Chapter 4

Software Robustness with Regards to Dysfunctional Values from Static Analysis

4.1. Introduction

This chapter describes how to demonstrate software robustness with regards to dysfunctional values. For this we use a static analysis tool based on abstract interpretation.

Our approach is original in at least two ways:

– it introduces the use of a formal method during the software development phase instead of the specification phase. ~~The standard~~ [CEN 01a] recommends the use of formal techniques (A.2 point 1 on page 48 and A.4 point 1 on page 50) such as B, Z during design and specification phases, and static analysis during software verification phase. We propose to implement static analysis during the development of the program;

Comment [RE1]: Please specify which standard you are referring to here

– it implements static analysis in ~~a delayed~~ way. Static analysis is generally used to detect targeted errors: runtime errors, memory errors or numerical errors. We propose to use a static analysis tool to verify the consistency between the specified functional domains and the source code of software, but also to calculate the value domains of unspecified inputs.

In section 4.2, we position our approach with regards to standards associated with critical systems. In section 4.3, we ~~elaborate on~~ the software robustness proof method described in section 4.4. Section 4.5 explains how to use static analysis to

Comment [RE2]: Shouldn't these two sections be swapped over so that the method is described first? This makes more sense

---

Chapter written by Christèle FAURE, Jean-Louis BOULANGER and Samy AÏT KACI.

automate one part of our method ~~to calculate~~ the "required control". In section 4.6, we present the application of the robustness verification method to Thales ~~Engagement product~~ PING. We give further perspectives on the method in section 4.7 and conclude the chapter in section 4.8.

4.2. Normative context

~~In critical systems~~ (~~transport~~, ~~air~~, railway, nuclear ~~power stations~~), ~~failures~~ can put the life of one or more people in danger ~~and therefore lead to the system being unsafe~~. For this class of systems, standards require ~~programs to demonstrate~~ the absence of failures. Their design is therefore subject to meeting very strict technical frames of reference (standards, trade documents, state of the art).

Electric/electronic systems have been used to execute functions linked to safety in most industrial sectors. The CEI/IEC 61508 standard [IEC 98] presents a generic approach to all activities linked to the safety lifecycle of electric/electronic/ programmable electronic (E/E/PES) that are used to carry out safety functions.

In most cases, safety is obtained by the addition of several systems based on various technologies (mechanical, hydraulic, pneumatic, electric, electronic and programmable electronic). The safety strategy must take into account all elements contributing to safety. The CEI/IEC 61508 standard [IEC 98] therefore provides a security analysis scheme to be applied to security systems based on other technologies (mechanical, hydraulic, etc.) and is specialized to E/E/PES systems.

Due to the large variety of E/E/PES applications and the very different degrees of complexity, the exact nature of safety measures to be implemented is application specific; this is why in the CEI/IEC 61508 standard [IEC 98] there is no general rule but there are recommendations concerning the methods of analysis to be implemented.

Standards provide scales that enable the allocation of a criticality level to each system. In complex systems based on electronic and/or programmed components, the CEI/IEC 61508 standard [IEC 98] defines the notion of s*afety integrity level* (SIL). SIL enables us to quantify the safety level to be achieved and has five values:

– 0 (no danger, material destruction);

– 1 (slight injury);

– 2 (severe injury);

– 3 (death of a person); or

– 4 (death of several people).

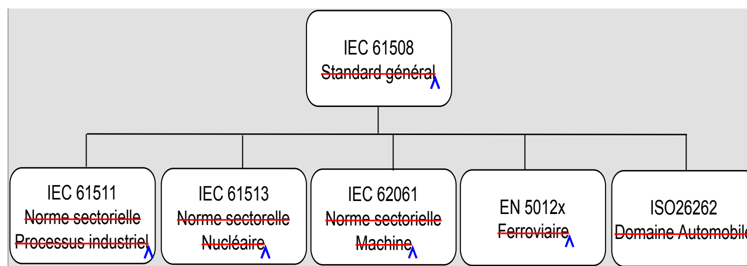
Figure 4.1 shows that the railway standard CENELEC EN 5012x is a declination of the generic CEI/IEC 61508 standard [IEC 98] that takes into account specificities of the railway domain as well as successful experiences (Sacem, TVM, SAET-Meteor, etc.).

The railway domain is therefore mainly dominated by three standards derived from CEI/IEC 61508 [IEC 98], which cover different aspects of system security:

– the CENELEC EN 50126 standard [CEN 00] describes methods to be implemented during specification and reliability, availability, maintainability and safety demonstrations;

– the CENELEC EN 50128 standard [CEN 01a] describes the actions to be taken in order to demonstrate software safety;

– the CENELEC EN 50129 standard [CEN 03] describes the structure of the safety files.



Comment [iste3]: Please supply English translation for fig annotation

Figure 4.1. *The IEC 61508 general standard and declinations*[1]

Comment [RE4]: Is this standard now available?

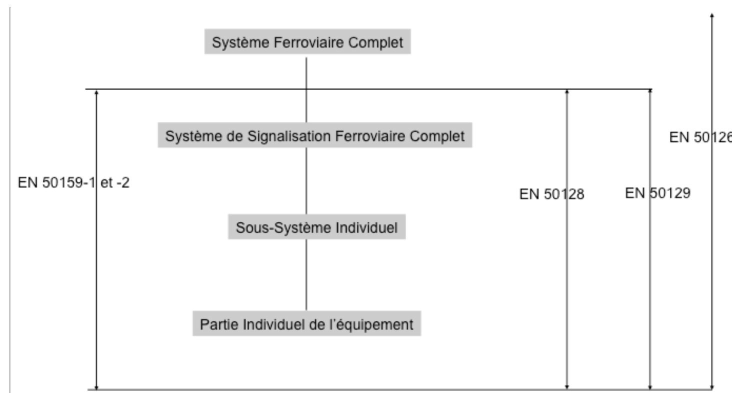
Figure 4.2 presents the architectural levels covered by each railway standard: system and subsystem.

Software development depends on a specific criticality level called s*oftware* SIL (SSIL), which varies from level 0 (no danger, no impact) to level 4 (critical, causing the death of several people). The SSIL level is reached by mastering the software quality through the application of a pre-established and systematic development process. This standard proposes a classic lifecycle in V and requires the

Comment [iste5]: Please provide English translation for fig annotation

1 ISO 26262 standard [ISO 09] is not yet available, but the automobile industry is preparing for its implementation, as shown in Chapter 9 in [BOU 09]. It is also worth noting that standard CEI/IEC 61513 cannot really be linked to standard CEI/IEC 61508 if we consider the history of standards in the nuclear domain.

implementation of techniques such as: application of formal methods during specification and design, traceability of requirements, unit tests, test coverage, etc.



Comment [iste6]: Please provide English translation for fig annotation

Figure 4.2. *Standards applicable to railway systems*

If a ~~strongly~~ recommended measure or technique (HR ~~defines~~ [CEN 01a] page 46) is not applied, this choice must be explained in detail and justified. It is necessary to show that, thanks to the global process implemented and/or to the use of other techniques, this measure or technique is not necessary.

Comment [RE7]: What does HR stand for?

Comment [RE8]: Does this refer to page 46 or CEN 01a?

We must notice that for railway systems, and particularly for railway software, it is mandatory to have an independent evaluation (see CENELEC EN 50128, [CEN 01a, Chap. 14]). Software evaluation is carried out by an entity that is independent of the development and known as an *independent safety assessor* (ISA). During the independent evaluation of the software, the conformity to the standard is verified: each non-conformity is studied and the corresponding justification is either validated or rejected by the ISA.

4.3. Elaboration of the proof of the robustness method

Let us consider the common context of a SSIL 3-4 application for which certain high recommendations (HRs) have not been followed. An objective lies behind each recommendation in a standard. We have elaborated a method for reaching ~~the~~ initial ~~objectives~~ without applying a chosen subset of HRs. We demonstrate the objectives coverage by showing the relevance and completeness of the software robustness with regards to dysfunctional values.

Table 4.1 lists three ~~common~~ choices that are ~~often~~ made during software development but which lead to the violation of a subset of HRs [CEN 01a]. The first column presents the development choice, the second contains the recommendation not followed because of this choice and the third associates a reference to this recommendation. This index is used in the remainder of this chapter.

Development choice	**High recommendation (HR) not followed**	**Reference**
The chosen programming language is C, which does not offer strong typing	Use of a strong typing programming language (Table A.4, point 7, p. 50)	HR-1
The software is integrated in *big-bang* mode, i.e. all the modules (or large packages) are directly integrated with no observable intermediate values	The integration of the software modules must be a process of progressive regrouping of each of the software modules tested beforehand (section 10.4.17, p. 24)	HR-2
Unit tests and integration tests do not allow the demonstration that the software meeds the recommendations given in the standard	The supply of an account of the coverage of tests for each module, showing that statements of the source code are executed at least once (section 10.4.14-ii, p. 24)	HR-3
	The execution of the test catalog based on an analysis of the values at the limits (Table A.13, point 1, p.55)	HR-4

Table 4.1. *Correspondence between common development choices and HR recommendations [CEN 01a] that are not followed*

Not following the HR-1 recommendation may lead to weaknesses in robustness since static typing is not carried out. To make up for this absence of automatic verification, developers add (dynamic) value control in the source code. To be comparable to static typing, this value control must guarantee that all data handled remain in their functional ~~domains~~[2] throughout the executions and must be systematically integrated into the application source code. This can only be ensured by an *a posteriori* verification of the presence and correctness of control points set in

2 The functional domain is a set of values that are specified as acceptable for a variable. In C, scalar types are stored in memory spaces that are ~~octet~~ multiples, or even bit fields that enable the representation of 2^n values where n is the number of bits. These types are used for all variables, whatever their functional domains, and even the enumerated types are in fact processed as *int*. For this reason, the domain of values associated with the type of variables is much larger than their functional domain. ~~C language therefore does not allow the automatic control of functional domains.~~

the piece of software. We use static program analysis to carry out part of this verification.

In the left-hand column, Figure 4.3 presents a ~~non controlled~~ source code written in C, and in the right-hand column contains the piece of code that integrates the value control: the value of the variable *state* is controlled between its computation by *compute_state* and its use by *use_state*. The functional domain {*FREE, BUSY*} of the variable *state* is defined by its enumerated type. In the controlled piece of code, if the value of the *state* belongs to this domain, the execution continues without modification. If this is not the case, the execution is modified by the call to the ERROR function.

Comment [RE9]: Uncontrolled?

typedef enum {FREE,BUSY} EVT;	*typedef enum {FREE,BUSY} EVT;*
…	…
EVT state;	*EVT state;*
state = compute_state(…);	*state = compute_state(…);*
	if ((*state* == *FREE*) || (state == *BUSY*))
	{
…=use_state(state);	*…=use_state(state);*
	}
	else
	{
	ERROR(state);
	}

Figure 4.3. *Example of value control on the state variable*

Violation of the progressive integration of HR-2 and its effects on HR-3 and HR-4 ~~can~~ be dynamically covered with unit tests and out-of-bounds integration tests. The number of tests required for the combinations of all possible values of a set of variables is roughly the product of the cardinals of their functional domains: these tests are frequently too numerous to be carried out during the project time allocated. We use *static analysis* to propagate *functional domains* from the production of software inputs to the *value control* when it is present in the source code and to data consumption in general. If a control point verifies a domain that is different from the one calculated by propagation, it means that it is possible to produce dysfunctional values based on correct inputs or that the control is stricter than expected. If the control verifies the domain calculated by propagation, however, it means that for all possible executions the data produced belong to the expected domains. This therefore completes the limit tests, proves that the execution is inside the control present in the piece of code, and that the control is no more restrictive than expected.

Comment [RE10]: Is this interpretation correct?

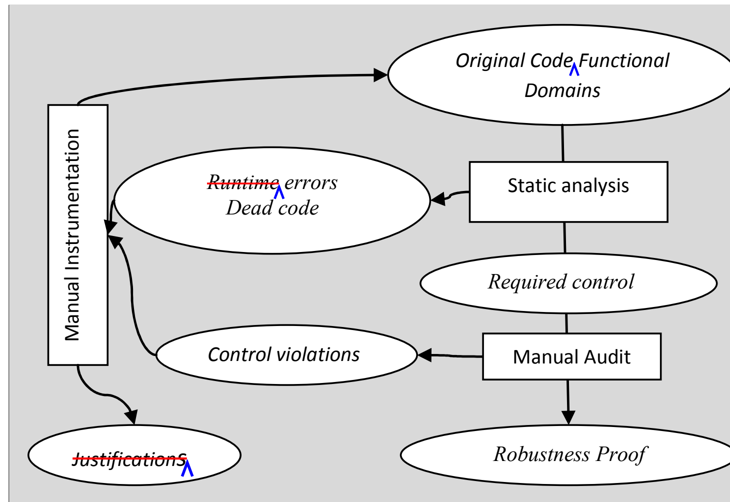


Figure 4.4. *Robustness verification method*

Comment [RE11]: Change 'JustificationS' to 'Justification(s)' and 'runtime' to 'run-time'

Finally, we consider the piece of software as being robust with regards to dysfunctional values if it implements value control and the correctness (consistency with regards to functional domains) of this control is established.

We have elaborated a specific method by which to verify software robustness implemented as the value control. This method, presented in Figure 4.4, uses static program analysis but also requires manual processing. It is made up of two main stages:

– a static analysis that calculates the required control from functional domains;

– a manual audit that shows the compliance of the value control.

It is worth noting that here static analysis does not mean direct use of a static analysis tool, but a combination of manual and automated stages.

At each stage, this method detects inconsistencies between the original source code and the specified functional domains: execution errors are detected during the first stage, and non-conformities are identified during the second stage. Table 4.2 associates the objective to be proven by our verification method, and the errors thus detected, to each recommendation: if an error is detected, the corresponding objective is not achieved and the recommendation is not covered. The conformity of the control is therefore only proven if all ~~the errors detected statically~~ and all the

non-conformities that result from the audit are instructed. In other words, that they are corrected in the source code or in the specified domains or are justified. The errors instruction stage is therefore essential in the method, but is not described in this chapter as it depends on the target application.

Reference	Objective to be proven	Errors to be instructed
HR-1	The production of data outside their functional domains is detected by a control violation	Run-time errors Control violations
HR-2	Modules do not produce erroneous data based on the input's correct values	Run-time errors
HR-3	Non-executable statements of the program are detected	Dead code
HR-4	The bounds of a module's inputs are attainable and reached by modules that consume them; and an input that takes the value outside its functional domain is detected as incorrect by the function that consumes it	Run-time errors

Table 4.2. *Errors to be instructed*

Furthermore, this method enables us to implement additional recommendations from standard [CEN 01a]:

– LR-5 *defensive programming* (Table A.3, point 1, p. 49);

– LR-6 *programming by assertion* (Table A.3, point 5, p. 49);

– LR-7 *use of static analysis* (Table A.5, point 3, p. 51);

– LR-8 *analysis of values at the limits* (Table A.19, point 1, p. 58).

4.4. General description of the method

This section informally defines the notions of *required control* necessary to ensure the robustness of the software and the *effective control*, which is effectively set in the source code. Software robustness is defined as the consistency between the required control and the effective control. This chapter presents the principal aspects of their computation on the source code, as well as the verification of robustness.

4.4.1. *Required or effective value control*

Briefly, value control [FAU 09] is ~~implanted~~ in the piece of code by control points that verify that the value of the variable at the chosen program point belongs to its functional domain.

If the control carried out is successful, then the variable is functionally correct and the execution continues. If it ~~is unsuccessful,~~ an error is ~~detected~~ and the software carries out a predefined security action.

The security action is based on the security rules that are applicable to the target software and can correspond to different behaviors: putting the software in its final state (fallback position in railway); correcting the current state and continuing the execution; or restoring ~~the initial~~ state and restarting the execution.

The action chosen is generally similar for all control points, since it is defined by the security rules ~~that can be applied~~ to the target software.

if (correct_control)	if (correct_control)
{	{
/* do nothing */	/* do nothing */
}	}
else	else
{	{
Signal_Fault(state);	Signal_Fault(state);
}	FALLBACK_POSITION;
	}

Figure 4.5. *Example of control points with different security actions*

Figure 4.5 presents two examples of security actions:

– in the left-hand column, the execution continues after having signaled the error by *Signal_Fault*;

– in the right-hand column, the software goes into fallback position after having signaled the error *Signal_Fault*.

~~Once the location strategy is known and the input to control is determined, the construction of a control point requires knowedge of:~~

– *its location*, which is determined by the constraint "verifying the value before use", which is translated in terms of source code by "before the consumption statements". As the value is only known after its production, the control point must

be located between its production and its consumption. This corresponds to a set of possible locations within the program;

– *its functional domain* ~~is known if the input is specified.~~ If this is not the case, it is calculated by hand from the specified functional domains and the piece of code that is executed.

Required control ~~is~~ the control necessary to ensure the robustness of the software. As its location is chosen as a point between production and consumption, the required control point is described by the quadruplet (*input*, *value domain*, *production locations*, *consumption locations*) where *input* is the name ~~of the variable or a memory access path~~, *value domain* is a description of the correct values for the input between each *production* and the corresponding *consumptions* ~~where~~ a location is described by a triplet (*file name*, *line number*, *column number*). It is worth noting that the production and consumption locations are potentially situated in different functions and files.

    1: 2: 3: 4: 5: 6:	*typedef enum {FREE,BUSY,UNK} EVT;* *...* *EVT state;* *state = compute_state(...);* *if (condition)* *{ ...=use_state_1(state); ...}* *else* *{ ...=use_state_2(state); ...}* *...=use_state_3 (state);*	(*state*,{*FREE,BUSY*},{1},{3,5})

Figure 4.6. *Example of a required control point*

~~In the right-hand column,~~ Figure 4.6 presents ~~the required control for the~~ original code, ~~which is presented~~ in the left-hand column: the input is *state*, ~~the~~ functional values are {*FREE BUSY*} and the locations are described by the line numbers {1, 2, 3, 4, 5} to facilitate reading. The required control (*state,{FREE,BUSY},*{1},{3, 5}) means that a control point must be established to protect statements 3 and 5 from an error in the value calculated by ~~statement~~ 1. The value control performed before ~~the~~ statements {3, 5} ensures that the value passed to statement {6} is always correct, ~~so~~ no control point is necessary between lines 5 and 6.

We define *effective control* as the control that is effectively present in the source code of an application. ~~It~~ is described by the triplet (*input*, *effective value domain*, *effective location*). The location of ~~the effective~~ control point is the result of a choice from the set of locations between each production and the corresponding

consumptions. This choice is not generally left to the developer, but is directed by a location strategy that is globally chosen for a piece of software or a whole project. The two extreme location strategies are:

– the “as early as possible” strategy, which leads to an effective control point being set just after the production statement;

– the “as late as possible” strategy, which leads to several effective control points being placed just before the consumption statements.

Figure 4.7 presents two examples of controlled code according to ~~the~~ two extreme strategies:

– “as early as possible” left-hand column (one control point); and

– “as late as possible” right-hand column (two control points).

1:
 	
2:
3:
4:
5:
6:
	state = compute_state(…);
	if ((*state* == *FREE*) ||
	(*state* == *BUSY*))
	{
	if (condition)
	{ …=use_state_1(state) ;…};
	else
	{ …=use_state_2(state);…};
	…=use_state_3(state);
	}
	else {ERROR(state) ;};
		1 :
		2 :
		
		3 :
		
		4 :
		
		5 :
		
		6 :
			state = compute_state(…);
			if (condition)
			{ if ((*state* == *FREE*) ||
			(*state* == *BUSY*))
			{ *…=use_state_1(state);…* }
			else { ERROR(state) ;};
			}
			else
			{ if ((*state* == *FREE*) ||
			(*state* == *BUSY*))
			{ *…=use_state_2(state);…* }
			else {ERROR(state);};*};*
			…=use_state_3(state);

	{(*state*, {*FREE,BUSY*}, 1)}		{(*state*, {*FREE,BUSY*}, 3), (*state*, {*FREE,BUSY*}, 5)}

Figure 4.7. *Examples of effective control points*

To verify that a piece of software is robust with regards to dysfunctional values, it is therefore necessary to calculate all the required control points, and then verify that they are all ~~implanted~~ (presence and correctness) by one or more effective control points, depending on the ~~location strategy chosen.~~

Even if control must be set during development, the complexity of ~~implantation~~, modifications in ~~the~~ software ~~and its~~ specification can lead to incoherencies in the

control of the software. It is therefore always necessary to verify the correctness of effective control ~~*a posteriori*~~ with regards to the required control.

Figure 4.8 presents an erroneous ~~implantation~~ of the required control point presented in Figure 4.6. The effective control point is erroneous for two reasons:

– the first consumption (line 3) is protected in too restrictive a way, since ~~not all of the correct values of~~ *state* ~~are accepted~~; and

– the second consumption (line 5) is not protected from incorrect values of *state*.

```
1:  state = compute_state(...);
2:  if (condition)
       { if (state == FREE)
3:        { ...=use_state_1(state); ... }
          else { ERROR(state) ;};
       }
4:  else
5:     { ...=use_state_2(state) ; ... };
6:  ...=use_state_3(state);
```

Figure 4.8. *Example of an erroneous effective control point*

4.4.2. *Computation of the required control*

All the required control points can be manually calculated from the source code of the application and the ~~known~~ functional domains, thanks to the three following steps:

– identification of software and function inputs;

–location of production and consumption; and

– computation of functional value domains.

4.4.2.1. *Identification of software and function inputs*

The identification of all the software and function inputs is done by an analysis for each function. The inputs of the software are variables associated with the ~~calls to IO 5Input/Output) functions (getc, fget, etc.)~~. The inputs of a function are parameters, static variables ~~and the outputs are produced by called functions~~; ~~and global variables are~~ consumed directly or indirectly by the function. In some cases, inputs are not implanted in the form of variables, but are components of variables (structure fields, array components). Generally, an input is a memory zone described

Comment [RE12]: Is this interpretation correct? Doesn't read well

by a path built from the name of a variable and accessors (to array components, to structure fields) defined by the language.

4.4.2.2. *Location of production and consumption*

The location of the production and consumption of an input requires a complex interprocedural analysis to follow variables (paths) that are renamed via function calls. The production of global or local variables is realized by ~~the call~~ to functions that access the environment (getc) or any other function of the application. The consumption of a value corresponds to an explicit computation based on that value: we consider that ~~parameter passing or value storing~~ in another variable are not true consumption.

If the input is a scalar object, the production/consumption is atomic. However, if the input is composite, as in a structure or an array, the production/consumption is partial and multiple: the production/consumption points must be collected for each ~~input~~ component.

4.4.2.3. *Computation of functional value domains*

The functional value domains cannot be calculated based on the source code alone. ~~If functional domains are known for all their inputs, no computation is necessary. In general, however, functional domains are only known for the software inputs as they are part of its specification: the values of these inputs are produced by the environment and generally have a restricted value domain.~~

The domains of function inputs are often unknown, particularly in the case of *big-bang* integration. They must then be calculated by ~~retroengineering of~~ the source code of the program ~~from~~ expert knowledge. In particular, these domains can be calculated by propagating the ~~known~~ functional domains ~~through~~ all the statements of the software.

4.4.3. *Verification of effective control*

To verify the effective control already present in ~~the~~ target code, it is necessary to calculate the required control points, as previously described, then examine the piece of software to verify whether each required point is ~~implanted~~ in the source code. An effective control point (*var*, *dom*, *loc*) ~~implants~~ a required control point (*var**, *dom**, *prod**, *conso**) if it applies to the same variable *var*==*var** with the same functional domain *dom*==*dom** and its location satisfies the location strategy $loc \in$ strategy(*prod**, *conso**).

The general algorithm for the verification of control points is as follows. For each required control point (*var*, *dom∗*, *prod∗*, *conso∗*), we look for effective control points that control the value of the variable *var*:

– if no effective point exists, then a non-conformity is added:

(*var, dom∗, prod∗, conso∗*) ? *None*

– for each effective control point (*var*, *dom*, *loc*), we verify that the location *loc* is correct with regards to the location strategy and for locations of production *prod* and consumption *conso*:

- if the location is incorrect, then the following non-conformity is added:

(var, dom∗, prod∗, conso∗) ? (var, dom, loc) | NC(prod, conso, loc);

- if the location is correct, then the domains are compared,

- if *dom =/= dom∗*, then the following non-conformity is added:

(*var*, *dom∗*, *prod∗*, *conso∗*) ? (*var*, *dom*, *loc*) | NC(*prod*, *conso*, *loc*, *dom*),

- if *dom == dom∗*, then the following partial conformity is added:

(*var*, *dom∗*, *prod∗*, *conso∗*) ?(*var*, *dom*, *loc*) | PC(*prod*, *conso*);

– then the conformities are analysed to evaluate their coverage:

- if all the production and consumption points are associated with an effective control point, then a total conformity is added:

(*var*, *dom∗*, *prod∗*, *conso∗*)?TC,

- if not, certain production and consumption points are not associated with effective points and we add a non-conformity:

(*var*, *dom∗*, *prod∗*, *conso∗*)?PC.

Once all of the required points have been studied, the results are:

– the required points that are not implemented in the source code; and

– potential non-conformities.

Finally, these potential non-conformities are instructed and can lead to a ~~modification~~ of the source code or specified functional domains, or to a justification.

4.5. Computation of the control required

The control required for software is long and difficult to compute as it demands interprocedural analyses on complex data, such as value domains. The tools based on static analysis (by abstract interpretation) automatically carry out such analyses, but do not directly verify control points, calculate the required control points or collect the effective control points. We have developed a method to calculate the required control that uses the abilities of static analysis tools as they are.

Static analysis tools simulate all executions of the target application in a symbolic execution and verify dynamic properties. This symbolic execution requires the abstraction of concrete values into abstract values (lattice elements), and the computation of fixed points to cover the recursion often present in programs (loop, recursive function). These tools compute abstract values of each variable at each program point. The abstract value of a variable, *v*, represents all the values *v* can take during all possible executions of the program. From these abstract values, tools verify properties such as the absence of run time, numerical or memory errors.

Our method requires the following functionalities from the static analysis tool:

– detection of run-time errors;

– computation of a subset of dead code;

– *observe_value* operator enabling the extraction of the value of a variable;

– *assume_value* operator whose semantic is assert and then assume (see section 4.5.2);

– computation of the software call graph;

– computation of the data dictionary.

Our algorithm for computing required control points from the source code and functional domains operates in two main stages, as presented in Figure 4.9:

– stage 1 identifies inputs and localizes their production and consumption points; and

– stage 2 calculates the value domains at the production points.

These two stages apply static analysis to meet different objectives.

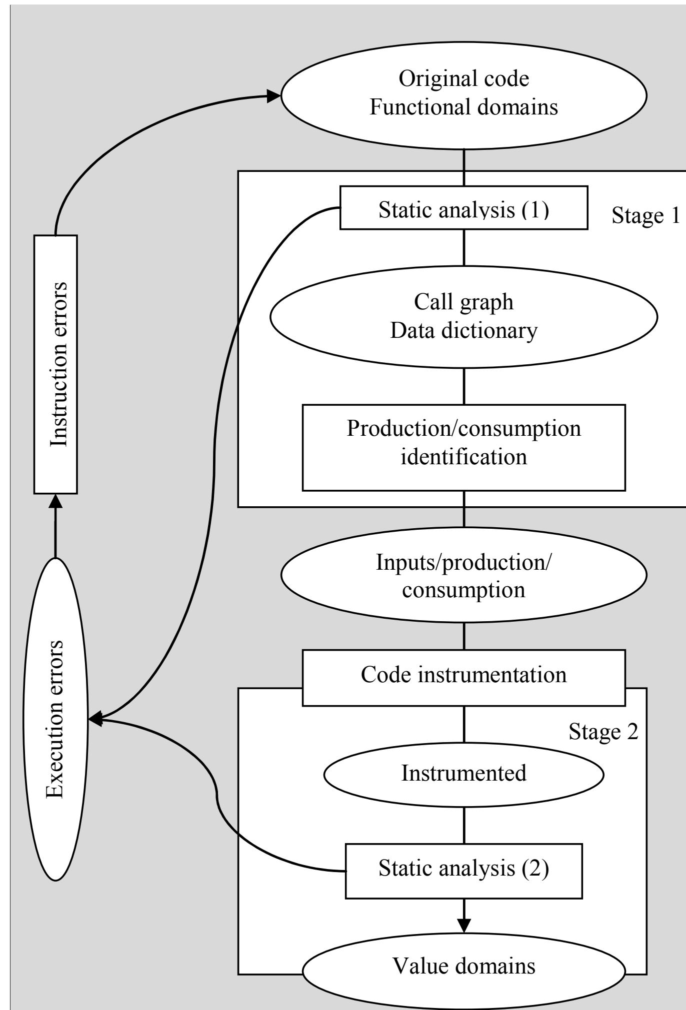


Figure 4.9. *Computation of the required control*

Comment [RE13]: Figure needs to be altered via paste special

During the first stage, which is explained in section 4.5.1, the static analysis tool is used to calculate the call graph and data dictionary of the original code. During this computation, the tool also verifies the absence of compilation and run-time errors. These errors must be instructed before ~~being~~ the other results can be used.

This is represented in Figure 4.9 by an arrow going from the run-time errors to the original code. Once the errors have been corrected, the static analysis tool produces the data dictionary and the call graph of the application.

During the second stage, explained in section 4.5.2, static analysis is used to compute unknown functional domains and thus complete the definition of the required control points. The errors detected during this stage must also be instructed before using th~~ese~~ domains.

4.5.1. *Identification of production/consumption of inputs*

Stage 1 begins with the static analysis of the original code, which aims to calculate the call graph and data dictionary for global and static variables. This information is used to calculate inputs, locations of production and consumption as described below:

– *Software inputs*: the calls of the input~~/output~~ functions of the language (getc, scanf, fscanf, etc.) are found in the call graph. These call locations are also the location of the production of input values for the software. By examining the source code at these locations, the assigned variables (software input variables) are determined. We thus obtain the software inputs and their production places. Then, the consumption locations are found throughout the chains of function calls.

– *Function global or static inputs~~/outputs~~*: the data dictionary contains the list of the application's global or static variables. If it also contains their production and consumption places – either direct or indirect – there is nothing left to calculate. If not, it is necessary to find direct uses in the source code and then follow the call graph to find all the indirect access points.

– *Function parameters*: if the data dictionary contains function parameters, and their direct or indirect input or output, there is nothing to calculate. If not, the list of the parameters ~~of a function~~ is obtained by looking at their definition. To calculate the inputs~~/outputs~~, it is necessary to determine all the~~, places the parameters can be accessed~~. We consider that the production place of a parameter is situated before the execution of the first statement of the function. The consumption places are calculated by following their value throughout calls to function (and therefore potential renaming). This computation is done entirely by hand if the static analysis tool does not give information regarding the function parameters.

If a new kind of input is dealt with, it is necessary to define how to ~~calculate inputs of this kind~~ and their production/consumption. The rest of the method is applied without change.

4.5.2. *Computation of value domains*

Stage 2 of the method described in Figure 4.9 aims to calculate the value domains ~~that are compatible with the domains specified for the software inputs~~. ~~It also~~ verifies the consistency between the specified ~~functional input~~ domains and the source code of the target application.

~~It~~ begins by a manual phase to instrument the source code, followed by a static analysis of the instrumented program, which calculates the unspecified functional domains. The instrumentation phase places constraints on the inputs that have a specified domain, and observation points for those that do not. The specified domains are translated into constraints by using the *assume_value* operator so that any violation of the constraint leads to halting of the execution (a common semantic of the *assert* operator), otherwise the execution continues by taking the constraint as an hypothesis (assume semantic). These constraints are placed as early as possible after the production of value. The observation points are translated thanks to the *observe_value* operator placed in the source code at production locations, i.e. at the beginning of procedures for parameters and global (static) variables. The location strategy "as early as possible", used for setting constraints and observation points, minimizes the instrumentation volume and only requires knowledge of the production places.

In the second column in Figure 4.10 we can see an example of source code and in the third column its instrumented version. The observation of the variable *in* is posed before statement *1* and the constraint on the variable *line* is set after statement *1*.

Comment [RE14]: Obtained?

1:	*gets(line);*	observe_value(in);
		gets(line);
		assume_value(line);

	…	…
2:	*state = compute_state(line,in);*	*state = compute_state(line,in);*
3:	*if (condition)*	*if (condition)*
4:	*{x=use_state_1(state);…}*	*{ x=use_state_1(state); …}*
5:	*else*	*else*
6:	*{ x=use_state_2(state);…};*	*{ x=use_state_2(state) ; …};*
7:	*y=use_state_3(state);*	*y=use_state_3(state);*

Figure 4.10. *Computation of the required control point*

Static analysis of the instrumented code propagates specified value domains for the inputs towards observation points. The result of the analysis contains the list of domains calculated for unspecified inputs. The calculated domains approximate all

of the values attained from specified domains by symbolically executing the source code and by collecting the constraints ~~linked~~ to the correct executing of this piece of code (division by zero, for instance). This approximation is due to the abstraction used by static analysis to take into account all possible executions (sound). For example, if the domain [3..4] ∪ [5..7] is calculated, it ~~is~~ over approximated in [3..7] by using the interval lattice.

Comment [RE15]: Obtained?

Furthermore, static analysis automatically verifies the consistency between the domains of the software inputs and the instrumented code. Run-time errors, dead code or violations of the control signal incoherencies between the source code and instrumented constraints. These errors are instructed to be corrected or justified. ~~If they need to be corrected, the correctness~~ can require the modification of the source code or the specified functional domains. It is sometimes difficult ~~to return to the error found at the violated constraint~~ if the influence of the latter is not immediate. It is only ~~once~~ all the~~se~~ errors have been instructed, as shown in Figure 4.9, that the calculated functional domains are correct and therefore exploitable.

4.6. Verification of the effective control of an industrial application

The computation method for the required control described in section 4.5 has been applied to verify the value control of a real application, the development of which did not follow recommendations HR-1, 2, 3 and 4, but implants the value control as previously defined.

Comment [RE16]: Please give the relevant section number here

4.6.1. *Target software*

The software for Thales ~~signaling~~ product~~s~~ (PING SSIL3-4) implements a rail root-management system. This product contributes to the safety of the root-management functions of railway signaling posts, to the control and command of elements of the track (switches, signals, etc.) and to the control–command exchanges with external systems, as Figure 4.11 illustrates.

This software integrates the control of the values of its inputs. ~~We call function inputs parameters, static local variables, global variables consumed directly, but also produced (parameters, statics or glabals) by the called functions.~~ The location strategy chosen for this ~~kind~~ of input implies that ~~consumption is protected by a~~ control point placed in the function that consumes the value “at the latest” before consumption. ~~We call the variables (parameter, local or global) assigned by the call of an input/output function software inputs. These inputs~~ need to be protected by a control point placed in the function, which produces the value “as early as possible” after production.

Comment [iste17]: Please provide English translation for fig annotation

Comment [RE18]: This doesn’t make sense. Please rephrase this sentence

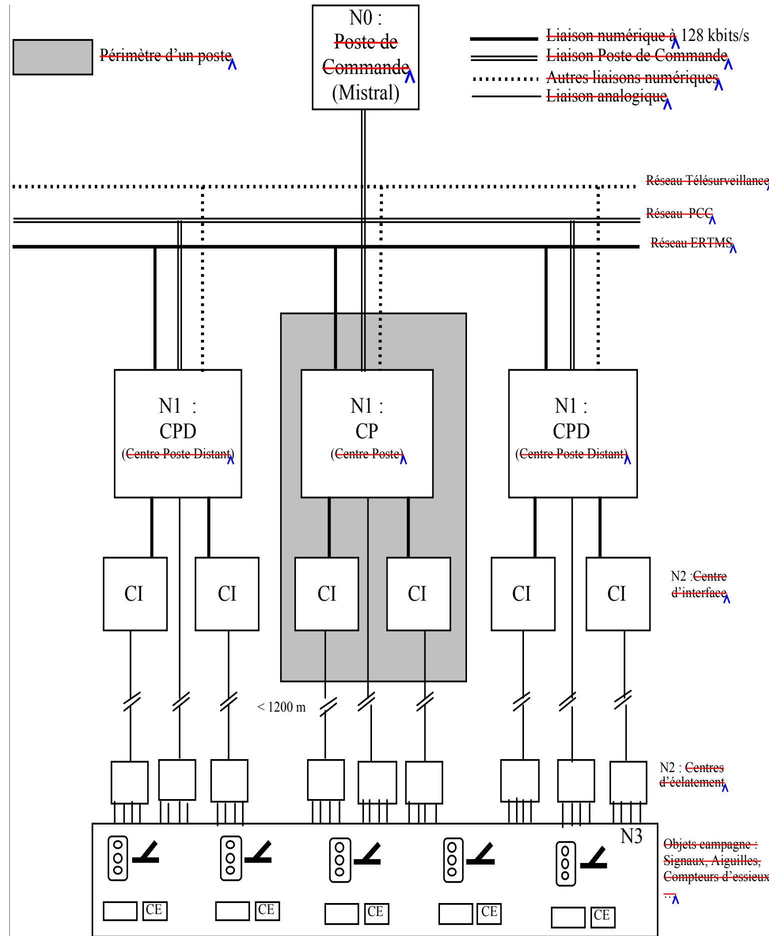


Figure 4.11. *Architecture of the Thales engagement product*

Comment [iste19]: Please provide English translation for fig annotation

The general form of the value control is given in Figure 4.12. The call to the *Fatal_fault* function logs the errors detected by the value control: the logged message is constructed from the name of the *module* and the *fault_message* linked to the control point that has been violated. The call to *FALLBACK_POSITION* commands the restrictive behavior of the system.

```
if (Nbr_variables_safety >= MAX_VARIABLES_SAFETY)
  {
    Fatal_fault (module, fault_message);
    FALLBACK_POSITION
  }
  /* else Nominal processing */
```

Figure 4.12. *Example of a control point present in Thales ~~engagement~~ product ~~software~~*

~~We can note that the~~ *Fatal_fault* function is not limited to the control of values but is used to process ~~cases of error~~ in general. Furthermore, the macro *ASSERTION* that defines a control point is not only used to pose the effective control points. The presence of all these patterns leads to a lack of uniformity in the source code ~~produced~~. Therefore the search for effective control points, which ~~in theory~~ could be done automatically by ~~searching for patterns~~ in the source code ~~(*pattern matching*)~~, must be carried out manually.

The functional domains are specified in a table that associates the full variable name or memory access path (together with the module and function names) and the value domain. In our case, domains are only given for what is called software inputs above and is described as set of values *{FREE, BUSY}*, intervals [3..12], or value properties *non_null(p)*, *sizeof(s)==4*.

4.6.2. *Implementation*

We have chosen Polyspace®[3] as the static analysis tool because it is the tool used by the industrialist who developed the source code and because it offers all the functionalities we are ~~searching~~ for.

Comment [RE20]: Revise chapter number

Polyspace® is a static analysis tool based on abstract interpretation, the aim of which is to detect run-time errors as well as non-deterministic behaviors in the source code of ~~an~~ application written in C, C++ or ADA. A run-time error is a program ~~error~~ state ~~that is~~ perfectly identified in the standard of the target language as ~~leading to an~~ *unspecified*, *undefined or implementation-defined behavior*. The tool calculates the target program points that could cause one of these behaviors and gives them an error status: impossible (green), potential (orange), certain (red) or unattainable (grey) code. Polyspace® produces the association list of these program points and their error status. Furthermore, it calculates a subset of dead code: ~~the~~ statements never executed and ~~the~~ procedures never called.

---

3 ©~~MathWorks~~, see Chapter 3 of this book.

Polyspace® offers the two operators that are essential to the method:

– Observation points (*observe_value*): this is implemented ~~into~~ an ~~inspection point~~ (IPT) on the variables at the chosen points. The command *#pragma Inspection_Point var1 var2* asks the tool to ~~produce~~ the possible values (abstract values) of variables *var1 var2* at the point in the program where it is set. The calculated value domains are added to the analysis results. It is worth noting that the inspection points ~~can set only be~~ on scalar-type variables, which restricts the possibilities of observation: in particular, the structures and arrays cannot be globally observed but ~~can~~ be observed component by component.

Comment [RE21]: Inserted into?

Comment [RE22]: What does this stand for?

– Constraints (*assume_value*): each constraint is translated into assertions. The semantic of the statement *assert(test)* for Polyspace® is: if the test is verified at the point where it is set, the rest of the execution is restrained to the values for which the test is true, otherwise ~~it stops the execution.~~ The status of the assertions (definitely violated, never violated or potentially violated) is also present in the results but does not directly prove the control: in particular, the fact that the assertion is never violated (green) proves that the control domain is included in the domain calculated by Polyspace® but does not ~~show~~ that the two domains are equal. The manual audit described in section 4.4.3 ~~therefore~~ remains essential ~~for~~ the verification objective ~~to be met~~.

Polyspace® makes the intermediate results necessary to ~~the application of~~ our method available to the user:

– unexecuted functions (dead code);

– functions call graph; and

– data dictionary.

4.6.2.1. *Preliminary analysis of the application*

The first stage of the static analysis of a source code is its "compilation" by the tool. To enable the compilation, we have taken into account the specificities of this application and configured Polyspace®, as described in Table 4.3.

Moreover, the Polyspace® compilation is stricter than ~~those~~ carried out by common compilers, since it systematically verifies the respect of the ~~C ANSI standard~~. Thus, certain modifications have been brought to the application's source code to correct the aspects that do not conform to ANSI ~~compliances~~. Once this configuration is finished, the analysis of the application by Polyspace® is possible.

Comment [RE23]: Please provide reference details for this standard in the bibliography

The results show that the interruptions are not simulated. In particular, infinite loops were wrongly detected. We have deleted these cycles, which only delay the execution of the rest of the statements without changing their behavior. ~~At this stage~~ the analysis was correct but too expensive in terms of time.

Polyspace® options	Specificity of the application
-target i386 -OS-target no-predefined-OS -I APPLICATIONS/WATCOMC_Includes	Compiler WATCOMC
-dos	The delimiter "\" is used instead of "/" in the names of files included that are processed
-discard-asm	The pieces of assembler are not processed but are automatically skipped and stubbed
-D INTERRUPT= -D __far= -D FAR=	The "interruptions" and "far pointers" are not recognized or simulated but are skipped

Table 4.3. *List of options necessary for the compilation*

We simplify the source code to enable a more efficient Polyspace® analysis. To reduce the number of pointers handled, we redefine functions without functional contribution (message logging), by associating a nohup semantic to them. We also replace the *Fatal_Fault*, *Fallback_ Position* and *Pseudo_-Fatal_Fault* functions with definite stops of the execution. We define the function *ALLOCATE_MEMORY* to the standard *malloc* functions. By doing this, we divide the number of aliases calculated by Polyspace® by a factor of 5.8. Furthermore, we redefine the macro *ASSERTION*, which implants the control points, to the call of the function *assert* recognized by Polyspace®. During this preliminary work, we are able to detect and correct three run-time errors in the source code of the application.

These adaptations are often necessary to make static analysis practicable at the source level because not all existing compiler extensions can be imbed in the analyzer and because some source code characteristics, such as the number of aliases, limit the efficiency of static tools.

4.6.2.2. Instrumentation and analysis of the instrumented code

In our method, the instrumentation of the source code has two goals: to add the specified constraints and to add the observation points. These two kinds of instrumentation are controlled independently thanks to the *Active_constraint* and *Active_observation* macros that are activated by the compilation options *–D Active_ constraint* and *–D Active_ observation.*

Constraints are implemented in the form of C functions, grouping the assertions together for the same variable. Figure 4.13 presents the translation of the functional domain *[0..NB_ TYPE_CARTE-1]* of the *typeCarte* variable according to the

specification in the *constraint_–typeCarte* function. Certain specified functional domains are not translated into constraints because they are too complicated to be propagated by Polyspace®. ~~An example of this is~~ ~~bit fields values (16-bit heavy weight [0..4], weak weight [0..255])~~ ~~IP address structures (192.168.[0..255].[0..255])~~.

```
void constraint_typeCarte(E_TYPE_CARTE typeCarte)
{
#define TMP_typeCarte typeCarte
  assert(TMP_typeCarte >= 0);
  assert(TMP_typeCarte <= NB_TYPE_CARTE - 1);
}
```

Figure 4.13. *Example of a constraint function*

The constraint functions are duplicated ~~in the case where~~ the constraints must be set at several points in the program. In general, the constraints are set once between production and consumption – just after production for the sake of simplicity. However if there is a *Cast* operation between production and consumption, the constraint is set once just after production and a second time before consumption, if it can be translated on the ~~new type~~. This reinforces the effect of the constraints, because it enables Polyspace® value propagation otherwise stopped by the casts, especially if they concern components of structured objects (array component, structure field). Figure 4.14 presents the use of the *constraint_typeCarte* function in the *GetIdCarte* function.

```
1:  | T_idCarte DIP_GetIdCarte(E_TYPE_CARTE typeCarte)
2:  | {
    |
    | #ifdef Active_observation
    |   OBS_DIP_GetIdCarte(typeCarte)
    | #endif /* Active_observation */
    |
    | #ifdef Active_contrainte
    |   contrainte_typeCarte(typeCarte);
    | #endif /* Active_contrainte */
    |
3 : |    DIP_ASSERTION(typeCarte < NB_TYPE_CARTE);
    |
5 : |    return gLesIdCarte[typeCarte];
6 : | } /* FIN DIP_GetIdCarte */
```

Figure 4.14. *Instrumentation of the GetIdCarte function*

The observation points are implemented as macros that expand in Polyspace® *Inspection_Point* for the software inputs and its functions. It is worth noting that Polyspace® only enables ~~us to observe~~ scalar inputs. This limits the possibility of observation on structured objects. Figure 4.14 presents the use of the observation macro *OBS_DIP_GetIdCarte* in the *GetIdCarte* function of the original application.

The instrumented code is then analyzed by Polyspace® using the maximal precision options (-O3 -to pass4) as well as the options required by the analysis of the original code presented in Table 4.4.

Polyspace® options	
-target i386	-D INTERRUPT=
-OS-target no-predefined-OS	-D __far=
-I APPLICATIONS/WATCOMC_Includes	-D FAR=
-dos	-O3
-discard-asm	-to pass4

Table 4.4. *Polyspace® analysis options for the instrumented software*

The analysis of the instrumented code generates run-time errors if the specified domain is not consistent with the source code. Three kinds of errors can be automatically detected: functional constraint violations; value control violations; and run-time errors. The violation of a functional constraint appears as red *assert* in the function constraints definition file. For example in Figure 4.13 if the *GetIdCarte* function is called with an erroneous value (*typeCarte ≥NB_TYPE_ CARTE or ~~negative~~*) the assertions contained in the function *constraint_ typeCarte* (see Figure 4.12) ~~will be~~ violated. The violation of an original code control point also appears as a red assert or dead code. In the example in Figure 4.13, if the control point in line 3 is poorly defined (*typeCarte > NB_TYPE_CARTE*) a violation appears. Finally, a general run-time error can appear. For example, in Figure 4.13 an out-of-bounds access line 5 can be detected if the *gLesIdCarte* variable size is declared to be too small with regards to the control implemented (for example, size(*gLesIdCarte) < typeCarte)*.

The analysis of the errors is simple for constraint errors, but a lot more complicated for other types of errors. The correction of the source code with regards to these errors is quite hard without functional expertise on the application. Once these incoherencies have been corrected in the source code or the specified functional domains, the results of the analysis ~~have been~~ used for the audit. The domains calculated at the inspection points serve as a reference domain for the unknown functional domains.

4.6.2.3. *Source code audit*

The aim of the source code audit is to verify that the effective control points implement the required points. The general algorithm for verifying the effective control points, presented in section 4.4.3, can be simplified by the instantiation of a strategy for setting control points (chosen location), as described hereafter.

In particular, the source code traversal algorithm can be specialized. Each executable function~~,~~ *f*, is covered according to a traversal algorithm adapted to the kind of input that is controlled:

– for parameters, local statics or globals variables: it is necessary to follow the execution backwards from each consumption in *f* to the beginning of the definition of *f*;

– for output parameters and global variables of a function *g* called in function *f*: it is necessary to follow the execution forward from the call to *g* until ~~reaching the end of~~ the definition of function *f*;

– for software inputs: it is necessary to follow the execution forward from the call ~~of~~ the input function that produces ~~its~~ value until the function that contains the consumption of this value is reached. It is complex research due to its inter-procedural nature.

However, the conclusion algorithm on compliance is the same:

– if no effective control point is found, it means that the production is not verified and we add it to the list of potential non-compliances;

– if at least one effective control point is found, the verification is carried out in two successive stages;

- for each effective point we verify that the controlled domain is indeed the one calculated by Polyspace®:

i) if the domains are not equal, then we add it to the list of potential non-compliances, and

ii) if the domains are equal, we move on to the next stage,

- we verify that the protection offered by these control points is ensured for all possible executions by the computation of the paths covered in the source code:

i) if the effective points protect all possible paths, we add them to the list of non-compliances;

ii) if this is not the case, we add them to the list of potential non-compliances.

The non-compliances thus obtained are instructed one-by-one. The errors are corrected in the source code or in specified functional domains, and the other errors are justified. This process is beyond the scope of this chapter.

4.6.3. *Results*

4.6.3.1. *Direct results*

We have applied the method to two successive versions of the Thales ~~rngagement~~ product ~~software~~.

The Thales root-management ~~product~~ software V45.05 implemented in 92 Klines of C is made up of 66 files and 717 functions. The functional domains of 95 inputs were specified, which enabled the development of 95 functions in a file of 2,397 lines of C code. We have placed the constraints at 196 program points and the observation on 1,509 scalar inputs, the functional domains of which are unknown. The Polyspace® tool automatically detected one inconsistency between the source code and the specified functional domains. Static analysis of the constrained code led to eight source code corrections and one correction of the functional domain in the specification.

The manual verification of the control required the examination of 2,644 effective control points and showed that 71% of points were correct, 18% were non-conforming and the status of the remaining 11% had to be established by the industrialist as it required functional knowledge about the application.

The Thales root-management ~~product~~ V49.03 ~~software~~ implemented in 122 Klines of C is made up of 87 files and 1,409 functions. The specified functional domains of 358 inputs led to the development of 358 constraint functions (containing 1,800 constraints) with 8,099 lines of C code. We placed the constraints at 225 program points, and ~~the observation~~ of 4597 scalar inputs, the functional domains of which ~~are~~ unknown.

The Polyspace® tool automatically detected around 20 inconsistencies between the source code and the specified functional domains, which led to the correction of 22 functional domains in the specification. The validation of results is ongoing and the manual audit has not yet begun.

Software verification ~~has~~ therefore not finished, but the increase in the number of constraints processed visibly improved the quality of calculated domains. More than 84% of domains were specific (different from the domain associated ~~wiht~~ the data type), whereas for the previous version only 48% of calculated domains were precise.

This enables us to say that the manual compliance audit of the control points present in the source code with regards to the control points ~~required~~ will be more rapidly performed.

4.6.3.2. *Indirect results*

Comment [iste24]: Please provide English translation for fig annotation

The application of this methodology reinforced vertical traceability (see Figure 4.16) of the V model development cycle (see Figure 4.15), in particular the data refinement from systems and subsystems towards the lowest levels of software application.

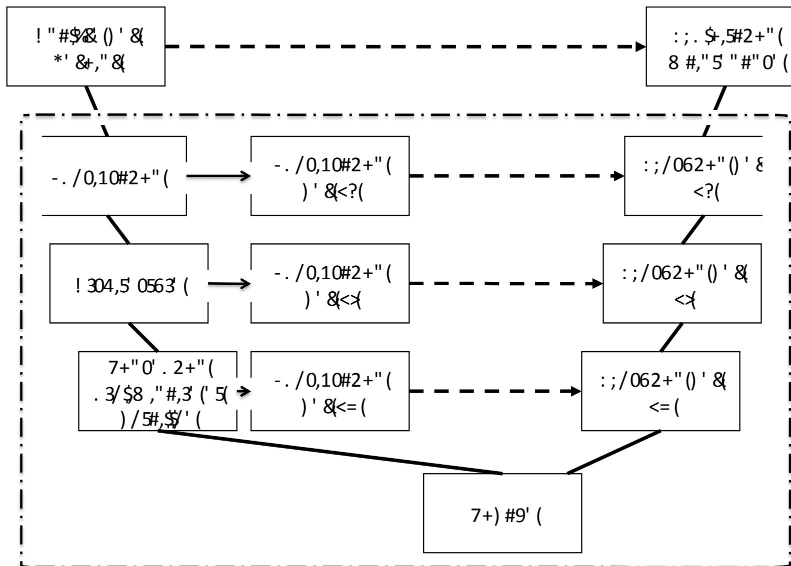
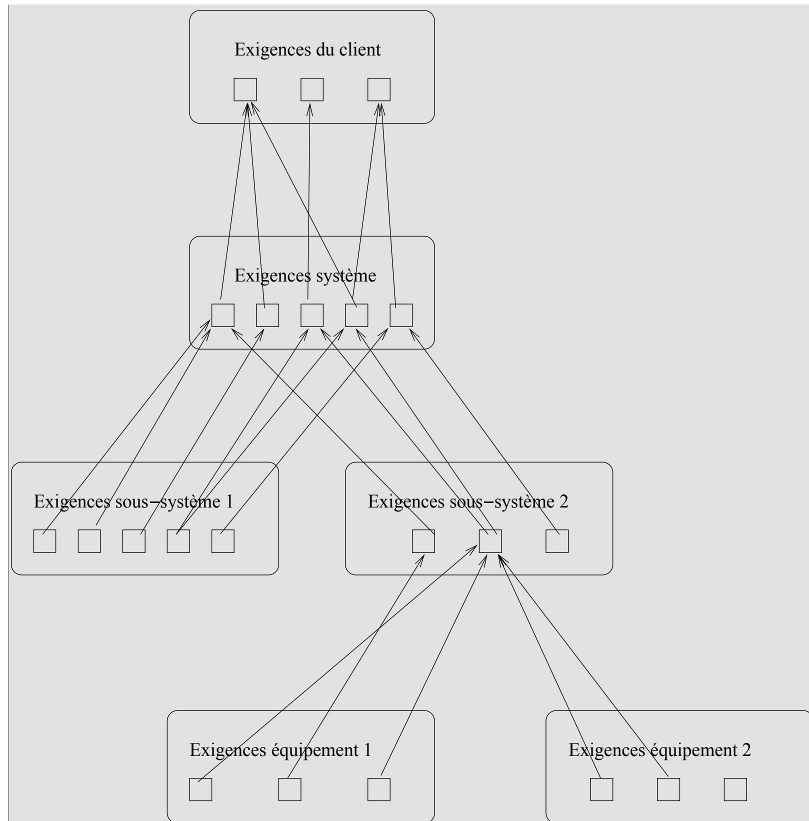


Figure 4.15. *V development cycle*

Indeed, manually verifying the consistency of the functional domains handled by the software from values provided by systems and subsystems is arduous, complex and error prone due to the multiplicity of potential production and data consumption and to data cross-dependencies that can invalidate the established domains.



Comment [iste25]: Please provide English translation for fig annotation

Figure 4.16. *Vertical traceability*

The horizontal traceability application of our method (see Figure 4.17) also statically shows the partial or complete achievement of objectives:

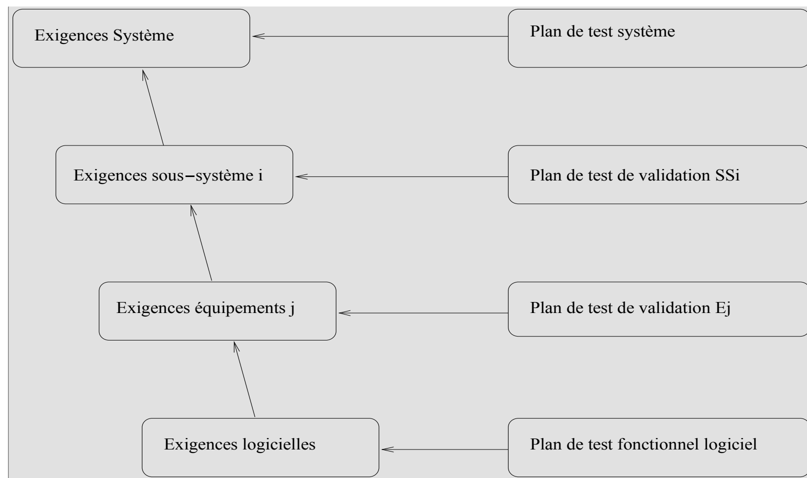
Comment [iste26]: Please provide English translation for fig annotation

– HR-1 (statements coverage): ~~the methodology using~~ Polyspace® detects unattainable code and classifies it as a non-compliance;

– HR-2 (progressive integration of software modules): the application of the methodology establishes whether or not the bounds of the output domains can be effectively produced by the modules;

– HR-3 (strong typing): the application of the method shows whether or not the use of data outside their functional domain is detected by a control violation; and

– HR-4 (tests based on a value limit analysis): the application of the methodology establishes whether or not the module input bounds are ~~attainable and expected~~ by the modules that consume them; and that an input that takes a value outside its functional domain is detected as incorrect by the function that consumes it.



Comment [iste27]: Please provide English translation for fig annotation

Figure 4.17. *Horizontal traceability*

4.7. Discussion and viewpoints

This chapter describes the verification of value control present in the source code that ensures the robustness of software with regards to dysfunctional values. This value control deals with random faults (disturbed environment, hardware failure, etc.) but also takes into account systematic failures (bugs in the application). The implementation of effective control, and the consistency between control and application, are difficult to ensure. ~~This conformance is therefore better ensured throughout the development and not solely verified during the verification phase.~~

The method that we put forward enables us to semi-automatically perform an *a posteriori* verification of the control. Our method can be extended to implement effective control from the “required control”. The method for computing the “required control” presented in this chapter is general and can be applied to add control. In this case, the original code does not contain a control point and the final verification stage is replaced by the addition of the control ~~in~~ the source code: each

required point is implanted in one or more effective points according to the location strategy ~~chosen~~.

A large portion of manual operations necessary ~~to~~ the computation of required control are difficult to carry out on software programs of industrial size and complexity without errors. The description in section 4.5 (~~then~~ applied in section 4.6) clearly reveals automation possibilities: function inputs can be calculated by an interprocedural analysis known as *in-out* computation. Similarly, software inputs can be automatically detected and the production and consumption locations can be computed ~~by using the notion of a~~ *def-use* chain. Moreover, the instrumentation necessary for the computation of the required control can be automatically generated from the source code and the functional domains. Finally, the required points can be entirely automatically calculated by combining all of these methods.

It is necessary to remark that a large proportion of ~~these computations is carried out in an intermediate way as it is useful for~~ verification of the absence of run-time errors. This information is, however, not accessible in general. This is true for Polyspace®, as well as numerous other static analysis tools that "know" a lot of things about the source code they are symbolically executing but do not ~~transmit~~ this information to the user. To our knowledge, no static analysis tool would have enabled the automation of all the computations that we carried out manually.

The verification or automatic setting of control points is much harder to automate due to ~~the different forms the effective control points can take because of the complexity of the implementation of the localization strategy chosen and because of the freedom the~~ developer ~~has in chosing~~ the best compromise between respect of the location strategy and minimization of the number of control points set. ~~To enable~~ automation, ~~it is necessary to define a more~~ restrictive location strategy ~~than the one currently used, which "in the function which consumes and at the latest between production and consumption"~~. ~~Once a restrictive rule is specified, establishing control points by~~ instrumentation of the ~~original~~ software or ~~verifying this by analysis,~~ the software ~~can~~ be entirely automated. ~~In this chapter, we study~~ this automation in a static analysis platform offering basic functionalities and allowing the development of new modules.

Comment [RE28]: Is this interpretation correct?

4.8. Conclusion

Verification of the robustness of industrial applications, such as the Thales ~~engagement~~ product, would have been impossible to carry out by hand. The computation of unknown functional domains cannot be done by hand in a precise

enough way. We have shown that it is possible to use static analysis of programs to make this verification possible.

Static analysis tools based on abstract interpretation are generally used to demonstrate the absence of run-time, memory or numerical errors and calculate abstract values to do so. We have used this functionality to propagate the functional domains of the software input data to the internal inputs. This way, the consistency between the source code and the inputs' functional domains is automatically proven: if no runtime error or non-executable statement is found, consistency is ensured. Furthermore, we have shown that it is possible to specify all the control points ensuring software robustness. This enables the definition of a verification method: if a required control point is not implemented or is badly implemented in the source code of the software, then its robustness is not ensured.

As expected, these two activities enable the demonstration of the robustness of industrial software conforming ~~with~~ the CENELEC EN 50128 standard [CEN 01a]. To limit the manual implementation time, we ~~can use~~ the method by automatically adding the control and by operating the verification during development. It ~~is~~ then possible to carry out verification of the control in an incremental way by conserving the successive audit results. One final verification ~~will remain~~ necessary on the automatic instrumentation tool, and a differential audit ~~will be needed in relation to the prior results~~.

One strength of the method put forward is that it can be carried out by people without functional knowledge about the application, since the relevant information is automatically extracted from ~~its~~ source code. Another strong point of our method is the complex use (non press button) of static analysis tools, of which there are few examples in the literature. This type of use will be increasingly important in the future as it provides greater confidence in the results, ~~since~~ part of the information is automatically calculated, and ~~more confidence in~~ the method~~, since it can~~ itself be audited.

Comment [RE29]: Is this interpretation correct?

Our method implies the use of a static analysis tool. Any tool or combination of static analysis tools that have the functionalities listed in section 4.5 can be used. We have used Polyspace® to verify a railway application and plan to carry out the same verification ~~with one or more tools,~~ such as Astrée[4] or Frama-C[5], to compare the precision of results.

Comment [RE30]: Format footnote number

Comment [RE31]: Format footnote number

4 www.absint.com/astree/index_fr.htm.
5 http://frama-c.com/.

4.9. Bibliography

[BOU 09] BOULANGER J.-L., *Safety of Computer Architectures*, ISTE/WILEY, 2009.

[CEN 00] CENELEC, *NF EN 50126, ~~Applications Rails. Spécification et Démonstration de la Fiabilité, de la Disponibilité, de la Maintenabilité et de la Safety (FMDS),~~* CENELEC January 2000.

[CEN 01a] CENELEC, *NF EN 50128, ~~Applications Rails. Système de Signalisation, de Télécommunication et de Traitement – Software pour Système de Commande et de Protection Rail,~~* CENELEC, July 2001.

[CEN 01b] CENELEC, *EN 50159-1, ~~Standard Européen. Applications aux Chemins de fer: Systèmes de Signalisation, de Télécommunication et de Traitement – Partie 1: Communication de Safety sur des Systèmes de Transmission Fermés,~~* CENELEC, March 2001.

[CEN 01c] CENELEC, *EN 50159-2, ~~Standard Européen. Applications aux Chemins de Fer: Systèmes de Signalisation, de Télécommunication et de Traitement – Partie 2: Communication de Safety sur des Systèmes de Transmission Ouverts,~~* CENELEC, March 2001.

[CEN 03] CENELEC, *NF EN 50129, ~~Standard Européen. Applications Rails: Systèmes de Signalisation, de Télécommunications et de Traitement Systèmes Électroniques de Safety pour la Signalisation,~~* CENELEC, 2003.

[COU 77] COUSOT P., COUSOT R., "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints", in *Conference Record of the 6th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Los Angeles*, pp. 238-252, ACM Press, New York, 1977.

[FAU 09] FAURE C., *Computer Aided Extrinsic Robustness Verification.* Extended Abstract. SAFA Annual Workshop on Formal Techniques, 2009 ~~(available at: www-spi.lip6.fr/~jaume/CFaureSAFA.pdf)~~.

[IEC 98] IEC, *IEC 61508 – ~~Safety Fonctionnelle des Systèmes Électriques Électroniques Programmables Relatifs à la Safety. Standarde Internationale,~~* IEC, 1998.

[ISO 09] ISO, *ISO/CD-26262, Road Vehicles – Functional Safety*, ISO, 2009 (unpublished).

[ISO 90] ISO, *Programming Languages – C. International Standard ISO/EIC9899:1990 (E)*, ISO, 1990.

Comment [RE32]: Please reference these two standards in the text

Comment [RE33]: Please reference these two standards in the text or delete them from the bibliography

Comment [RE34]: Are these references available in English? If so, please replace them with the English versions

Comment [RE35]: Is this reference available? If so, please replace it

Comment [RE36]: Has this now been published?