two variables model, with Mac. The results are illustrated in Figure 7.4. The verification time was fairly higher than for the CR model. This was what we expected due to the model being

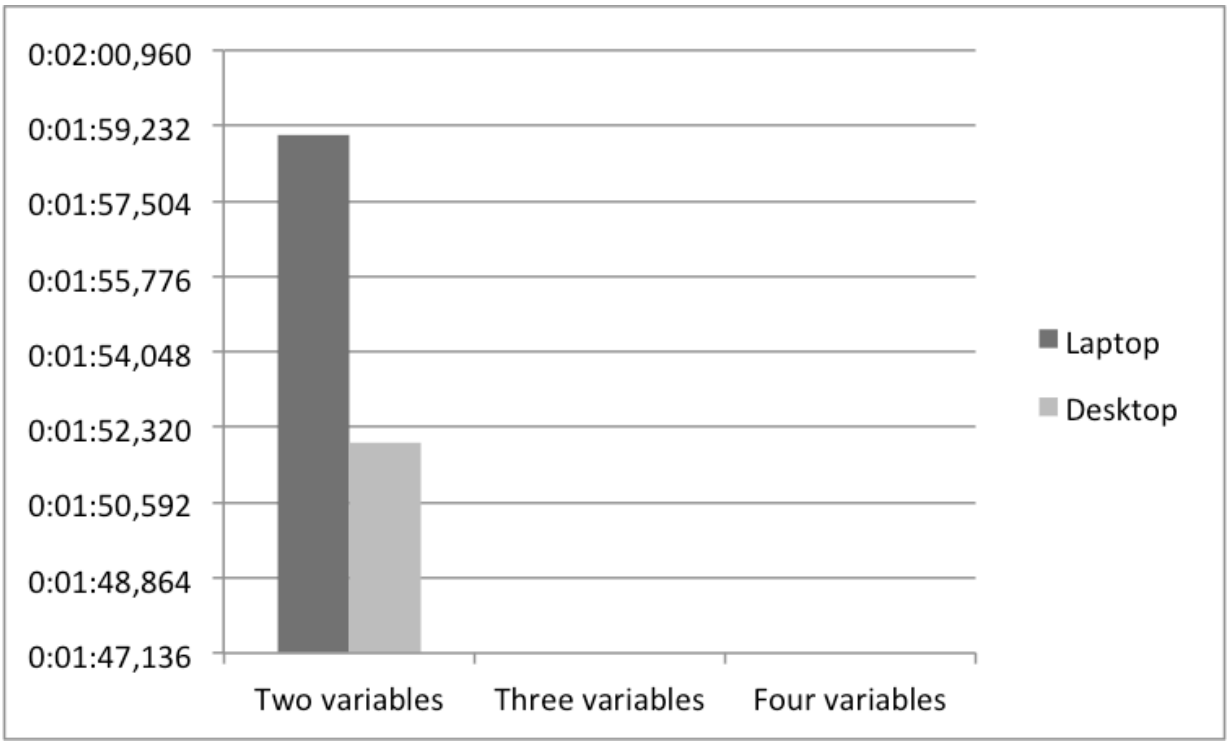


Figure 7.4: Verification times of EV model.

more complex (mainly because it included a more detailed treatment of variables). The three variables version of the model was in verification for 24 hours and did not produce any result. In fact, after a couple of hours the model checker was consuming the entire computer memory (8Gb) and CPU load was very low.

After this result we tried to verify the property in the PC with 24Gb of ram. In the PC the model was in verification during a weekend, and no result was produced. Like the Mac the model checker was consuming the entire computer memory (24Gb). Because of that, the EV model with three and four variables was not verified. This shows that applying appropriate abstraction to the models can have a huge impact on the feasibility of the analysis.

Although the addition of more variables does not add much to the system in terms of verification, because they all behave the same way, a model of the CR subsystem with 42 variables was developed. That model is available at the *HCIspecs* web site[1].

[1] http://hcispecs.di.uminho.pt

7.3 Tool analysis

From the modelling made in the IVY Workbench, a number of possible improvements to the tool were identified that could help when applying it to the modelling and verification of "large" and complex systems such as was the case here. Some improvements are language enhancements, which help to express more easily some of the logic of the system. Others are suggestions of new plugins/functionalities that can improve the verification functionalities of the tool.

7.3.1 Language enhancements and limitations

In models designed with an hierarchy of Interactors there is, most of the time, a need to synchronise Interactors (actions and variables), as was seen in Section 4.4. That synchronization is done with constraints, typically on the root Interactor of those we want to synchronise. The language already has some ways to simplify (reduce the complexity of) expressing the constraints. For example, the coordination expression:

```
per(monitTMT.recAlar(_v,_e)) -> effect(monitALAR.recAlarm(_v))
```

is a simplification of writing all the instances of *monit.recAlar* and *monitALAR.recAlarm*. This would be writing axioms such as:

```
per(monitTMT.recAlar(BD1A,AlaRec)) -> effect(monitALAR.recAlarm(BD1A))
```

for all the combinations of the action parameters values. But it is not possible to simplify this kind of expressions all the time, for example this expression:

```
per(monitALAR.setState(_b,_e)) ->
 (effect(monitTMT.setValue2(_b,0))
 | effect(monitTMT.setValue2(_b,1))
 | effect(monitTMT.setValue2(_b,2))
 | effect(monitTMT.setValue2(_b,3))
 | effect(monitTMT.setValue2(_b,4))
 | effect(monitTMT.setValue2(_b,5)))
```

The obvious simplification would be:

per(monitALAR.setState(_b,_e)) -> (effect(monitTMT.setValue2(_b,_v))

This simplification cannot be done because currently it is not possible to use a variable in the right side of the implication or equivalence, that is not in the left side. What we need is a simple cartesian product of the possible actions. In this example the simplification is not too significant, but in case of *_v* be from 0 to 100, the simplification is welcome.

In the case of *_v* ranging from 0 to 100, we may want to express parts of the set 0 to 100. As an example:

per(monitALAR.setState(_b,_e)) -> (effect(monitTMT.setValue2(_b,_v1))

per(monitALAR.setState(_b,_e)) -> (effect(monitTMT.setValue2(_b,_v2))

The first expression with *_v1* be from 0 to 49, and the second with *_v2* be from 50 to 100. One way to express this could be adding a *where* keyword to define the variable.

```
per(monitALAR.setState(_b,_e)) ->
 (effect(monitTMT.setValue2(_b,_v1)) where _v1 in {0..49}

per(monitALAR.setState(_b,_e)) ->
 (effect(monitTMT.setValue2(_b,_v2)) where _v2 in {50..100}
```

This solution is similar to the "syntactic sugar" used in set's inclusion with the keyword *in*. For non consecutive sets, for example a set ranging from 0 to 20 and ranging from 60 to 70, is easy to include the *union* keyword to express that (e.g. *_v1 in {0..20} union {60..70}*).

The synchronisation between Interactors is, in most of the cases, a necessity because of the tree-like compositional nature of the language as seen in Chapters 3 and 4. Figure 7.5 illustrates how direct communication between Interactors at the same level in the hierarchy is currently not possible to achieve. In the figure the variables *varA* from the Interactors *Interactor1* and *Interactor2* represent the same value. *invCoordination* represents coordination invariants, which in this case would coordinate the variables *varA* of *Interactor1* and *Interactor2*. Beyond that, we would also have to coordinate actions *setA* of those Interactors. A good example of the extra code to express this kind of communication is described in Section 5.3.4.

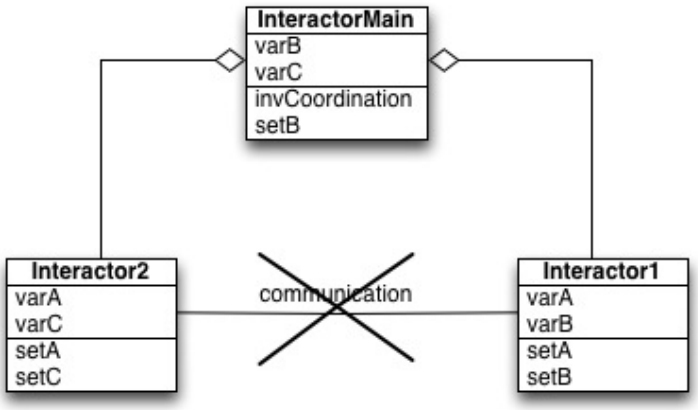


Figure 7.5: Communication between Interactors.

A solution to allowing communication between Interactors at the same level is not so simple as was a solution for the previous issue. The MAL model is compiled to *SMV* which has a tree-like compositional nature language, like MAL. A possibility would be to further explore the coordination based approach to composing Interactors proposed in [12, 13].

7.3.2 New tool features

As seen in Section 2.4 the IVY Workbench is a plugin-based tool. IVY can easily be extended with more features as plugins. One plugin that was developed was a simulator called *WildAniMal*. The simulator proved to be quite useful for checking basic aspects like our mental model against the model itself. Now it is an essential plugin to test paths that we think may have potential problems. Until now, we would have to define a formula that returns a counterexample with the path that we want to test, which most of the times is quite difficult.

A problem we face after modelling large systems, as the one modelled, is the verification time. IVY Workbench uses the symbolic model checker *NuSMV*[2]. As the model was growing, the verification time was growing in a bigger proportion. For large systems, theorem provers are typically used instead of model checkers [14]. They require less computational power, but are usually considered to require more

[2]http://nusmv.fbk.eu/

expertise from the part of the user, as the proofs are developed semi-automatically only (while in model checking the proof process is fully automated). Nevertheless, developing the models to a point where model checking is feasible (finding the right abstractions and the right encoding of the properties) is also a skillful process that requires very specific expertise.

Changing the tool, in this case the compiler, to compile to more than just a *SMV* specification would be a great improvement for the tool. A good solution would be the compiler generate different specifications for several model checkers or theorem provers. In the verification phase the user would decide which model checker or theorem prover to use. This kind of tool architecture is similar to SAL, and makes the tool more versatile. Some research is being made in order to generate PVS specifications to be applied to the PVS[3] theorem prover.

7.4 Conclusion

With the verification of properties we ended up finding an error in the manual. We were not expecting to find any kind of error, both because the modelling had been done from the manual, and because the system has been in use for many years. Unfortunately, it was not possible to verify the more complex models, due to the memory consumption of the model checker.

Concerning the expressiveness of the language, it was possible to model all the aspects of the subsystems. Still, some “syntactic sugar” could be added, in order to make some expressions simpler to write.

[3]http://pvs.csl.sri.com/

Chapter 8

Conclusions and Future Work

Typical Interactive Systems' evaluation techniques are just not enough for Safety Critical Interactive Systems. Such techniques can not verify all interactions, whether through forgetfulness, being manually applied, or because that would require too many resources. In many cases it will be simply impossible to check every interaction. We have explored how a mathematically rigorous method (model checking) can be the way to verify all the interaction, and to do that automatically. State space explosion is a serious problem with model checking, and abstraction of the system during the modelling phase is the way to reduce the state space to be checked. As mentioned in section 2, analysing a superset of the original system can achieve verification, which is what "traditional" methods can not do.

8.1 Results

We have applied IVY Workbench to two subsystems of the IAE's TPGS. Two different modelling strategies were used: building a "flat" model and building an hierarchical model; although the first was discarded early on, due to verification times. The language proved to be expressive enough to model the systems. However in some cases, such as invariants, it can become very verbose as the model grows and, with that, the model is difficult to maintain. The simulator that became available during the research project proved to be quite useful, especially in the initial phase of modelling. It is easier to validate the initial design by experimenting the model, rather than specifying CTL properties that return counterexamples that show the paths intended to validate the model.

In the verification of models, we ended up discovering an error in the manual. We were not expecting this kind of results, nor was it one of the objectives, but ended up reinforcing the importance and usefulness of such techniques. The verification times recorded showed that, in terms of verification, when the model becomes more complex times become intolerable. The solution lies in trying to verify small parts of the system. Nevertheless, it should also be mentioned that, since all variables have the same type of behaviour, the inclusion of more than three variables in the model does not had much in terns of verification.

Still regarding verification times, it was found that the *NuSMV* model checker only use a core in the verification. Since the machines used for verification had several cores, there is a lack of use of the existing computational power.

8.2 Future Work

Although the objectives have been met, the solutions are problems to be solved. The language is already robust, however the suggestions made will make it even more attractive and easier to use. Not being a laborious task, incorporating the suggested "syntactic sugar" should be a priority.

A sensitive part is the verification, which in large models becomes very time consuming and uses a lot of resources (memory). In *NuSMV* it is possible to rearrange the variables' ordering in the BDD, and this has an impact on the performance of the model checker. It would be interesting to explore how the rearranging on the variables affects the verification times.

NuSMV uses *Minisat* and *ZChaff* as SAT solvers. These SAT solvers do not use the various cores available. There are parallel SAT solvers (e.g *ManySAT*[1]). It is expectable that using a parallel SAT solver could considerably reduce the verification time.

[1] http://www.cril.univ-artois.fr/~jabbour/manysat.htm

Bibliography

[1] E. Barboni, D. Navarre, P. Palanque, and S. Basnyat. A formal description technique for interactive cockpit applications compliant with arinc specification 661. In *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pages 250 –257, july 2007.

[2] E. J. Bass, K. M. Feigh, E. Gunter, and J. Rushby. Formal modeling and analysis for interactive hybrid systems. In *Proceedings of the Fourth International Workshop on Formal Methods for Interactive Systems*, Potsdam, 2011. EASST.

[3] J. Berstel, S. C. Reghizzi, G. Roussel, and P. S. Pietro. A scalable formal method for design and automatic checking of user interfaces. *ACM Trans. Softw. Eng. Methodol.*, 14:124–167, April 2005.

[4] R. S. Boyer and J. S. Moore. A mechanical proof of the unsolvability of the halting problem. *J. ACM*, 31(3):441–458, June 1984.

[5] J. C. Campos. Using task knowledge to guide interactor specifications analysis. In *DSV-IS*, pages 171–186, 2003.

[6] J. C. Campos and M. D. Harrison. Model checking interactor specifications. *Automated Software Engg.*, 8:275–310, August 2001.

[7] J. C. Campos and M. D. Harrison. Interactive systems. design, specification, and verification. chapter Systematic Analysis of Control Panel Interfaces Using Formal Tools, pages 72–85. Springer-Verlag, Berlin, Heidelberg, 2008.

[8] J. C. Campos and M. D. Harrison. Interaction engineering using the ivy tool. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '09, pages 35–44, New York, NY, USA, 2009. ACM.

[9] E. Clarke, O. Grumberg, and D. Long. Model checking. In *Proceedings of the NATO Advanced Study Institute on Deductive program design*, pages 305–349, Secaucus, NJ, USA, 1996. Springer-Verlag New York, Inc.

[10] 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.*, 8(2):244–263, Apr. 1986.

[11] A. Dix, J. Finlay, G. D. Abowd, and R. Beale. *Human Computer Interaction*. Pearson, Harlow, England, 3. edition, 2003.

[12] L. S. B. M. A. Barbosa and J. C. Campos. Towards a coordination model for interactive systems. *Electronic Notes in Theorectical Computer Science*, (183: Proceedings of the First International Workshop in Formal Methods for Interactive Systems (FMIS 2006)):73–88, 2007.

[13] J. C. M.A. Barbosa, L.S. Barbosa. A coordination model for interactive components. In *Fundamentals of Software Engineering*, volume 5961 of *Lecture Notes in Computer Science*, pages 416–430. Springer-Verlag, 2010.

[14] M. Ouimet and K. Lundqvist. Formal software verification: Model checking and theorem proving. Technical Report, Målardalen University, March 2007.

[15] P. A. Palanque, R. Bastide, and V. Senges. Validating interactive system design through the verification of formal task and system models. In *Proceedings of the IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction*, pages 189–212, London, 1996. Chapman and Hall, Ltd.

[16] J. Rushby. Using model checking to help discover mode confusions and other automation surprises. *Reliability Engineering and System Safety*, 75(2):167–177, 2002.

[17] M. Ryan, J. Fiadeiro, and T. Maibaum. Sharing actions and attributes in modal action logic. In *Theoretical Aspects of Computer Software*, pages 569–593. Springer Verlag, 1991.

[18] H. Thimbleby. Contributing to safety and due diligence in safety-critical interactive systems development by generating and analyzing finite state

models. In *Proceedings of the 1st ACM SIGCHI symposium on Engineering interactive computing systems*, EICS '09, pages 221–230, New York, NY, USA, 2009. ACM.

[19] H. Thimbleby and J. Gow. Applying graph theory to interaction design. In *Proceedings of the 2007 Engineering Interactive Systems Conference*, pages 501–519, Berlin, 2007. Springer.