CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Florence Research

UNIVERSITA' DEGLI STUDI DI FIRENZE

Dipartimento di Sistemi e Informatica

Dottorato di Ricerca in
Ingegneria Informatica, Multimedialità e Telecomunicazioni
ING-INF/05
Ciclo XXV

TESTING AND VERIFICATION METHODS FOR SAFETY CRITICAL SYSTEMS

Daniele Grasso

Advisors

Prof. Alessandro Fantechi

Prof. Enrico Vicario

Ph. D. Coordinator

Prof. Luigi Chisci

ABSTRACT

This dissertation is the results of the experience at General Electric Transportation Systems (GETS). The Company is a railway signaling manufacturer that develops embedded platforms for railway signaling systems. The safety critical nature of these applications makes the verification activities extremely crucial to ensure dependability and to prevent failures.

At the end of 2008 GETS decided to introduce new verification and testing methods inside the company processes to ensure higher code safety and cost effectiveness at same time.

Traditionally in the railway context the unit test was the main technique adopted to detect design errors and ensure the correctness of the components before the final integration and validation phase. Testing activities normally require high costs and at the same time do not ensure that the software is completely free from errors. In the same context the need to evaluate functional correctness of applications before the final integration and validation phase persuaded the Company to investigate the applicability of Model Checking technique to verify railway applications. In this domain and with these research objectives collaboration between GETS and the Computer Engineering Department (D.S.I) of the University of Florence started.

This work reports the results obtained during this collaboration going through methods, process, results of experimentations in the verification and testing domain of safety critical applications.

TABLE OF CONTENTS

Introduction **1**

Chapter 1 .... **3**

1 BACKGROUND **3**

1.1 Safety Critical Systems 3

1.2 Formal Methods 5

1.3 Abstract Interpretation 9

1.4 Interlocking Systems 15

1.5 CENELEC Standard 16

1.6 Thesis Statement 19

Chapter 2 .... **21**

2 STATIC ANALYSIS - ABSTRACT INTERPRETATION **21**

2.1 Static Analysis – Polyspace Tool 21

2.2 Two-Steps Process 25

2.3 Case Study 1 – BL3 ATP Application 32

2.3.1 Generated Code - Handwritten Code 35

2.4 Case Study 2 – METRO RIO ATP Application 40

Chapter 3 .... **42**

3 MODEL CHECKING - INTERLOCKING **42**

3.1 Interlocking System Representation 42

3.2 Control Table Generator 46

3.3 NuSMV Model Checker 49

3.4 SPIN Model Checker 51

3.5 Results 53

3.6 NuSMV Complexity Study 57

***Chapter 4* 61**

4 SUMMARY AND DISCUSSION 61

4.1 Static Analysis Conclusions 61

4.2 Model Checking Conclusions 63

Conclusions 65

Bibliography 67

Appendix A 72

Appendix B 85

LIST OF FIGURES

Figure 1 – X and Y values representation 12

Figure 2 - X and Y type range domain 13

Figure 3 – X and Y range represented through abstract interpretation 14

Figure 4 – Table A.17 from the EN 50128 Standard 18

Figure 5 – Chromatic Semantic Legend 24

Figure 6 – Two Steps Process 26

Figure 7 – Example of unsafe statement (array out of bound exception) 28

Figure 8 – Example of unsafe statement (underflow exception) 30

Figure 9 – First Step Results, BL3 32

Figure 10 – Orange classes associated to the approximations 33

Figure 11 – Second Step Results, BL3 34

Figure 12 – Comparison of verification cost of BL3 against a comparable project 35

Figure 13 – First Step Results, Generated Code 36

Figure 14 – First Step Results, Handwritten Code 37

Figure 15 – Second Step Results, Generated Code 38

Figure 16 – Second Step Results, Handwritten Code 39

Figure 17 – First Step Results, Metro Rio 40

Figure 18 – Example of Control Table 47

Figure 19 – Example of translation from Control Table to SMV Model 48

Figure 20 – SMV Model Example 50

Figure 21 – PROMELA Model Example.. 52

Figure 22 – NuSMV Results 54

Figure 23 – SPIN Results... 55

Figure 24 - Reachable states for combinations of Input Var. and Equ. Number 59

Figure 25 - Reachable states for combinations of Input Var. and Equ. Length 59

Figure 26 - Reachable states for combinations of Input Var. and Equ. Length 60

Introduction

The adoption of modeling and formal methods technologies into the different phases of development lifecycle of safety critical systems is constantly growing within industry.

Industrial applications of formal methods and model-based development for railway signaling systems are discussed in many case studies. The Paris Metro [Ref. 1], the SACEM system [Ref. 2], and the San Juan metro [Ref. 3] are past and recent examples of successful stories about the usage of these technologies in the railway domain.

General Electric Transportation Systems develops embedded platforms for railway signalling systems and, inside a long-term effort for introducing formal methods to enforce product safety, employed modelling first for the development of prototypes [Ref. 4]and afterwards for requirements formalization and automatic code generation [Ref. 5]. GETS has adopted the Model Based Development (MBD) technology in an effort to deal with the growing scale and complexity of its applications. Within the new development context also the verification and validation activities have experienced an evolution toward a more formal approach.

In particular, the code-based unit testing process guided by structural coverage objectives, which was previously used by the company to detect errors in the software before integration, is the object of this dissertation together with the study of the applicability of model checking to the railway domain problems.

The rest of the dissertation is structured as follow. In chapter 1 are reported the state of

the art, the background and the thesis statement. In chapter 2 are reported the results obtained during the research studies applied at actual company's projects and the polyspace based process defined according to the results. In chapter 3 is described the interlocking problem chosen as railway domain application for the model checking analysis, the results obtained in terms of applicability of the method to the interlocking problem.

In chapter three summarize the results of the two main dissertation areas and the future work planned by the Company according to the results obtained and the continuous improvement process.

Chapter 1

1 BACKGROUND

1.1 Safety Critical Systems

The Safety Critical Systems are defined as the ones that are involved in a domain in which a malfunctioning can bring risks for the human health and for the environment in which they act.

Applications that shall be free from anomalies and that belong to the Safety Critical Systems class are common in avionic, railway or health context: in these business areas a software or hardware failure could raise serious damages in terms of human lives and serious problems and concerns for the environment.

Considering an entire System composed by hardware, software and environment in which it acts, it is possible to define a system level risk as a state of the system that can provoke an unexpected behavior that can drive to deaths, damages and contamination of the environment.

In these contexts a fundamental activity to be carried out before the release of a new product is the verification of the correctness by the point of view of design and

implementation: the aim is to verify and provide the evidences that the Product cannot bring to unexpected behaviors raising catastrophic failures.

Safety Critical Systems are typically constituted by hardware and software components. Safety cannot be achieved over the single software or hardware components but needs to be assessed also on logical and physical interactions.

Once during the years the doubts on robustness and capabilities to manage hardware failures for the software were ridded out, the use of the software components inside safety critical systems is gradually increased and propagated.

Software and hardware are used for different purposes and for their specific characteristic are involved in different ways and aspects in terms of safety. The common aspect that associated software and hardware in the safety critical context is the opportunity to move in a failure state. By this point of view the software can be considered as well as all the other components that can generate unsafe conditions.

The main difference, that is a remarkable advantage by the point of view of verification and testing, is that the software failures are deterministic: with a fixed state and a fixed inputs vector the software will fail every time and with the same effect (there are no degradation in software components and no impacts from environmental conditions).

To verify the correctness of a software procedure and assess its behavior for all the possible inputs and states means to have the absolute guarantee of the software procedure design and implementation and allows the procedure to be considered as an atomic safe operation.

Even if examples of accident caused by software failures exist, as the case of the Ariane 5 [Ref. 6] and the Therac 25 [Ref. 7], the amount of accidents attributable to software anomalies is minimal. In the aerospace context it is estimated the hour failure rate equal to 10^-7.

The verification phase of a software system is historically committed to a massive use of the testing activity that is the verification of the software correctness executing the source code with specific input stimuli (called also Dynamic Analysis).

This activity, even if very expensive, is not exhaustive. Verification and Validation activities of a system can exceed the 60% of the entire development cost.

It is easy to prefigure that these reasons incite the industries working in the safety critical domain to modify their processes introducing new techniques and new modeling design approaches in order to reduce the development costs and to increase the confidence about the products safety.

1.2 Formal Methods

The Formal Methods are a set of mathematically based techniques through which it is possible to specify, develop and verify hardware, software or entire complex systems. The more appropriate description associated with the term "formal methods" can be traced to that provided by the standard IEC 61508, in which the methods are defined as those that describe in mathematical form a system that can be subject to mathematical analysis for the detection of inconsistencies or errors . Generally a formal method is used to provide a notation and a technique to obtain a description

in this notation and various forms of analysis to verify different properties of correctness.

The application of these techniques along the software development process is highly recommended in the standard CENELEC 50128 particularly during the requirements specification and software design phases with the highest level of safety SIL 4. The methods use at the beginning of the process as their applications on the system requirments specification allows to deploy and to manage the efforts of verification. These are usually entrusted to the testing phase during and throughout the development cycle. The ability to perform a formal verification applied to the requirements specification and design phases therefore allows the improvement about the capability to detect at an early stage any logical errors that would later be transformed into implementation errors.

These techniques can be applied to all development phases of the process until the detailed source code verification phase. Each algorithm has some implicit properties that shall be checked during its execution, the formalization allows them the opportunity to verify its correctness.

Although the method has a high level of applicability independent of the functional purpose of the final system, in industry, the process of introduction, integration and application of formal methods for the development and verification of safety related systems is dependent by the level of technological innovation system, its support design tools and by the system project organization. This introduction process also depends on the typology of validation standard requirements that the project shall satisfy.

The presence of a large amount of different formalisms that the technical features and its related support tools for the verification implies the need to realize - for the company that would use the method - a detailed preliminary study antecedent its direct application.

Another factor that impacts on the introduction process about the formal methods use is the level of complex systems representation. The complex systems definition through syntax and semantics shall be very rigorous and robust.The quantification of the benefits in terms of costs as the assumptions listed above depends strongly by the structure, size and project organization. Despite this background, the present method has already found wide use in many companies.

The partial simplification of the processes of formal verification is reached with the introduction of new tools, automatically, perform the formal analysis of the models or specifications that describe them: the model checker. The number of existing model checkers and available following the number of existing formal methods. It is important to note that the negative outcome of the verification of a property for a model using model checking depends on this from both the real absence of coverage of the property from the model that by an error introduced in the modeling phase of the system and its relative properties desired in the formalism.

The evidence of error typology is suggested by the model checker that provides a counterexample in case of failure. The verification that the example leds the error detection belongs to the domain of the system leads to the statement that the property is actually not fulfilled by the system under analysis.

A model checker example is NuSMV, it is [Ref. 9] a tool that bases its operation on the properties representation to check in the logical form the formulas in CTL and LTL. The system modeling is carried out through the use of particular types of graphs, the Binary Decision Diagram (BDD), which allows representing the system state space in a contracted form in order to avoid a major obstacle to the adoption of the massive model checking: the explosion of the state space. Use of this formalization is primarily born in the hardware systems verification.

A model checker widely used, especially in situations where it is important to the description of concurrent processes, is SPIN [Ref. 10]. The system needs to be modeled according to a formalism similar to the C language, PROMELA and the properties to check shall be expressed in temporal logic. The Institute ISTI of CNR in Pisa, for example, carried out the study of a project in the railway environment using just the SPIN model checker [Ref. 11] concluded that the use of the model checker leads to the detection of some potentially serious errors, but as the modeling phase of the system is complex because of the need to formalize Interlocking time and the difficulty of managing the explosion problem states.

1.3 Abstract Interpretation

In the industrial domain the verification and validation phase of the Software components is historically performed through testing and static analysis activities. The dynamic analysis (testing) mainly focus on checking functional correctness against software requirements, boundary values analysis, control flow and data flow of the procedures to detect possible runtime errors and unexpected behaviors on the range limit of the variables. The Static Analysis aims to identify static properties of the code: as coding rule restrictions application, dead code, coding style, uninitialized variables and unused functions. Examples of techniques traditionally adopted for static analysis are data flow analysis [Ref. 12], program slicing [Ref. 13], constraint solving [Ref. 14], and, with an increasing spread in the latest years, abstract interpretation. To distinguish which are the differences between static analysis and dynamic testing we can refer to the fault-error-failure model in its most used form in the context of testing and verification of L. Hatton [Ref. 15].

The error is the origin of the fault, the failure occurs in the case where the system is exercised in correspondence of a fault. The fault is a static characteristic of the code while the failure is a dynamic property that only occurs during the execution of the code. The direct implication from the definitions above is that the faults included in the software code will not always raise a failure. Many studies were developed to estimate the amount of faults that drive a failure as consequence; E. N. Adams [[Ref. 16] estimated that 1/3 of the software faults bring to a failure after 5000 years of execution.

The goal of the static analysis is to detect all defects in the code not on the basis of the failures that have generated, but directly by analyzing the static properties of the code. Since many of the failures occur only with particular combinations of input, detecting defects in the code only through the study of failures (testing) is a technique that does not allow detecting them all. Some tools that perform static code analysis algorithm based their operation on abstract interpretation.

Abstract interpretation is a particular static analysis method that allows to infer dynamic properties of the code and to detect runtime errors and faulty states of the program without executing the code. The theory beyond this technology was presented by P. Cousot and R. Cousot [Ref. 17] [Ref. 18] [Ref. 19] in the 70s. The core idea of the theory is to define some approximation of the semantics of a program to obtain an abstract semantics. Formal proof of the program can be done at this different level of abstraction in which irrelevant details are removed to reduce the complexity of the verification process. The method defines an over-approximation of all the program reachable states in order to check all the possible program runs. If a property is satisfied for the analysed set then it is satisfied for the real domain of the program, a domain that represents a subset of the one verified. As one can infer from the theory, tools for abstract interpretation may lead to false positives, caused by the analysis of runs that do not belong to the real domain of the code, and normally these situations have to be checked manually.

The Abstract Interpretation provides the mathematic method to move from a real domain to a different one in which particular operations become feasible and exhaustive.
The elements composing a C code statement are not interpreted according to the programming language semantic but in a different domain.
Variables and runtime errors are constituted by a set of equations: the abstract interpretation can solve the equations describing runtime errors (detecting the presence of errors) using the mathematic data coming from the equations that represent the variables. The statement below as example:

$$X = X/(X-Y)$$

The statement above can generate a certain set of runtime errors (not initialized variables, overflow..) but to clarify how the static analysis through abstract interpretation works it will be analyzed the possibility of the occurrence of a zero division when the values of the two variables are the same.
In Figure 1 it is represented an example of the possible combination of values of the two variables X and Y.

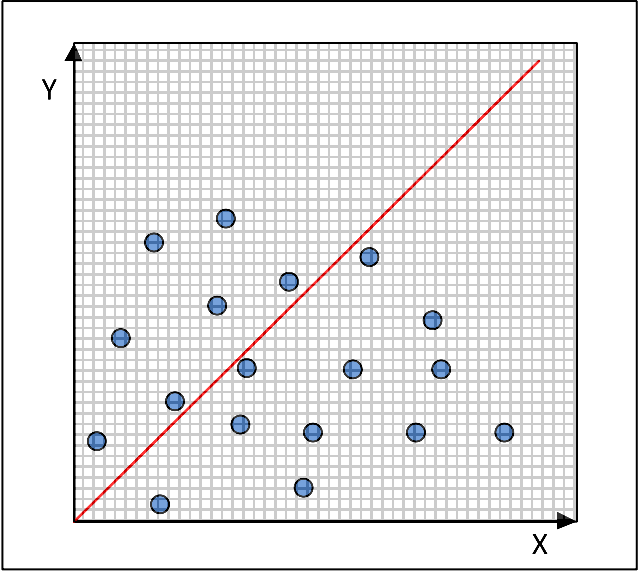


Figure 1 – X and Y values representation

The red line joins all the cases in which the statement raises a runtime error for zero division (variable X and Y assume the same value).

Compilers and support tool providing warning about erroneous use of variables use the type analysis abstraction. The tools select the minimum and the maximum value of the variables according the type of the variable and draw the correspondent square (Figure 2). If a property is true inside the square, it is valid also for all of the real possible combinations of the values that are included in the set. This kind of abstraction allows avoiding false negative cases but generates an high amount of false positives. In the example if only the blue circles are the combinations allowed according the data flow of the procedure, using the type abstraction a large set of

false positive will be generated (all the points in the red area) even if no blue circle resides on the line.

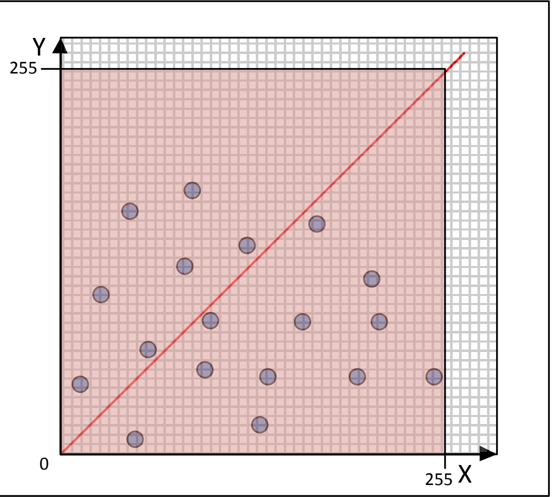


Figure 2 - X and Y type range domain

The abstract interpretation through the use of prisms, lattices and mathematical methods for data representation identifies the best shape and structure for the representation of the data. This abstraction allows to avoid many of the false positives points and allows to verify that no combinations of the data flow drive to a combination of X and Y that could raise the zero division runtime error.

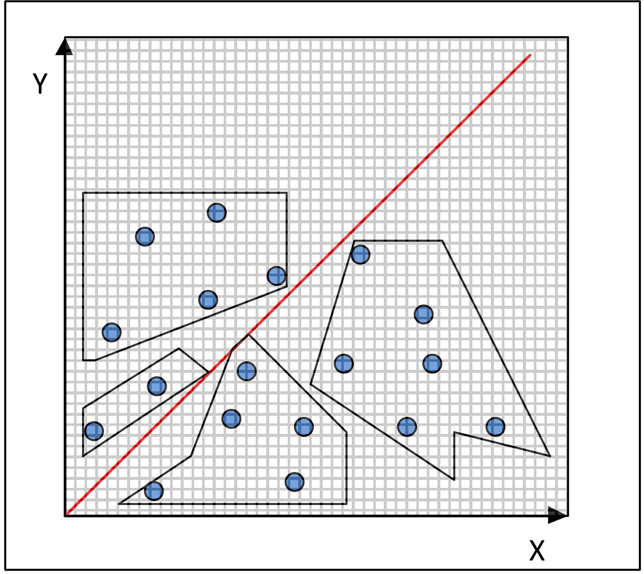


Figure 3 – X and Y range represented through abstract interpretation

The first experimentation with the Polyspace tool abstract interpretation based [Ref. 20] was related to the Mars Exploration Rover flight software from NASA. The results showed an high number of warnings that need to be checked manually requiring time consuming activities for the developer. In the avionics sector, successful experiments for the reduction of warnings [Ref. 21] have been performed using the tool Astrée [Ref. 22] currently distributed by AbsInt.

1.4 Interlocking Systems

In the railway signaling domain, an interlocking is the safety-critical system that controls the movement of trains in a station and between adjacent stations. The interlocking monitors the status of the objects in the railway yard (e.g., points, switches, track circuits) and allows or denies the routing of trains in accordance with the railway safety and operational regulations that are generic for the region or country where the interlocking is located. The instantiation of these rules on a station topology is stored in the part of the system named control table that is specific for the station where the system resides [Ref. 23]. Control tables of modern computerized interlockings are implemented by means of iteratively executed software controls over the status of the yard objects.

Verification of correctness of control tables has always been a central issue for formal methods practicioners, and the literature counts the application of several techniques to the problem, namely the Vienna Development Method (VDM [Ref. 24], property proving [Ref. 25] [Ref. 26]. Colored Petri Nets (CPN) [Ref. 27] and model checking [Ref. 28] [Ref. 29] [Ref. 30]. This last technique in particular has raised the interest of many railway signaling industries, being the most lightweight from the process point of view, and being rather promising in terms of effciency. Nevertheless, application of model cheking for the verification of safety properties has been

successfully performed only on small case studies, often requiring the application of domain related heuristics based on topology decomposition.

The literature is however quite scarce on data concerning the size of interlocking systems that have been successfully proved with model checking techniques. This is partly due to confidentiality reasons, and partly to the fact that the reported experiences refer to specific case studies, with a limited possibility of scaling the obtained results to larger systems.

1.5 CENELEC Standard

Products traditionally developed by GETS, like any railway signaling application developed for Europe, shall comply with the European CENELEC standards.

This is a set of norms and methods to be used while implementing a product having a determined safety-critical nature. We shortly refer in the following the ones that have a direct impact on the design of computer-based railway signaling equipments, and we focus in particualr on the one that regulates software design.

The EN 50126 "Railway Applications - The specification and demonstration of Reliability, Availability, Maintainability and Safety (RAMS)." addresses system issues on the widest scale.

The EN 50129 "Railway Application –Communications, signaling and processing system. Safety Related electronic system for signaling" addresses the approval

process for individual systems which can exist within the overall railway control and protection system.

The EN 50159-1 and 50159-2 addresses the approval process for communication, signaling and processing systems related to closed and open transmission systems.

The EN 50128 provides a set of requirements with which the development, deployment and maintenance of any safety-related software intended for railway control and protection applications shall comply. It defines requirements concerning organizational structure, the relationship between organizations and division of responsibility involved in the development, deployment and maintenance activities. Criteria for the qualification and expertise of personnel are also provided in this European Standard.

The key concept of this European Standard is that of levels of software safety integrity. This European Standard addresses five software safety integrity levels where 0 is the lowest and 4 the highest one. The higher the risk resulting from software failure, the higher the software safety integrity level will be.

This Standard identifies techniques and measures for the five levels of software safety integrity. The required techniques and measures for software safety integrity levels 0-4 are shown in the normative tables.

The Standard does not give guidance on which level of software safety integrity is appropriate for a given risk. This decision will depend upon many factors including the nature of the application, the extent to which other systems carry out safety functions and social and economic factors.

The norm encourages the usage of models and formal methods in every phase of the software development cycle, starting from the design to the verification. The rationale is that models are more related to abstract concepts than the technologies used for their implementation into code, and are therefore closer to the domain of the problem.

TECHNIQUE/MEASURE	Ref	SIL 0	SIL 1	SIL 2	SIL 3	SIL 4
1. Data Modelling	D.65	R	R	R	HR	HR
2. Data Flow Diagrams	D.11	-	R	R	HR	HR
3. Control Flow Diagrams	D.66	R	R	R	HR	HR
4. Finite State Machines or State Transition Diagrams	D.27	-	HR	HR	HR	HR
5. Time Petri Nets	D.55	-	R	R	HR	HR
6. Decision/Truth Tables	D.13	R	R	R	HR	HR
7. Formal Methods	D.28	-	R	R	HR	HR
8. Performance Modelling	D.39	-	R	R	HR	HR
9. Prototyping/Animation	D.43	-	R	R	R	R
10. Structure Diagrams	D.51	-	R	R	HR	HR
11. Sequence Diagrams	D.67	R	HR	HR	HR	HR
Requirements: 1) A modelling guideline shall be defined and used. 2) At least one of the HR techniques shall be chosen.						

Figure 4 – Table A.17 from the EN 50128 Standard

"Formal Methods" refer to mathematically rigorous techniques and tools for the specification, design and verification of software and hardware systems.

"Mathematically rigorous" means that the specifications used in formal methods are well-formed statements in a mathematical logic and that the formal verifications are

rigorous deductions in that logic (i.e. each step follows from a rule of inference and hence can be checked by a mechanical process.) The value of formal methods is that they provide a means to symbolically examine the entire state space of a digital design (whether hardware or software) and establish a correctness or safety property that is true for all possible inputs (Definition from EN 50128 standard).

Abstract Interpretation is not currently part of the recommended practices of the EN 50128 norm, since this has been published before Abstract Interpretation became a mature tool supported technique. However, due to the evident benefits that it can bring in terms of runtime errors detection, companies started practicing it as a completion of the verification process to enforce the safety of its products. The abstract interpretation approach studied and defined in this dissertation shows how the company has employed a commercial tool (Polyspace) in its application domain.

1.6 Thesis Statement

General Electric Transportation Systems is a railway signaling manufacturer that develops embedded platforms for railway signaling systems.
At the end of 2008 GETS decided to introduce new verification and testing methods inside the company processes to ensure higher code safety and cost effectiveness at same time.

Testing activities normally require high costs and at the same time do not ensure that the software is completely free from errors: the Company decided to investigate the introduction in its process of a more deep and incisive method to support the testing phase, in particular the static analysis through abstract interpretation. In the same context the need to evaluate functional correctness of applications before the final integration and validation phase persuades the Company to investigate the applicability of Model Checking technique to verify railway applications and in particular interlocking applications. In this domain and with these research objectives collaboration between GETS and the Computer Engineering Department (D.S.I) of the University of Florence started.

The thesis aims to address the following:

Evaluating and Introducing new Verification and Testing Methods in the Safety Critical Domain Processes.

The dissertation faced two different development phases:

- Verification in Design Phase through the study of the applicability of Model Checking to the Interlocking System Verification
- Verification in Implementation Phase through the study of the applicability of the Static Analysis through Abstract Interpretation to the railway domain software source code.

Chapter 2

2 STATIC ANALYSIS - ABSTRACT INTERPRETATION

2.1 Static Analysis – Polyspace Tool

Concerning the abstract interpretation phase (Static Analysis) of the verification process, GETS has adopted Polyspace [Ref. 8], a commercial tool provided by The MathWorks. From an industrial perspective, having the same producers for several tools employed in the development process (Stateflow/Simulink suite used in the Model Based Development Approach) gives more confidence on their compatibility, and simplifies the interface with the tool providers.

Polyspace analyses the C code and detects the statements that could produce errors during the execution of the code.

The tool presents its results through chromatic marks on the analysed code:

- green, if the statement can never lead to a runtime error;
- orange, if the statement can produce an error under certain conditions;
- red, if the statement leads to a runtime error in every run;
- grey, if the statement is not reachable.

The analysis of results has two different levels of difficulty. The green and red codes obtained can be interpreted immediately: green indicates that the code is totally free of runtime errors; the red code indicates that the code is suffering from a permanent runtime error that occurs whenever statement is executed.

For the analysis of gray it is necessary to check what may have been the causes that led to the execution of failure: this step may require a greater commitment being in the context of safety-critical application the absence of unreachable code. The analysis of the orange is the critical part of the audit tool PolySpace and generally it represents the critical stage for any type of verification that is based on abstract interpretation: the orange indicates that at least one computation run-time error has been detected.
The main runtime errors that static analysis can detect are:

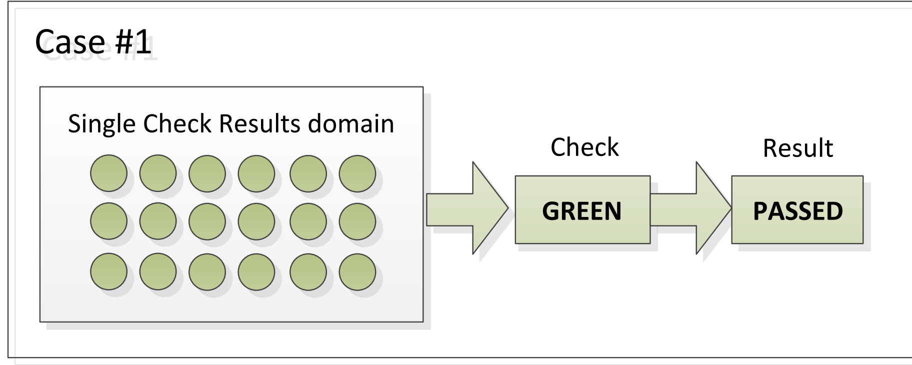
- access to uninitialized variables, local or otherwise (and NIVL NIV)
- access to uninitialized pointers (NIP)
- illegal access through pointers (IDP)
- access to an array out of bounds (OBAI)
- arithmetic overflow and underflow (OVFL, UVFL)
- infinite loops and calls that do not terminate (NTL and NTC)

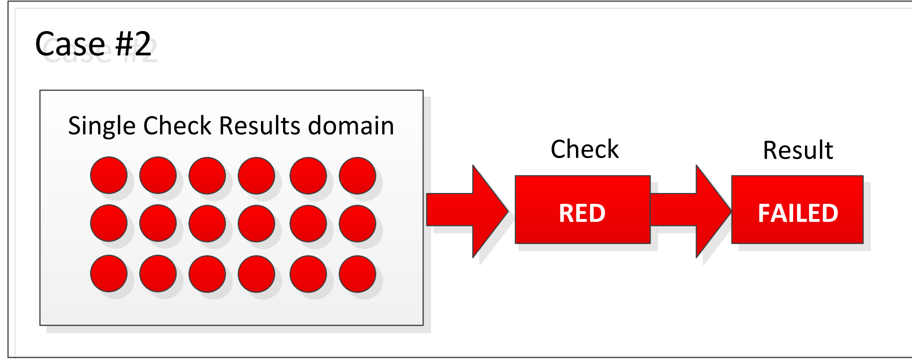
Every possible runtime error in the code is analyzed for its all possible computations. In Figure 5 there is described the criteria which PolySpace assign color codes to the code analyzed.

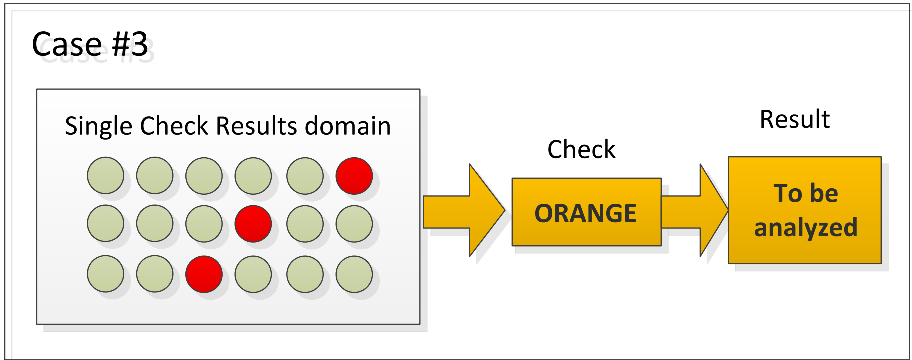
If there is the supposition that we have a statement that can generate a given run-time error, such as a OBAI

- If all computations do not generate the excess of the array then the check will be colored green;
- if all instead cause the error will be reported then a code red, otherwise;
- if at least one of computations generated the runtime error will be issued a warning orange

In order to perform static analysis on the code, the tools based on abstract interpretation techniques, such as Polyspace, build an abstract domain that represents an over-approximation of the real domain. The abstraction process might bring to the generation of false positives during the verification: this behaviour is caused by errors raised in those runs which are allowed only in the extended domain, but not in the original one.
For this reason, it is essential, for the adoption of this technique, to define a well-structured process that permits to reduce the cost of the analysis of false positives, a cost that represents the price to pay to obtain the exhaustive verification of the code behaviour.







Run for which the check does not raise runtime error

Run for which the check raises runtime error

Figure 5 – Chromatic Semantic Legend

2.2 Two-Steps Process

In order to address the problem and, at the same time, to obtain a substantial improvement of the confidence on the correctness of the code, the research lead to the definition of a two-steps process (Figure 6) promoted by the results obtained reported and discussed later in this chapter. The first step is performed with a very large over-approximation set. The second one capitalizes the information obtained by the analysis of the previous one, and executes the verification with the use of a finer approximation set [Ref. 33].

The purpose of the first step is mainly to detect systematic runtime errors (red), that is, errors which arise in all the runs considered in the verification, and unreachable statements (grey). Examples of systematic runtime errors are infinite loops, out of bound array accesses and usage of not initialized variables.

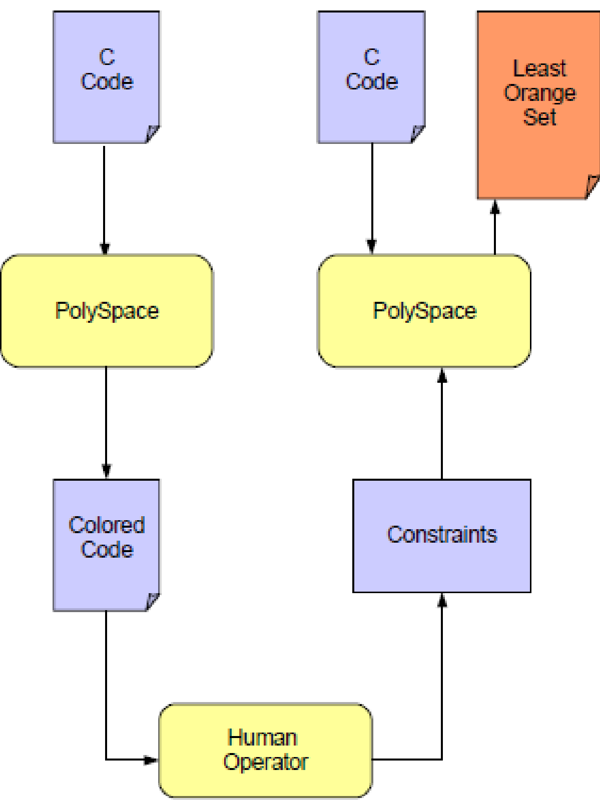


Figure 6 – Two Steps Process

Although unreachable code might seem to be a low severity problem, in our experience grey marks are often indications of erroneous modelling of the specifications: a code block that is never executed might be the translation of an unreachable state in a Stateflow chart. Only in some limited cases, grey marks are related to additional defensive-programming instructions introduced by the translator to maintain control even in presence of completely unexpected input, which may be due to hardware or software faults. For example, the default statement in a switch/case block (which is the natural translation of a state-machine), is likely to be

never executed, in which case it will be marked with grey. Obviously, these grey marks do not harm the safety of the code because they represent collectors used to handle unexpected behaviours.

The first step is performed using all the possible over-approximations settings provided by Polyspace:

- *Full-interleaving:* the tool automatically generates the function calls for the public procedures of the module under test if they are not invoked by other functions defined in the same module. All the possible interleaving of the automatic function calls are analysed in the verification.
- *Static variables initialization:* the static variables defined in the module in every run are initialized with all the values of their type range (in the following we will refer to this kind of approximation as the *full-range* initialization).
- *Global variables initialization:* the global variables defined in the module are managed in the same way of static variables.
- *Generation of function calls*: the formal parameters of the function for which the tool generates the call are initialized at full-range.

Since these approximations are used in the first step in order to be sure that the analysed runs include all the actual runs, the results obtained are not selective enough. The large set of spurious runs that are analysed in the step leads to an outstanding number of orange checks.

```
[…]
/* Global Variables Declaration */
int buffer[100];
int index;
[…]
/* Function Definition */
int get_value(int *buffer)
{
   return buffer[index];
}
[…]
```

Figure 7 – Example of unsafe statement (array out of bound exception)

For example, in Figure 7 the statement highlighted by the arrow could raise a runtime error, in particular the return value of the function *get_value* can be out of what is the expected: the statement could access a memory location which is outside the bound of the array named buffer.

The code reads a location of the array *buffer* indexed by the global variable *index*. In the first step, Polyspace automatically initializes all the global variables with full-range values. For this reason, when the tool analyses the statement highlighted in Figure 7, it finds that for some values of the variable *index* there is an out of bound access to the array. The result suggests that narrower bounds have to be introduced on the values that the variable *index* can assume in order to reduce orange marks and manage unexpected inputs in case of error propagating from other software components.

In the example, Polyspace signals a possible erroneous behaviour on the array bracket, but the actual cause of the orange mark is the full range initialization of the variable index. It is on this variable that one has to work in order to avoid the warning.

This situation is similar for any orange mark coming from the first step: in order to narrow the approximation for the subsequent step of Polyspace, each orange mark has to be related to the cause that produced it. The generic classes of causes that generate the orange marks are well known, and can be referred to the over-approximations settings that have been listed before. Therefore, an analyst with a minimum proficiency with the tool can easily evaluate the orange marks and quickly classifies their causes.

In the case of the example, the analyst recognizes the orange mark on a bracket as referring to the global variable initialization setting, so can pinpoint the variable that has been initialized full range. Another case might be the one in which one module has two interface functions, the first to initialize static variables, and the second to actually perform the functionalities required to the module (this is actually the normal structure of the automatically generated code). In the actual usage of the program, the initialization function will always be called before the other one. However, the tool will issue orange marks on all the static variables used by the execution function: due to the full-interleaving over-approximation, the tool assumes that the second function might be called before the initialization one, leaving the static variables without an initial value. Also in this case, the analyst recognizes a bunch of oranges on static variables, and can associate them to the full-interleaving class. Then, (s)he can add constraints concerning the order of execution of the functions, for consideration in the next step of the Polyspace application.

As exemplified, the identified classes are used to define input constraints to be given to the tool to restrict the analysed abstract domain of the program. Sometimes, editing the configuration file that defines the constraints might require advice from the developers, since the analyst is often not aware of the actual domains of the

variables, or of the program context in which a certain function is used. However, we experienced that the analyst is much more independent if (s)he has to deal with the automatically generated code, since the repetitive structure of the software simplifies the review task.

The second Polyspace step, performed with the restrictive settings, allows a finer approximation of the real domain of the program and then a reduction of the number of false positives. At the end of this step, the remaining orange marks are due to the complex interactions between variables that cannot be constrained by simply introducing finer approximation bounds.

As an example, consider the code segment depicted in Figure 8 that describes a typical software procedure present in the railway signalling context: it deals with a train receiving messages from the car-borne equipment at every given distance, in proximity of a so called information point. Every time the train passes by the information point and receives a message, the code assigns the current value of the space covered by the train, maintained in the variable current_space, to the variable last_msg_space. Once the train gets by the information point, it uses the procedure in Figure 8 to compute the space covered from the last message received.

```
[…]
/* Function Definition */

uint32 compute_distance(uint32 current_space, uin32 last_msg_space)
{
   return (current_space - last_msg_space);
}
[…]
```

Figure 8 – Example of unsafe statement (underflow exception)

Polyspace produces an orange mark that signals the risk that the described statement could raise an underflow. This orange mark is reported also in the second step of the Polyspace verification, because in this case the constraint on the possible values assumed by the variables does not handle the particular bound that makes impossible the underflow.

According to our experience, the overall time employed for the configuration and set-up activities is 20% more than the time Polyspace takes to actually execute the two steps. The most time consuming task is the first review of the orange marks that takes about the 48% of the overall time required for the whole process. Due to the low number of residual oranges after the second step (normally about 2.6% of the total), the cost of the second review is basically negligible. Nevertheless, one has to consider that the absolute overhead of the orange review is acceptable: about 5 minutes for each orange, in average. The generated code is characterized by a limited number of different classes of motivations for the orange marks, and this makes most of the review a rather systematic activity.

2.3 Case Study 1 – BL3 ATP Application

The approach described has been experimented in the verification phase of a project concerning an Automatic Train Protection (ATP) system developed by GETS in 2008 [Ref. 33]. ATP systems are embedded platforms aimed to control the train speed according to the wayside signals and brake the train in case of SPAD (Signal Passed At Danger), which is known to be a common cause of railway accidents.

Results on the abstract interpretation phase are reported for a representative set of project modules (Figure 9).

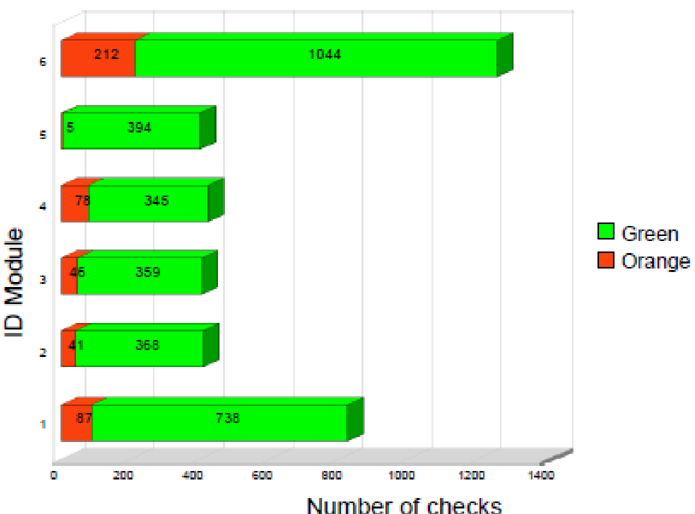


Figure 9 – First Step Results, BL3

As shown in Figure 3, no systematic error (red) has been detected during the first PolySpace verification. Nevertheless, there is a relevant amount of orange marks for which it is not possible to decide if they actually represent faulty states of the program. These orange warnings have been classified according to the kind of approximation that supposedly produced them. Manual analysis of the first results has detected only two classes of causes of oranges: wrong interleaving of function calls and automatic initialization of global variables and input function parameters (Figure 10).

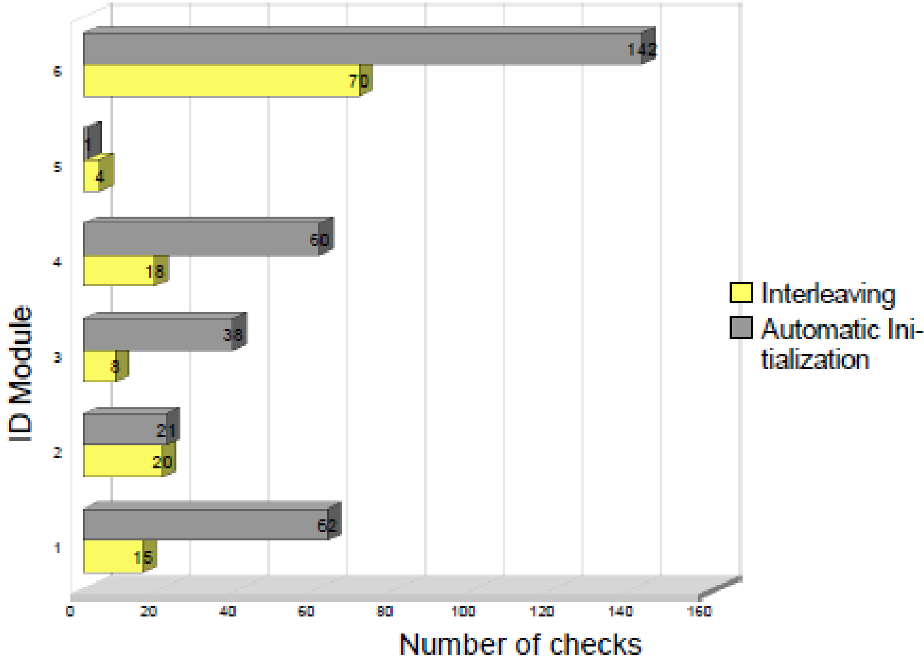


Figure 10 – Orange classes associated to the approximations

The analysis of these causes has determined the constraints for the second PolySpace verification. This step produced only a few orange warnings, as shown in Figure 11.

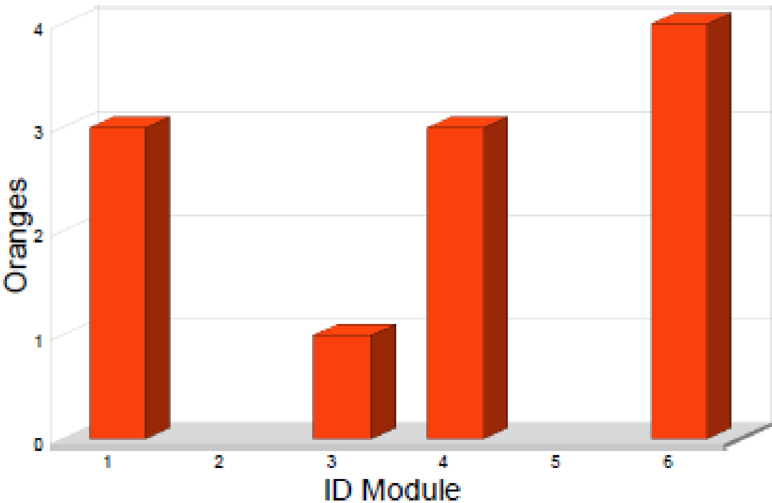


Figure 11 – Second Step Results, BL3

The remaining orange marks are due to complex interactions of variables that cannot be constrained by finer approximation bounds. However, an analyst with a sufficient knowledge of the actual meaning of the variables can quickly check if the warnings are false positives or not.

The Polyspace-based verification approach permitted a reduction of the overall verification cost of 70%, as reported in the Figure 12 compares the verification cost of the BL3 project to the effort spent for traditional structural testing on code (according to 100% boundary-interior path coverage), which was applied in a previous project of comparable size in terms of modules.

Verification Process	Modules	Paths	Hours
Structural Testing	19	2274	**728**
MBT + Abstract Interpretation	21	>8000	**227** (162 + 75)

***Figure 12* – Comparison of verification cost of BL3 against a comparable project**

The Verification Process includes the Model Based Testing performed in the design phase of the Stateflow Model from which the source code was generated.

2.3.1 Generated Code - Handwritten Code

GETS has adopted the Model Based Development (MBD) technology in an effort to deal with the growing scale of its applications. Model Based Development (MBD) is a software development approach where the fundamental artifacts are models.
Before getting into hand crafted code, the developer has to produce one or more abstract specification of the system in the form of models. Given this specification, software tools can provide simulation of the model behaviour and automatic code generation, this allowing a notable improvement for the process productivity.

This techniques drove as consequence that the software source code composing the railway application developed was constituted by a part of source code automatically generated and a remaining part that is traditionally handwritten by the developer. Considering the main characteristics of these two different types of source code, a

study was conducted also to identify the applicability and the performance of the static analysis through abstract interpretation for both type of software.

The results obtained shows that the generated code produces a higher number or false positives in the first step of polyspace based verification process (approximately 87% of green checks, Figure 13) then the ones produced for the handwritten code (approximately 92 %, Figure 14).

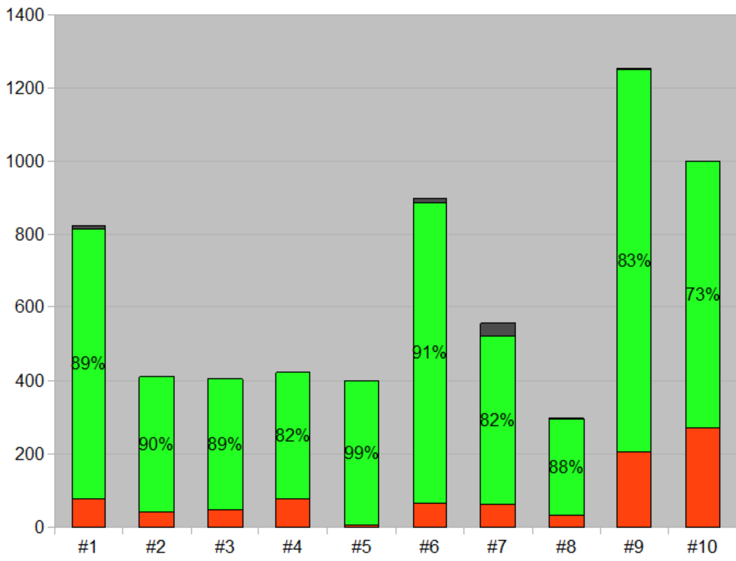


Figure 13 – First Step Results, Generated Code

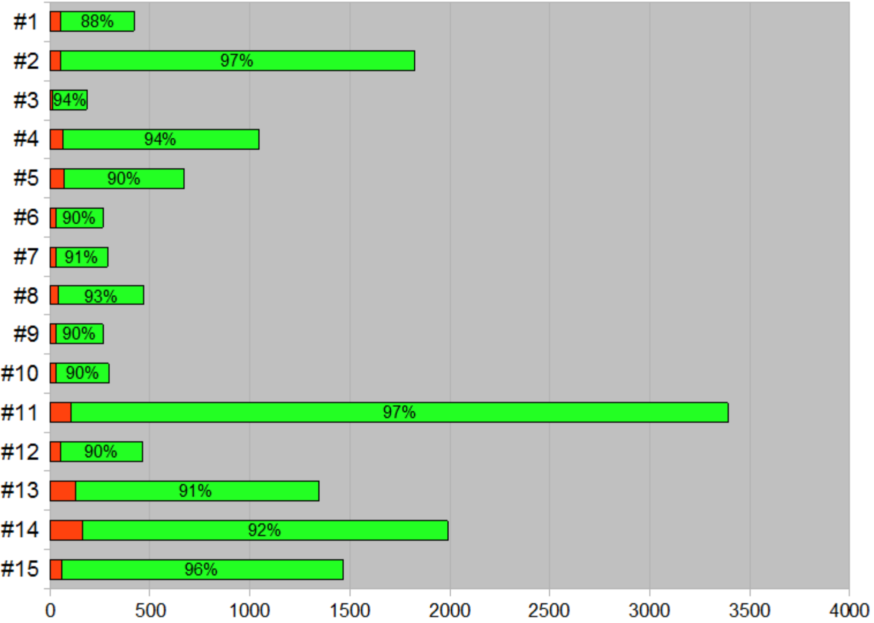


Figure 14 – First Step Results, Handwritten Code

The main difference during the application of the two steps process for the different source code types is identified in the constraints identification phase and in the second step results analysis.

In the case of generated code the orange marks are always caused by the same over-approximations and can be discarded in a systematic way: the two-steps process applies effectively in these situations, and eases the actual error discovery. If it appears that an orange does not belong to the classes of causes common to the generated code, it is likely to be an error.

In the case of handwritten code, the verifier has to inspect the code to understand which are the data used as input by a function (i.e., data that are only read) and those one that are output (i.e., data that are written). Furthermore, very often pointers to functions and void pointers are used in the hand-crafted code, and these are difficult

to be constrained in a way that can ensure an high reduction of false positive in the second step of polyspace based verification process.

The considerations above are supported by the results obtained for the second step of the process.

The results of the generated code after the addition of the constraints on the input parameters of the source code interfaces show that all the checks are green: all the false positive were identified and excluded during the second part of the process and the source code can be considered free from bugs (according to the assumptions related to the functional correctness of the model that generated them and correctness of the constraints added to perform the second step).

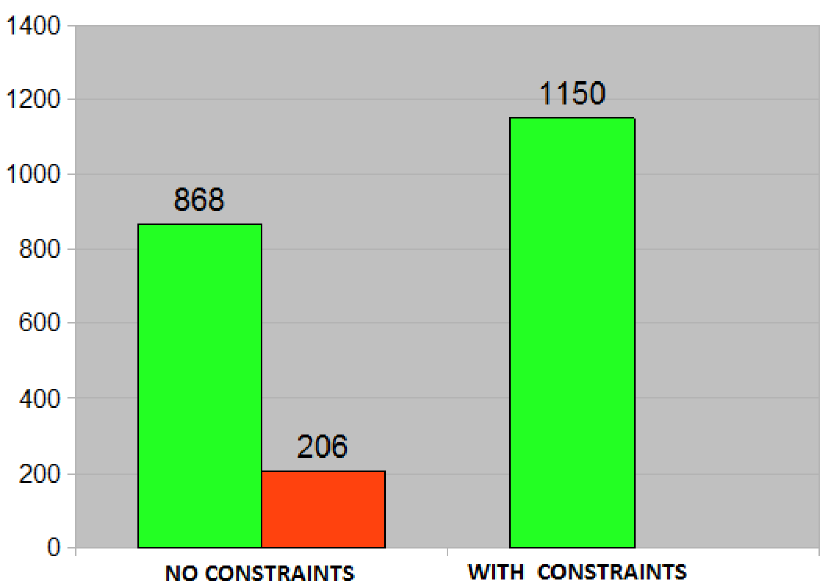


Figure 15 – Second Step Results, Generated Code

After the addition of the constraints added by the analyst on the basis of the first step, the results on the handwritten code, show that a remaining part of the false positive

are still produced. The nature of these checks is related to the use of absolute address for low level drivers, access to hardware register, use of pointers. These checks cannot be excluded, requiring a manual analysis.

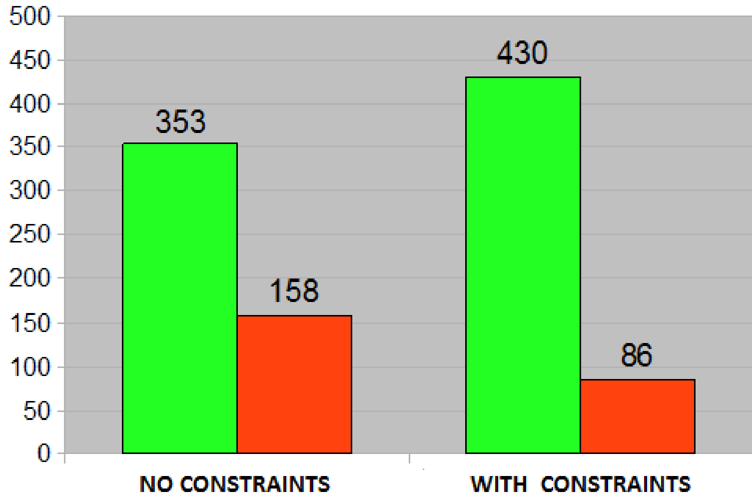


Figure 16 – Second Step Results, Handwritten Code

2.4 Case Study 2 – METRO RIO ATP Application

A more recent project developed by GETS is the ATP for the metro of Rio de Janeiro. The modifications introduced in the design phase [Ref. 34] of the Metrô Rio ATP application did not have a negative impact on the abstract analysis phase : the results obtained by the execution of the first Polyspace step shows a similar behaviour with respect to the ones obtained in the context of BL3 project, as depicted in Figure 17.

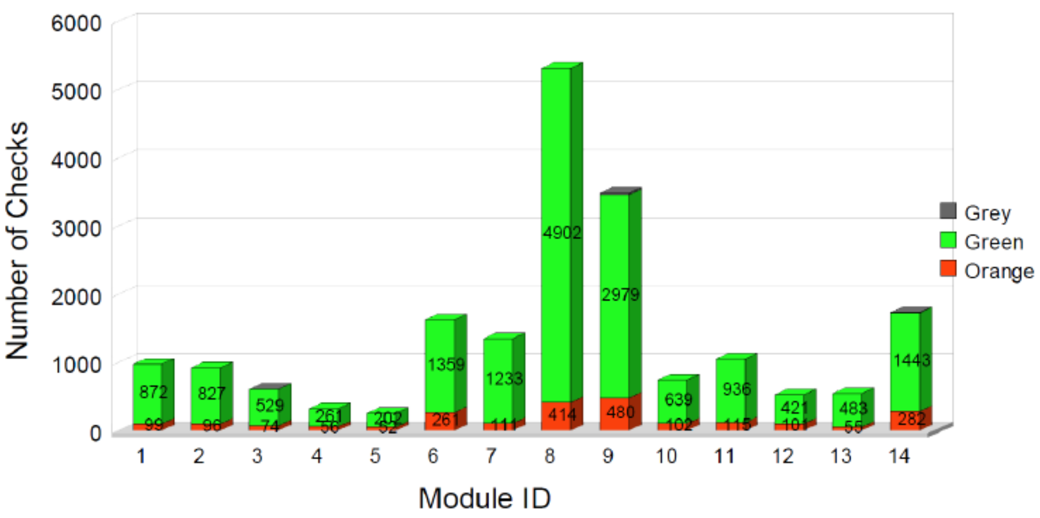


Figure 17 – First Step Results, Metro Rio

Although many oranges have been detected, thanks to the characteristics of the generated code, it has not been time expensive to classify these warnings according to the kind of approximation that supposedly produced them. Indeed, due to the

disciplined use of modeling guidelines, the generated code has a high number of simple structures and has well-defined module interfaces, which has helped to confine the causes of orange marks to the two only classes, already mentioned, of wrong interleaving of function calls and automatic initialization of input function parameters.

The second step of the Polyspace-based verification process has led to only few orange warnings, and most modules turned out to be entirely green. The results have been compared with the ones obtained on the previous project where Polyspace was first applied, but where modeling guidelines were less restrictive.

As in the previous project, the oranges detected in the first step are approximately 15% of the total number of checks for each module, but the time spent to classify the oranges and to determine the constraints for the second step have been considerably reduced thanks to the well defined structure of the generated code [Ref. 35] [Ref. 36] [Ref. 37].

Chapter 3

3 MODEL CHECKING - INTERLOCKING

3.1 Interlocking System Representation

In Relay Interlocking Systems (RIS), currently installed and operating in several sites, the logical rules of the control tables were implemented by means of physical relay connections. With Computer Interlocking Systems (CIS), in application since 30 years, the control table becomes a set of software equations that are executed by the interlocking. Since the signaling regulations of the various countries were already defined in graphical form for the RIS, and also in order to facilitate the representation of control tables by signaling engineers, the design of CIS has usually adopted traditional graphical representations such as ladder logic diagrams [Ref. 38][Ref. 39] and relay diagrams [Ref. 40]

These graphical schemas, usually called principle schemata, are instantiated on a station topology to build the control table that is then translated into a program for the interlocking.

As pointed out in [Ref. 26], the graphical representations and the related control tables can be reduced to a set of boolean equations of the form xi := xj ^ : : : ^ xj+k, where xj : : : xj+k are boolean variables in the form x or :x. The variables represent the possible states of the signalling elements monitored by the control table: system input, output or temporary variables. The equations are conditional checks over the current and expected status of the controlled elements.

In order to give a metric to the dimension of the problem in terms of parameters of the control tables, we define the size of a control table as the couple (m; n), where m is maximum number of inter-dependent equations involved, that means equations that, taken in pairs, have at least one variable in common, and n is the number of inputs of the control table.

We consider only inter-dependent equations because, if there are sets of equations that are independent, they can be verified separately, and slicing techniques such as the ones presented in [Ref. 23] and [Ref. 41] can be adopted on the model to reduce the problem size. In our experiments we basically consider control tables that have been already partitioned into slices (the size value of a control table is intuitively the one of its maximal slice).

Correctness of control tables depends also on their model of execution by the interlocking software. In building CIS, the manufacturers adopt the principle of "as safe as the relay based equipment" [Ref. 27], and often the implemented model of execution is very close to the hardware behaviour. According to the semantics of the ladder diagrams traditionally used for defining the control tables, we have chosen a synchronous model with global memory space where variables are divided into input, output and latch (i.e.,

local) [Ref. 26]. The model of execution is a state machine where equations are executed one after the other in a cyclic manner and all the variables are set at the beginning of each cycle and do not change their actual value until the next cycle. This is a reasonable generic paradigm for centralized control tables.

Timer variables are not considered in our models, since they are normally related to functional requirements of the system (e.g., the operator shall press the button for at least 3 seconds to require a route). Safety requirements such as the ones considered in this study are normally independent from timers. An intuitive argument in support of the fact that timers are not used to implement safety functions is that, in traditional RIS, timers were implemented by means of capacitors: these are components that have a rather high failure rate, making them unsuitable for safety functions.

We have developed a tool (see paragraph §3.2) that generates a set of equations coherent with this model of execution, expressed as models suitable for automatic verification with NuSMV or SPIN, and which represent typical control tables of parametric size.

Given a control table representation we want to assess that its design is correct. In the proposed experiment, we need to check that safety properties are verified, and this represents the worst case for a model checker: explicit and symbolic model checkers are challenged by verification of safety properties, since, in order to show their correctness, they have to explore the entire state space, or its symbolic representation. Safety requirements typical of signaling principles are normally expressed in the principle schemata or in the regulations.

This kind of properties have shown to be representable in Computation Tree Logic (CTL) in the CTL-AGAX form: AG(p ! AXq), where p and q are predicates on the variables of our model [6]. CLT-AGAX formulae have an equivalent Linear Temporal Logic (LTL) representation. The formula AG(p ! AXq) can be expressed in LTL syntax as a LTL-GX formula of the form G(p ! Xq). Intuitively, they represent fail-safe conditions, i.e., events that should happen on the next state if some unsafe condition occurs. One of the typical safety properties that is normally required to be verified is the no-derailing property: while a train crossing a point the point shall not change its position. This typical system level requirement can be easily represented in the AGAX form [14]:

$$AG(occupied(tci) \wedge setting(pi) = val \ ! \ AX(setting(pi) = val))$$

whenever the track circuit tci associated to a point pi is occupied, and the point has the proper setting val, this setting shall remain the same on the next state.

In order to force the worst-case full state space exploration, our test set has been designed on purpose to satisfy given properties expressed in CTL-AGAX (or LTL-GX) form, and model checking has been performed using these properties as formulae. Though not clearly evident, also for symbolic model checking we have experienced that satisfied invariants are the hardest problem.

3.2 Control Table Generator

The model of execution adopted (described in §3.1) is a state machine where equations are executed one after the other in a cyclic manner and all the variables are set at the beginning of each cycle and do not change their actual value until the next cycle. We developed a tool for control table generation that is in line with the execution model described.

The tool gets as input some configurable parameters and provides as output the SMV and PROMELA models of the control table set. The two model are built on the same control table information than are a valid criterion of comparison between the reaction of the two model checkers to the same problem.

The three main items of the control tables are the input variables, latch variables and output variables.

The input variables are only used to determine the value of the output variables: the values of the input variables are random and exhaustively initialized and modified by the Model Checker. Each input variable can only be associated to an input column of the tables.

The output variables are identified in the last column of the tables and for each output exists only one table that define its value.

The latch variables are the output variables that are also used as input for other tables: these variables are the state variables of the model represents information about the internal status of the modelled system.

In the following we will define that an input or a state variable support an output variable y if the table for the variable y includes as input the variable x. According the defined approach, the set of all the control tables define a finite state machine: in particular each table represents the transitions for the output variable related.

```
*****
tb0
*****
in0     in1     in2     ot7     ot16    ot14    ot0
0       0       0       1       0       1       1
1       0       1       x       x       0       1
1       0       0       x       x       x       1
x       x       x       x       x       x       0
```

Figure 18 – Example of Control Table

In the Figure 18 is shown an example of control table produced by the tool.
The table evaluates the values assumed by the output variable *ot0* in correspondence to the values assumed by the input variables *in0, in1, and in2* and assumed by the state variables (output variables for other control table) *ot7, ot16 and ot14.*
The symbol *X* specifies that in that particular case (the row of the table) the value of that variable does not influence the value of the output variable.

The tool populates with random values the tables except for the last row of each table that represent the "default" case of the table: if no one of the rows are satisfied by the actual inputs value, the last row represents all the combinations that does not activate the output (all the values that drive the

output at 0). The last row according the modelling approach is populated with all X (don't care) for the inputs and 0 for the value of the output.

In Figure 19 is reported the equivalent SMV model for the table showed above.

```
next(ot0) :=
 case
  !in0 & !in1 & !in2 & ot7 & !ot16 & ot14 |
  in0 & !in1 & in2 & !ot14 |
  in0 & !in1 & !in2 : TRUE;
  TRUE : FALSE;
 esac;
```

Figure 19 – Example of translation from Control Table to SMV Model

The input variables of the model can be considered as the states of the objects in the railway yard or the statuses of the requests coming from the centralized computer center of a station that ask for specific routes reservations for example. The logic engine of the interlocking system evaluates and allows executing the commands coming from the centralized computer according the statuses of the other objects in the yard and if needed changes the status of them according the computed values of the output variables of the control tables.

The control tables generated by the tool and the structure of the models generated can be considered generic control table of interlocking applications.

3.3 NuSMV Model Checker

NuSMV is an open-source symbolic model checker that provides the user with both Binary Decision Diagrams (BDD) based implicit model-checking and SAT solver based bounded model checking. Properties are encoded in CTL in the first case, and in LTL in the second case. Since we focus on the verification of safety properties, we need to be sure that every single reachable state is analyzed by the model checker; for this reason, we have not used NuSMV bounded model checking[1].

In NuSMV, the state is represented by the value of state variables. The next state is computed by first calculating the next values of state variables and then, atomically, updating all the state variables. This behaviour of the model checker is compliant with the chosen model of execution. Every equation is hence evaluated in sequence but the outputs are updated at the end of the whole evaluation phase. This behaviour permits to be free from the order of evaluation of the equations. NuSMV supports open models that mean that it computes all possible input variable values automatically: in its internal modeling language the keyword IVAR must be used for such variables. Input variables do not contribute to expand the state space of the system, but influence the number of reachable states. The variables under the keyword VAR are indeed state variables: the value of each of them in the model is determined by the evaluation of the conditions.

In Figure 20 is represented an extract of a NuSMV model used for our case study.

---

[1] Altough there are techniques that are able to guarantee in some cases the full exploration of the state space with bounded model checking, these have not been used in these experiments and could be the subject of further experiments.

An example of a CTL-AGAX property that is verified on this SMV model is:

AG(out0=0 ! AX(out1=0))

```
MODULE main
IVAR
in0:boolean;
in1:boolean;
[..]
VAR
out0:boolean;
out1:boolean;
ASSIGN
[..]
next(out0):= !in2 & in3 |
             !in4 & in5:1;
             1:0
next(out1):= !in2 & in3 |
             !in6 & out0:1;
             1:0;
[..]
```

Figure 20 – SMV Model Example

3.4 SPIN Model Checker

SPIN is a generic verification system based on explicit model checking. It performs a full search in the state-space to find whether or not a given set of system properties, that are expressed using Linear Temporal Logic (LTL), are satisfied.
The systems are modeled by the verification language PROMELA (PROcess MEta LAnguage), a C-like language, that provides instruments especially for the modelling of distributed asynchronous systems.
In this section we will distinguish between input and state variables of the system that is modeled using PROMELA: this distinction is just conceptual, since every variable in the model is in fact a state variable. SPIN updates the state variables on the fly, after the evaluation of the conditions of each equation; since we want to model a state machine compliant with the chosen execution model, we had to add to the PROMELA model a number of temporary variables equal to the number of actual state variables, that are updated only after the evaluations of all the equations, at the end of each processing step. A LTL-GX property, corresponding to the CTL-AGAX one given for NuSMV, that is verified on this PROMELA model is:

$$\square(out0=0 \; ! \; X(out1=0))$$

An example of model fragment written in PROMELA, corresponding to the one shown for NuSMV, is represented in Figure 21.

Since SPIN can only verify closed models, we had to model the environment behaviour in the PROMELA model: in order to model the non-determinism of the input variables values, we need to insert at the end of the PROMELA model an if statement for each input variable, as represented in the figure.

```
[..]
if
::((in2==0 & in3==1) |
   (in4==0 & in5==1))->tmp0=1;
::else tmp0=0;
fi;
if
::((in2==0 & in3==1) |
   (in6==0 & out0==1))->tmp1=1;
::else tmp1=0;
fi;
[..]
out0=tmp0;
out1=tmp1;
[..]
if
:: true -> in1=0;
:: true -> in1=1;
fi
[..]
```

Figure 21 – PROMELA Model Example

3.5 Results

In order to investigate the actual applicability bounds for the model checking of interlocking systems, the experiments were performed on generated models with different equations-inputs ratio. For each combination of inputs and equations at least three different generic control table models were generated and tested, in order to be able to avoid erroneous positive results caused by the generation of trivial equations: if at least one of the three model is not verifiable the whole class of models with the same ratio is considered not-verifiable.

Figure 22 shows the results of the verification runs using NuSMV. The non-verifiable
cases was related either to memory exhaustion or to several hours of execution without any answer (a threshold of 36 hours has been chosen)[2].

[2] The verification were run on a pc with 4.0 GB of ram and a 2.4 GHz core.

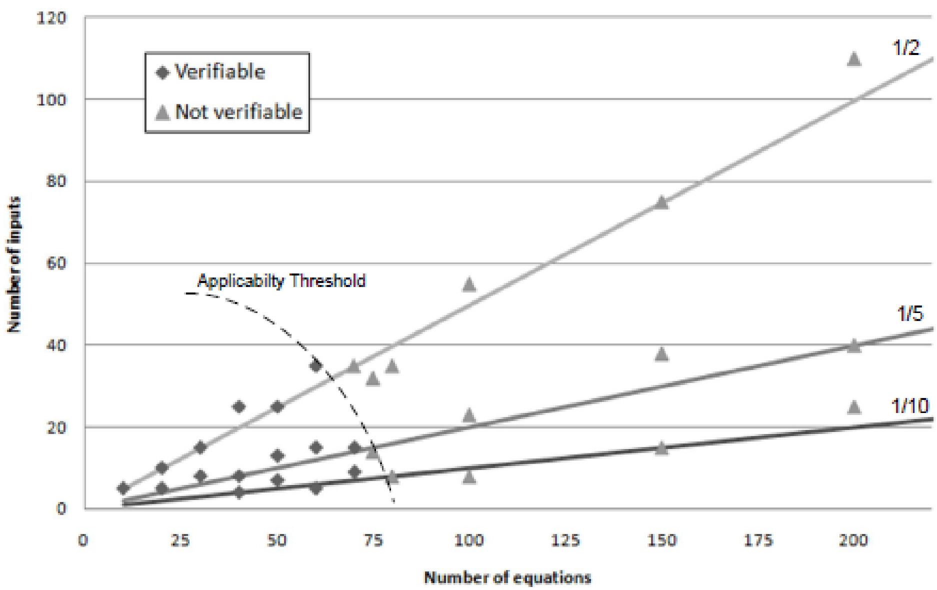


Figure 22 – NuSMV Results

The upper bound of applicability in the case of 1/10 ratio and 1/5 ratio is almost the same, approximately 70 equations. Otherwise, considering a ratio of 1/2, the number of different inputs causes the increase of the degrees of freedom and the consequent explosion of the reachable states: the computational time is considerably increased and the upper bound of applicability decreases to 60-65 equations. In the examined cases, using different optimization settings for NuSMV has not produced significant performance improvements.

Figure 23 shows the experimental results obtained by the execution of SPIN on the same dataset used in the experiments with NuSMV.

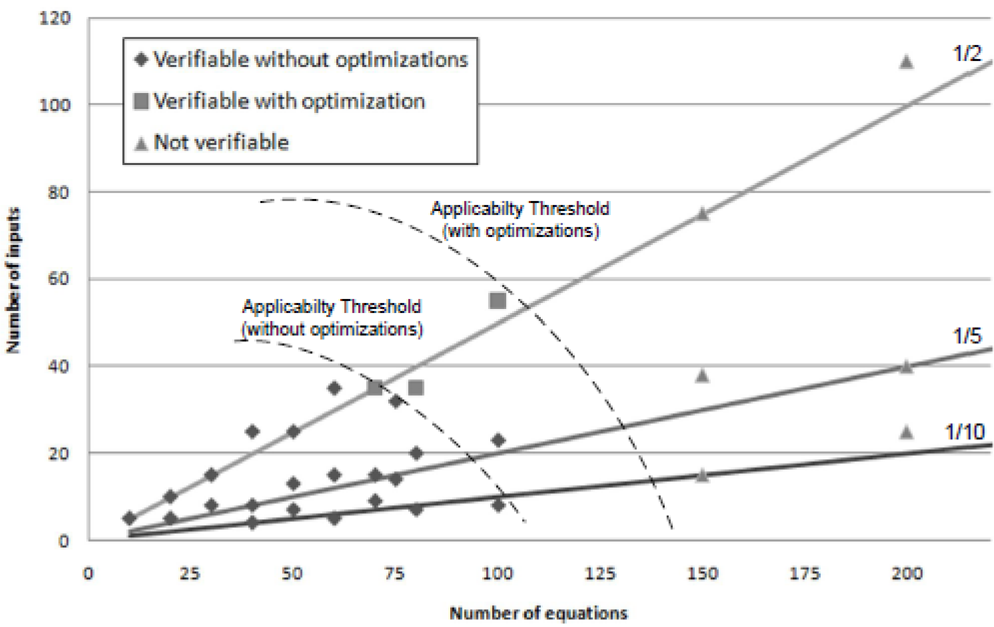


Figure 23 – SPIN Results

We observe that increasing the ratio between inputs and expressions causes SPIN not to conclude the verification in a fair time or, for higher values, to crash due to the massive usage of system memory. This behaviour can be tracked to the fact that input variables are actually state variables: so increasing inputs causes a state-space explosion.

A similar analysis can be performed for the equations, since every new equation brings a new state variable for the system. It was found that the upper limit for the applicability of SPIN to an interlocking problem is about 80 equations and 20 inputs without using any memory oriented optimization. SPIN offers several optimization strategies (e.g., hash-compact, bitstate hashing), and, according to

our experiments, the one that resulted in major benefits for our case study is the one called Minimized Automata, that consists in the construction of a minimal deterministic finite state automaton.

This optimization allows a significant memory usage reduction, increasing, on the other side, the time needed for the execution. The usage of such optimization increases the limit of applicability to about 100 equations and 60 inputs.

The results obtained with our approach show that the model checking applied to an entire interlocking system of medium size (normally some hundreds of equations) is already unfeasible.

We have however to note that the results are given on sets of strongly inter-dependent equations: an interlocking system where slicing techniques can be applied to separate sets of inter-dependent equations can be much larger. Clearly, slicing can be applied only if the actual topology of the tracks layout and the interlocking functionality do separate concerns about different areas of the layout, with little interactions among them.

Considering the real world interlocking of [Ref. 41] we can attempt to verify the correctness of only the smallest slice identified in the paper (4 signals, 7 track circuits and one switch), while the bigger slices might outrun the capability of the considered model checkers. Nevertheless, the entire interlocking is a large size one, and normally medium size interlocking present smaller slices, making the problem of their correctness addressable by model checking.

3.6 NuSMV Complexity Study

We decided to extract the relationship between the parameters of the models to forecast the complexity of the problems in terms of state space width [Ref. 42].
The experiments were executed on a models dataset parameterized in the equations length, equations number, input variables number and state variables number.

The models used have been defined in the following range:

- Equations number *Neq* [15,30];
- Input variables *Ivar* [5, 2*Neq];
- Equations length equal for all the control tables included in the model *Eq_l* (2,5,8,10,12,15,18,20);
- The length of the equations is related to the number of state variables that support the table: the generator produce control tables in which the input column are in relation of 4 state variables for each input variable (this is the approach closest to the real interlocking application).

The dataset generated is constituted by approximately 700 different models built with different combinations of parameters. In addition, since the random nature of the control table generator could raise in some cases to trivial model and in other to too much complex tables, for each combinations three models were generated and put under test. The results related to each combination were evaluated as the average of the results of the three different models generated.

We assumed as not verifiable combinations for which at least one of the three models verification did not end for memory expiring or time limit (we assumed a maximum time slot of 12 hours)[3].

The model checker computed the state space of the finite state machine described in the models starting from an initialization state.

The tool compute the total amount of reachable states before actually complete the execution, thanks this we have been able to perform the reachability analysis also for the not verifiable models.

We conducted the study of the number of reachable states in relation to input variables number, control tables (equations) number and control tables length. For each combination of two of these parameters were considered the changes of the reachable states considering different values for the third parameter. According the results obtained we tried to characterize the reachable states number for each single parameters.

The trend of the reachable states for the models with an input variables number that does not exceed the equations number results increasing with the parameters (Figure 24).

[3] Test executed with a CPU 2.2 GHz and 4 G of RAM Memory.

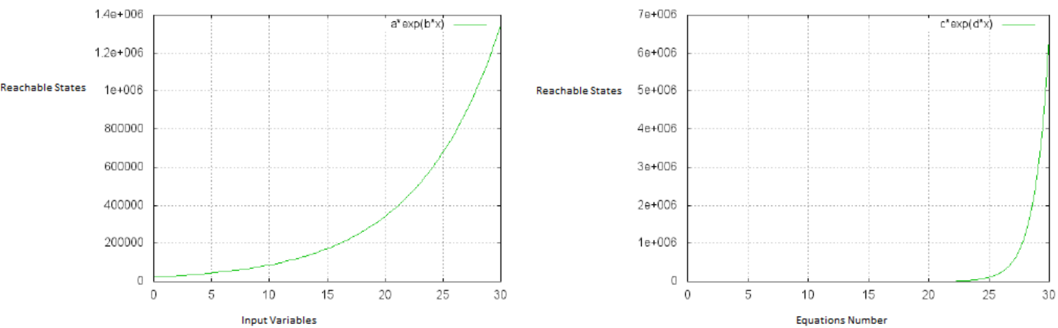


Figure 24 - Reachable states for combinations of Input Var. and Equ. Number

In both cases the trend is an exponential function even if with different coefficients:

after a slow start the number of reachable states in the case of number of equations increase faster than in the case of the input variables.

Similar results are obtained considering also the length of the equations (Figure 25).

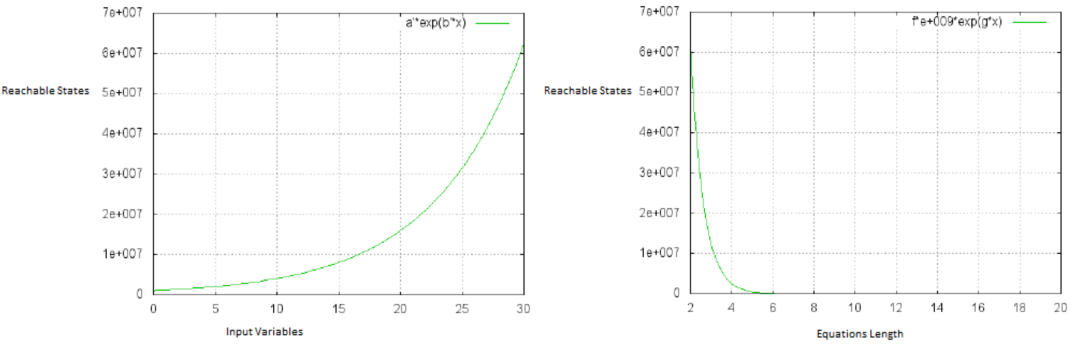


Figure 25 - Reachable states for combinations of Input Var. and Equ. Length

The reachable states decreases according an exponential trend when the equations length increases, yet in the case of a fixed number of input variables yet for a fixed number of equations (see Figure 25 and Figure 26).

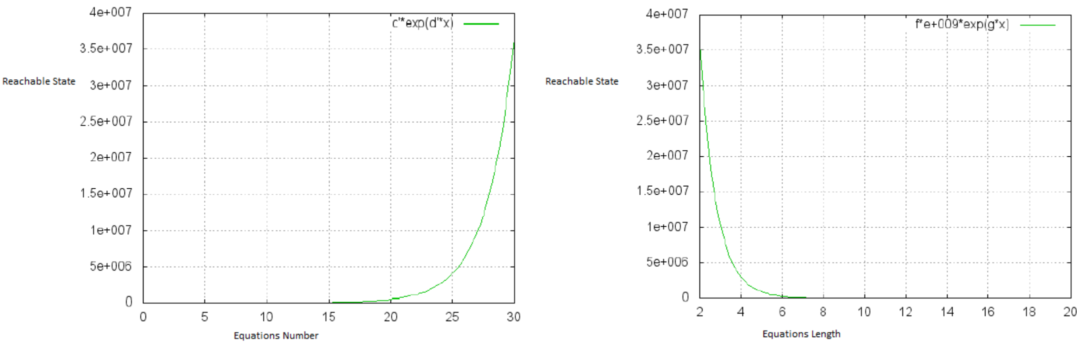


Figure 26 - Reachable states for combinations of Input Var. and Equ. Length

The decrease of the reachable states in the case of increasing equations length is related to the modellation approach adopted and control table generator tool. More the number of inputs for each table increases, more the number of state variables used as input for the tables increase: this bound specifies an high dependency between the tables and as consequence determine the reduction of the state space. However, the assumption used that produces an high dependency between the equations is close to the real interlocking applications.

Chapter 4

4 SUMMARY AND DISCUSSION

The dissertation focused to address the following:

Evaluating and Introducing new Verification and Testing Methods in the Safety Critical Domain Processes.

The dissertation faced with the verification of the design phase with the study of the applicability of the model checking techniques to the interlocking application problem and with the study, analysis and definition of the integration inside the verification process of the abstract interpretation method for source code correctness.

4.1 Static Analysis Conclusions

The definition of the Polyspace-based verification process required a considerable effort for understanding the technologies and merging them with the previously established development process. According to the results obtained on a pilot

project, the new approach has allowed a significant reduction of the verification cost in spite of the growing complexity of the code, and therefore the effort actually paid off.

The actual strength of our strategy is the abstract interpretation phase: since the code is not executed but formally analysed the approach allows to fully exploring the state space of the program, that is a prohibitive goal for traditional testing. At the same time, this technology determines the exact statement in which an error occurs. Instead, traditional testing entails an expensive report analysis to manually find the statement that has triggered the not correct output.

Our first results on the generated code part show that the new approach reduces the verification cost of 70%, even with code having a higher complexity in terms of path number. At the same time we obtain a verification accuracy that can not be achieved with traditional testing.

The productivity evaluated for the execution of the dynamic analysis through a commercial tool addressing: functional testing, MC/DC structural coverage, boundary value analysis and error guessing test cases definition is approximating is approximatin 15 executable code line (ELOC) for hour in case of an experienced resource. Considering that the use of the two-steps approach, defined during the three years research object of this dissertation, allows to cover boundary value analysis and error guessing in addition to the formal exhaustive analysis of each line of code the renewed verification process moved the scope of the two activities (dynamic analysis and static analysis). Since the productivity evaluated for the static analysis execution with the two-steps process defined is approximating 100 ELOC for hour, the renewed process allowed to increase the confidence on the correctness of the source code with the addition of a formal

method (abstract interpretation) that provide an exhaustive coverage in front of some of the testing goals, and also allowed to reduce the cost of the entire verification process.

The process defined in this dissertation is adopted and integrated in the GETS Verification Process and it has been assessed by two different Independent Safety Assessors (ISA) an interlocking subsystem currently in revenue service has passed the safety assessment by presenting evidence of the verification performed by means of the Polyspace-based abstract interpretation verification process

4.2 Model Checking Conclusions

We have studied the application of general purpose model checkers to railway interlocking systems, with the aim to define the upper bounds on the size of the problem that can be effectively handled. For this purpose, we have defined the size parameters of an interlocking systems on the basis of its control tables, and we have conducted experiments on purposely built test models of control tables with the NuSMV and SPIN model checkers. The results have confirmed that, although small scale interlocking systems can be addressed by model checking, interlocking that control medium or large railway yards can not with general purpose verification tools.

The benchmarking of the Nusmv model checker to evaluate complexity and dependency from the main interlocking parameters identified during the work allows forecasting changes to proceed with enhancements in the applicability of the model checking to the problem. According to the results obtained, more the

dataset model get close to the real interlocking applications (increasing relations between the control tables) the state space decreases as well as the computational time.

In order to increase size of tractable interlocking systems several directions will be pursued in future work, such as automated application of slicing, safe assumptions on the environment, that can tailor the input space to the one actually encountered in practice, considering the use of specialized model checkers for PLCs and the use of proper variants of SAT-based bounded model checking that are able to efficiently prove safety properties.

Conclusions

This dissertation is the results of the experience at General Electric Transportation Systems (GETS). The Company is a railway signaling manufacturer that develops embedded platforms for railway signaling systems. The safety critical nature of these applications makes the verification activities extremely crucial to ensure dependability and to prevent failures. At the end of 2007, the company decided to introduce the code generation technology within its development process and since that statement the research and the transformation of a large part of technology used until 2007 started. The integration inside the company's development process of the model based development design approach [Ref. 44], the associated automatic code generation method [Ref. 43] have been completed by the integration in the verification process of the polyspace based verification. The applicability of the static analysis with abstract interpretation to the generated code and to the handwritten code was verified and highlighted with this dissertation. The polyspace tool is used according to the guideline and the process defined during this thesis: it is currently starting the evaluation of the capabilities of the method to be applicable also for the software integration activities that became more restrictive and strongly requested by the new standard norm edition of EN 50128 (Edition 2011). The future work in terms of verification process enhancement are related to the addition at the validated and assessed static analysis process, on target testing execution activity of the dynamic testing suite: the expectations are to be able to run the tests defined on the host, directly on the

final target to increase source code correctness confidence and detect compiler issues addressing too the compiler validation critical point.

The model checking technique is continuously growing in industrial applications and also in new formalisms definition. Many of the limits for the real application of the technique is related to the computational limitations and to the critical point related to the translation of a certain problem from a domain to an other one in which the characteristics shall not be impacted in order to guarantee the correctness of the verification. In the case studied of the applicability of the model checking to the interlocking problem, the results showed that the actual computation technology limitations do not allow to get close the exhaustive verification of this kind of applications.

Bibliography

[Ref. 1] *A. Faivre, P. Benoit, Safety critical software of meteor developed with the B formal method and the vital coded processor. Proc. Of WCRR'99, 1999, pp. 84-89.*

[Ref. 2] *G. Guiho, C. Hennebert, Sacem software validation. Proc. 12th int. conf. on Software engineering, ICSE '90, IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, pp. 186-191.*

[Ref. 3] *M. Leuschel, J. Falampin, F. Fritz, D. Plagge, Automated property verification for large scale b models. FM 2009: Formal Methods, Vol. 5850 of LNCS, Springer, 2009, pp. 708-723.*

[Ref. 4] *S. Bacherini, A. Fantechi, M. Tempestini, N. Zingoni, A story about formal methods adoption by a railway signaling manufacturer. FM 2006: Formal Methods, Vol. 4085 of LNCS, Springer, 2006, pp. 179-189.*

[Ref. 5] *A. Ferrari, A. Fantechi, S. Bacherini, N. Zingoni, Modeling guidelines for code generation in the railway signaling context. Proc. 1st NFM symposium, 2009, pp. 166-170.*

[Ref. 6] *J.L. Lions: Ariane 5 flight 501 Failure, Report by the Inquiry Board. European Space Agency (1996)*

[Ref. 7] *N. Levenson, C. S. Turner: An Investigation on the Therac-25 Accidents. IEEE Computer (1993)*

[Ref. 8] *A. Deutsch, Static verification of dynamic properties, PolySpace White Paper, 2004*

[Ref. 9] *A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri: NUSMV: a new symbolic model checker. Springer Berlin (2000). Computer Aided Verification Lecture Notes in Computer Science Volume 1633, 1999, pp 495-499*

[Ref. 10] *Model Checker SPIN: http//spinroot.com*

[Ref. 11] *S. Gnesi, G. Lenzini, D. Latella, C. Abbaneo, A. Amendola, P. Marmo: An Automatic SPIN Validation of a Safety Critical Railway Control System.International Conference on Dependable System and Networks. IEEE Computer Society Press (2000)*

[Ref. 12] *Fosdick, L. D. & Osterweil, L. J. (1976). Data Flow Analysis in Software Reliability. ACM Computing Surveys, 8(3), 305-330.*

[Ref. 13] *Weiser, M. (1981). Program Slicing. In Proceedings of the 5th International Conference on Software Engineering (pp. 439-449). San Diego, CA. Washington D.C.: IEEE Computer Society.*

[Ref. 14] *Aiken, A. (1999). Introduction to Set Constraint-Based Program Analysis. Science of Computer Programming, 35(2-3), 79-111.*

[Ref. 15] *L. Hatton: Software failures: follies and fallacies. IEEE Computer (1997)*

[Ref. 16] *E.N. Adams: Optimizing preventive service of software products. IBM (1984)*

[Ref. 17] *P. Cousot, R. Cousot: Abstract Interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. Proceedings of the 4th ACM SIGACTSIGPLAN symposium on Principles of programming language (1977)*

[Ref. 18] *Cousot, P., & Cousot, R. (1979). Systematic Design of Program Analysis Frameworks. In Conference Record of the 6th Annual ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (pp. 269-282), San Antonio, Texas. New York, NY: ACM Press.*

[Ref. 19] *Cousot, P., & Cousot, R. (1992). Abstract Interpretation Frameworks. Journal of Logic and Computation, 2(4), 511-547.*

[Ref. 20] *Brat, G., & Klemm, R. (2003). Static Analysis of the Mars Exploration Rover Flight Software. In Proceedings of the 1st International Space Mission Challenges for Information Technology (pp. 321-326), Pasadena, CA.Washington D.C.: IEEE Computer Society.*

[Ref. 21] *Delmas, D., & Souyris, J. (2007). Astrée: From Research to Industry. In H. R. Nielson, & G. Filé (Eds.), LNCS 4634: Proceedings of the 14th International Static Analysis Symposium (pp. 437-451), Lyngby, Denmark. Berlin, Germany: Springer.*

[Ref. 22] *Blanchet, B, Cousot, P., Cousot, R., Feret, R., Mauborgne, R., Miné, A., Monniaux, D., & Rival, X. (2003). A Static Analyzer for Large Safety-Critical Software. In Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (pp. 196-207), San Diego, CA. New York, NY: ACM Press.*

[Ref. 23] *Tombs, D., et al.: Signalling Control Table Generation and Verification. Proceed-ings of the Conference on Railway Engineering (2002)*

[Ref. 24] *Hansen, K.M.: Formalizing Railway Interlocking Systems. Proceedings of the 2nd FMERail Workshop (1998)*

[Ref. 25] *Boralv, A.: Formal Verification of a Computerized Railway Interlocking. Formal Aspects of Computing 10 (1998) 338-360*

[Ref. 26] *Fokkink , W., Hollingshead, P.: Verification of Interlockings: from Control Tables to Ladder Logic Diagrams. 3rd FMICS Workshop (1998) 171-185.*

[Ref. 27] *Anunchai, S.V.: Verification of Railway Interlocking Tables using Coloured Petri Nets. Proceedings of the 10th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools (2009)*

[Ref. 28] *Winter, K., et al.: Tool Support for Checking Railway Interlocking Designs. Proceedings of the 10th Australian Workshop on Safety Critical Systems and Software (2006) 101-107*

[Ref. 29] *Mirabadi, A., Yazdi, M.B.: Automatic Generation and Verification of Railway Interlocking Control tables using FSM and NuSMV. Transport Problems : an Inter-national Scientific Journal 4 (2009) 103-110*

[Ref. 30] *Pavlovic, O., Ehrich, H.: Model Checking PLC Software Written in Function Block Diagram. 3rd ICST (2010) 439-448*

[Ref. 31] *Comité Européen de Normalisation en ÉLectronique et en ÉleCtrotechnique (1999). EN 50126, Railway Applications - The Specification and Demonstration of Dependability, Reliability, Availability, Maintainability and Safety (RAMS).*

[Ref. 32] *Comité Européen de Normalisation en ÉLectronique et en ÉleCtrotechnique (2001). EN 50128, Railway Applications - Communications, Signalling and Processing Systems - Software for Railway Control and Protection Systems.*

[Ref. 33] *Grasso, D., Fantechi, A., Ferrari, A., Becheri, C., & Bacherini, S. (2010). Model Based Testing and Abstract Interpretation in the Railway Signaling Context. In Proceedings of the Third International Conference on Software Testing, Verification and Validation (pp. 103-106). Paris, France.*

[Ref. 34] *Ferrari, A., Magnani, G., Grasso, D., Fantechi, A., & Tempestini, M.(2011). Adoption of Model-based Testing and Abstract Interpretation by a Railway Signalling Manufacturer. To appear on International Journal of Embedded and Real-Time Communication Systems (IJERTCS), Special Issue On Model-Based Testing for Embedded and Real-Time Communi- cation Systems (MBT4ERTCS)*

[Ref. 35] *Ferrari, A., Grasso, D., Magnani, G., Fantechi, A., & Tempestini, M. (2010). The Metrô Rio ATP Case Study. S. Kowalewski, & M. Roveri (Eds.), LNCS 6371: 15th International Workshop on Formal Methods for Industrial Critical Systems (FMICS 2010) (pp. 1-16), September 2010, Antwerp, Belgium.*

[Ref. 36] *Ferrari, A.,Fantechi A., Magnani, G., Grasso,D. Matteo Tempestini. "The Metrô Rio case study". Sci. Comput. Program. 78(7): 828-842 (2013)*

[Ref. 37] *Ferrari, A, Fantechi, A., Papini, M., Grasso, D. "An industrial application of formal model based development: the Metro Rio ATP case". SERENE 2010, London, UK, April 2010.*

[Ref. 38] *Fokkink , W., Hollingshead, P.: Verification of Interlockings: from Control Tables to Ladder Logic Diagrams. 3rd FMICS Workshop (1998) 171-185.*

[Ref. 39] *Kanso, K., et al.: Automated Verification of Signalling Principles in Railway Interlocking Systems. ENTCS 250 (2009) 19-31*

[Ref. 40] *Haxthausen, A.E.: Developing a Domain Model for Relay Circuits. International Journal of Software and Informatics (2009) 241-272*

[Ref. 41] *Winter, K., Robinson, N.J.: Modeling Large Railway Interlockings and Model Checking Small Ones. Proceedings of the 26th Australasian Computer Science Conference 35 (2003) 309-316*

[Ref. 42] *Baroncelli, S. Benchmarking di un model checker simbolico per sistemi di segnalamento ferroviario. Master Thesis. University of Florence (2012)*

[Ref. 43] *Ferrari,a. Adoption of Code Generation by a Railway Signalling Manufacturer. Ph.D. Thesis. University of Florence (2012)*

[Ref. 44] *Magnani, G. Formal Methods and Code Generation Techniques in the Development of Railway Signalling Systems. Ph.D. Thesis. University of Florence (to appear)*

Appendix A

Example of SMV Model

```
MODULE main
 IVAR
  in0 : {0, 1};
  in1 : {0, 1};
  in2 : {0, 1};
  in3 : {0, 1};
  in4 : {0, 1};
 VAR
  ot0 : {0, 1};
  ot1 : {0, 1};
  ot2 : {0, 1};
  ot3 : {0, 1};
  ot4 : {0, 1};
  ot5 : {0, 1};
  ot6 : {0, 1};
  ot7 : {0, 1};
  ot8 : {0, 1};
  ot9 : {0, 1};
  ot10 : {0, 1};
  ot11 : {0, 1};
```

```
 ot12 : {0, 1};
 ot13 : {0, 1};
 ot14 : {0, 1};
 ot15 : {0, 1};
 ot16 : {0, 1};
 ot17 : {0, 1};
 ot18 : {0, 1};
 ot19 : {0, 1};
ASSIGN
 init(ot0) := 1;
 init(ot1) := 1;
 init(ot2) := 1;
 init(ot3) := 1;
 init(ot4) := 1;
 init(ot5) := 1;
 init(ot6) := 1;
 init(ot7) := 1;
 init(ot8) := 1;
 ……….
 ………..
  init(ot17) := 1;
  init(ot18) := 1;
  init(ot19) := 1;


next(ot0) :=
```

```
 case
  ot13 = 0 & ot6 = 1 & ot5 = 1 |
  in0 = 0 & ot13 = 0 & ot17 = 1 |
  in0 = 0 & ot13 = 0 & ot17 = 1 : 1;
  TRUE : 0;
 esac;
next(ot1) :=
 case
  in1 = 0 & in2 = 1 & in3 = 0 & in4 = 1 & ot14 = 1
|
  in1 = 0 & in2 = 0 & in4 = 1 & ot14 = 1 |
  in1 = 1 & in4 = 0 & ot14 = 1 : 1;
  TRUE : 0;
 esac;
next(ot2) :=
 case
  ot9 = 1 & ot3 = 1 & ot11 = 0 & ot12 = 1 |
  ot3 = 1 & ot11 = 1 : 1;
  TRUE : 0;
 esac;
next(ot3) :=
 case
  in0 = 0 |
  in0 = 1 & in1 = 0 & in2 = 0 & in3 = 1 & in4 = 1 |
  in0 = 1 & in1 = 0 & in2 = 0 & in3 = 0 : 1;
```

```
  TRUE : 0;
 esac;
next(ot4) :=
 case
  ot13 = 0 & ot6 = 1 & ot5 = 0 |
  ot13 = 1 & ot6 = 1 : 1;
  TRUE : 0;
 esac;
next(ot5) :=
 case
  in0 = 0 & ot17 = 1 |
  ot0 = 0 : 1;
  TRUE : 0;
 esac;
next(ot6) :=
 case
  ot14 = 1 & ot9 = 1 & ot3 = 1 & ot11 = 1 |
  in1 = 1 & ot9 = 1 & ot3 = 0 |
  in1 = 0 & ot14 = 0 & ot9 = 0 & ot3 = 1 : 1;
  TRUE : 0;
 esac;
next(ot7) :=
 case
  ot12 = 1 & ot13 = 0 & ot6 = 1 |
  ot12 = 1 & ot13 = 1 & ot6 = 1 : 1;
```

```
 TRUE : 0;
esac;
  next(ot8) :=
   case
    ot5 = 1 & ot0 = 0 |
    ot5 = 1 & ot0 = 0 : 1;
    TRUE : 0;
   esac;
  next(ot9) :=
   case
    in2 = 1 & in3 = 1 & in4 = 1 |
    ot17 = 1 : 1;
    TRUE : 0;
   esac;
  next(ot10) :=
   case
    in0 = 1 & in1 = 1 & in2 = 0 & in3 = 1 |
    in1 = 0 & in3 = 1 & in4 = 0 |
    in1 = 0 & in2 = 0 & in3 = 0 & in4 = 0 : 1;
    TRUE : 0;
   esac;
  next(ot11) :=
   case
    ot14 = 0 |
    ot9 = 1 & ot3 = 0 & ot12 = 0 : 1;
```

```
  TRUE : 0;
 esac;
next(ot12) :=
 case
  in0 = 0 & ot13 = 0 |
  in0 = 1 & ot13 = 1 : 1;
  TRUE : 0;
 esac;
next(ot13) :=
 case
  in1 = 0 & in2 = 1 |
  in1 = 0 : 1;
  TRUE : 0;
 esac;
next(ot14) :=
 case
  in3 = 1 & ot6 = 0 & ot5 = 0 & ot0 = 0 |
  in3 = 1 & ot6 = 0 : 1;
  TRUE : 0;
 esac;
next(ot15) :=
 case
  in4 = 1 |
  in0 = 0 : 1;
  TRUE : 0;
```

```
 esac;
next(ot16) :=
 case
  in1 = 1 & in2 = 1 & in3 = 0 & in0 = 1 |
  in1 = 1 & in2 = 1 & in4 = 1 |
  in2 = 0 & in4 = 1 & in0 = 0 : 1;
  TRUE : 0;
 esac;
next(ot17) :=
 case
  in1 = 1 & in2 = 1 & in3 = 0 & in4 = 1 |
  in1 = 0 & in2 = 0 & in3 = 1 & in4 = 1 : 1;
  TRUE : 0;
 esac;
next(ot18) :=
 case
  in1 = 0 & in2 = 0 & in3 = 0 |
  in0 = 1 & in1 = 0 & in2 = 1 & in3 = 0 : 1;
  TRUE : 0;
 esac;
next(ot19) :=
 case
  in4 = 0 & ot17 = 1 & ot14 = 1 |
  in4 = 0 & ot17 = 1 & ot14 = 1 : 1;
  TRUE : 0;
```

```
   esac;
  SPEC AG(ot14 = 0 & ot17 = 0 & ot14 = 0 & ot3 = 0 ->
 AX(ot1 = 0 & ot19 = 0 & ot2 = 0))
```

Example of output with counter example of NuSMV

```
-- specification AG (!ot6 -> AX (!ot4 -> AX !ot2)) is
false
-- as demonstrated by the following execution
sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
ot0 = TRUE
ot1 = TRUE
ot2 = TRUE
ot3 = TRUE
ot4 = TRUE
ot5 = TRUE
ot6 = TRUE
ot7 = TRUE
ot8 = TRUE
ot9 = TRUE
-> Input: 1.2 <-
```

```
in0 = FALSE
in1 = FALSE
in2 = FALSE
in3 = TRUE
in4 = FALSE
-> State: 1.2 <-
ot0 = FALSE
ot1 = FALSE
ot2 = FALSE
ot3 = FALSE
ot4 = FALSE
ot7 = FALSE
ot8 = FALSE
ot9 = FALSE
-> Input: 1.3 <-
in0 = TRUE
in2 = TRUE
in3 = FALSE
-> State: 1.3 <-
ot1 = TRUE
ot6 = FALSE
-> Input: 1.4 <-
in0 = FALSE
in2 = FALSE
-> State: 1.4 <-
```

```
ot5 = FALSE
-> Input: 1.5 <-
in0 = TRUE
in4 = TRUE
-> State: 1.5 <-
ot2 = TRUE
elapse: 0.00 seconds, total: 1377.06 seconds
```

Example of Promela Model

```
bit in0=0;
bit in1=1;
bit in2=0;
bit in3=0;
bit in4;
bit in5;
bit in6;
bit in7;
bit in8;
bit in9;
bit in10;
bit in11;
……
…..
bit ot1;
```

```
bit tmp1
bit ot2;
bit tmp2
bit ot3;
bit tmp3
bit ot4;
bit tmp4
bit ot5;
bit tmp5
active proctype A(){
if
:: (( in0==0 && in2==0 && in3==0 ) ||( in1==0 &&
in2==1 && in3==1 )) -> tmp0=1;
::else tmp0=0;
fi;
if
:: (( in4==0 && in0==0 && ot0==1 ) ||( in5==0 &&
ot0==1 )) -> tmp1=1;
::else tmp1=0;
fi;
if
:: (( in7==1 && in4==0 && ot1==1 ) ||( in7==1 &&
in8==0 && ot1==1 )) -> tmp2=1;
::else tmp2=0;
fi;
```

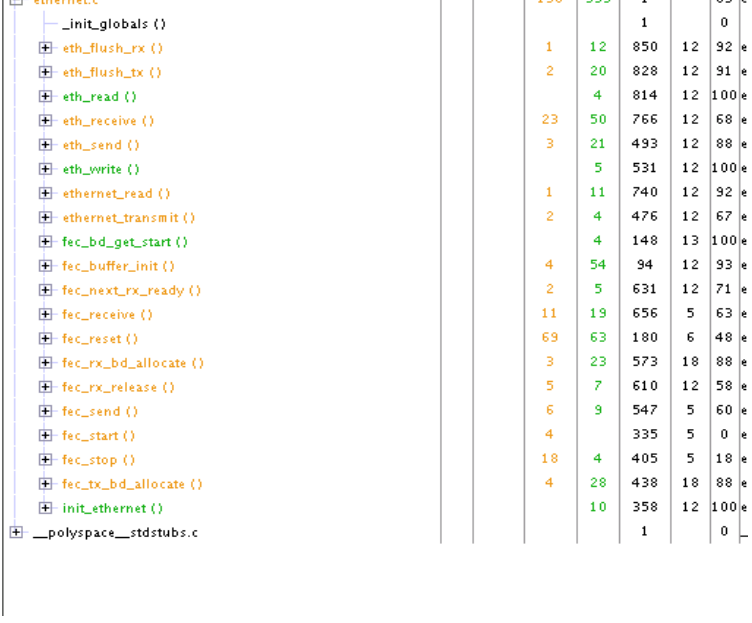
```
if
:: (( in11==0 && in7==1 && ot2==1 ) ||( in7==1 &&
ot2==1 )) -> tmp3=1;
::else tmp3=0;
fi;
if
:: (( in13==0 && in14==0 && ot3==0 ) ||( ot3==1 )) ->
tmp4=1;
::else tmp4=0;
fi;
if
:: (( in16==1 && in13==0 && ot4==0 ) ||( in16==0 &&
in17==1 && ot4==1 )) -> tmp5=1;
::else tmp5=0;
fi;
if
::true->in0=1;
::true->in0=0;
fi;
ot0=tmp0;
ot1=tmp1;
ot2=tmp2;
ot3=tmp3;
ot4=tmp4;
ot5=tmp5;
```

```
}
#define p ot0==0
#define q ot1==1
never {
T0_init:
if
:: ((p)) -> goto accept_S0
:: (1) -> goto T0_init
fi;
accept_S0:
if
:: (! ((q))) -> goto accept_all
fi;
accept_all:
skip
}
```

Appendix B

Example of Polyspace Results

In the example the source code file Ethernet.c is constituted by all the functions listed in the figure below. For each function are reported the number of green, orange, red and grey checks.



ethernet.c			158	353	1		69	e
_init_globals ()					1		0	
eth_flush_rx ()			1	12	850	12	92	e
eth_flush_tx ()			2	20	828	12	91	e
eth_read ()				4	814	12	100	e
eth_receive ()			23	50	766	12	68	e
eth_send ()			3	21	493	12	88	e
eth_write ()				5	531	12	100	e
ethernet_read ()			1	11	740	12	92	e
ethernet_transmit ()			2	4	476	12	67	e
fec_bd_get_start ()				4	148	13	100	e
fec_buffer_init ()			4	54	94	12	93	e
fec_next_rx_ready ()			2	5	631	12	71	e
fec_receive ()			11	19	656	5	63	e
fec_reset ()			69	63	180	6	48	e
fec_rx_bd_allocate ()			3	23	573	18	88	e
fec_rx_release ()			5	7	610	12	58	e
fec_send ()			6	9	547	5	60	e
fec_start ()			4		335	5	0	e
fec_stop ()			18	4	405	5	18	e
fec_tx_bd_allocate ()			4	28	438	18	88	e
init_ethernet ()				10	358	12	100	e
__polyspace__stdstubs.c					1		0	_

The figure below is an example of “coloured” source code.