Constraint-Based Testing and Tail Tests for Measurement-Based Probabilistic Timing Analysis

Samuel Jiménez Gil

Doctor of Philosophy

University of York

Computer Science

December 2020

Abstract

The Worst-Case Execution Time (WCET) of tasks is an important data to give confidence that Real Time Systems will meet its timing requirements. Unfortunately, due to its tractability, this data is generally unknown. Measurement-Based Timing Analysis (MBTA), which relies on observing execution times driven by test data, has become a promising approach in the recent years. Some of the current testing techniques may take a relative long time at triggering decisions because they require very specific data of the input space. Conversely, Constraint-Based Testing (CBT) can cope better with these decisions as well as being more efficient. State-of-the-art approaches integrating CBT with MBTA have applied code coverage metrics designed for functional testing but not for WCET. Therefore, important functions such as generating test data for a path potentially leading to a large execution time are omitted. A central contribution of this work embraces CBT. Its objective is to meet code coverage needs for WCET. The evaluation compares this approach to state-of-the-art Search-Based Testing (SBT) in MBTA and Random Testing (RT) methods and shows that, in most cases, CBT not only does not underestimate the largest observed execution time but also it achieves this data earlier.

A downside of MBTA is that it normally underestimates the WCET. To face this issue execution time data is recently combined with probabilistic models. The current probabilistic protocol is hard to automate. Others probabilistic approaches have found alternative ways to achieve similar results automatically. The second main contribution aims for integrating this latter protocol and evaluating its applicability. The evaluation, which uses execution time data from test generators, shows that SBT and RT are more likely to generate data that enable this new approach. The WCET predictions are found more accurate by definition.

Contents

List of Figures

Acknowledgements

I would like to thank to *Dr. Iain Bate* who not only has supervised this work but also has contributed to my being a better thinker.

I would also thank to my father *Francisco*, my mother *María Teresa*, my sister *Raquel* and my girlfriend *Viviana* for their unconditional support.

Declaration

I declare that this thesis is a presentation of original work and I am the sole author. This work has not previously been presented for an award at this, or any other University. All sources are acknowledged as References.

The only content that has been published is listed below.

- Samuel Jiménez Gil, Iain Bate, George Lima, Luca Santinelli, Adriana Gogonel and Liliana Cucu-Grosjean. *Open challenges for probabilistic measurement-based worst-case execution time.* In Embedded Systems Letters, 2017

Chapter 1

Introduction

Embedded systems play a central role in modern society because they allow us to have a more comfortable, efficient and safer life thanks to their ability to automate some tasks. These devices are present in products and services we use daily such as transport, mobile phones or electrical appliances. As a consequence, this industry is experiencing an unprecedented growth in the last years [1]. Likewise, the size of the software packed in these systems has been growing and is expected to follow this trend in the foreseeable future. An example of this growth in the realm of embedded avionics systems is illustrated in Figure 1.1.

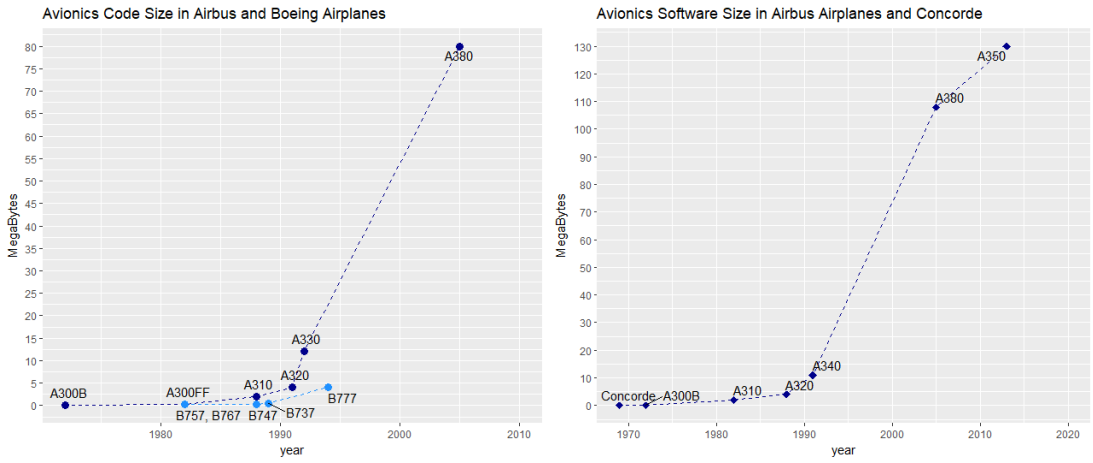


Figure 1.1: Software and code size evolution of Airbus and Boeing civil airplanes plus the Concorde. Left side data obtained from [2, 3] and rightside one from [4].

Despite the fact that computer systems offer multiple advantages, they produce some new engineering problems as well as some psychological or social ones [5]. The engineering costs become apparent when developing these systems. To meet the *quality standards* a verification process is performed as part of the development. This *quality assurance* is particularly relevant for a subset of these systems, known as *Safety-Critical Systems*, since in the event of a system not working correctly i.e., not delivering a correct output, they may endanger human life, potentially damage the surrounding area, and even cause monetary losses far higher that the cost of the system itself.

The verification costs of *Safety-Critical Systems* is around 50% in the aerospace domain [6]. Most of the *Safety-Critical Systems*, also require a correct output within a time bound [7]. These systems which must meet timing requirements are known as **Real-Time Systems**.

1.1 Real-Time Systems

The timing correctness of Real-Time Systems is only achieved when the tasks response time is no greater than their *deadlines* which are set in the specification [8]. Depending upon the consequences of overrunning a deadline, Real-Time Systems are often categorized as: A) **hard** when a failure to meet a timing constraint is considered a failure in the system. It may or may not be *catastrophic* e.g., an accident produced by a failure in the flight control systems. B) **firm** when the outcome is not useful if it is late, e.g., a camera catching a landscape where the item of interest is not available any longer but it was when the record button was pushed. C) **soft** when the result only has an impact on the performance of the application e.g., the usual *lag* when playing on-line games.

Occasionally, the *importance* of meeting the timing requirements is confused with the notion of *Safety-Integrity Levels* [9]. The Safety-Integrity Levels take into account the *consequences* (severity) of a potential failure that is handled in the system and the *likelihood of occurrence* to determine whether the Safety-Integrity Level is acceptable or not. However, the notion of Safety-Integrity Level is devised

for the functional domain of the system, and not for the timing one. For example, a software task may have a high Safety-Integrity Level but a low *importance* if the deadline is overrun and the other way around. In hard Real-Time Systems timing may impact safety so again it is mandatory to argue about what events lead to a potential deadline overrun, what the consequences are, and how likely they occur. Such an argument must consider a mitigation plan for the system when a deadline is overrun e.g., system reset.

It is important to remark that not all Real-Time Systems are Safety-Critical. A good example would be an Algorithmic Trading System where a delay in buying an asset could cause millions of dollars in loses. This system would be deemed as *Business-Critical* but not *Safety-Critical* as it has no means to cause physical damage.

To guarantee that a computing system will meet its timing requirements a necessary condition is that the system is built with an analysable timing behaviour [10]. Hence, the programming language must either provide with some guarantees of the software (timing) analysability [10] or the programmer should consider these requirements when writing the software [11]. *Worst-Case Response Time* analysis [8] is a process advocated to provide confidence that tasks deadlines will be met. Such an analysis relies on a *scheduling* algorithm which decides how to map the software tasks with the available computing resources that enable their execution. To deliver a reliable Worst-Case Response Time the Worst-Case Execution Time (WCET) of software tasks is needed. This data is not only indispensable for Worst-Case Response Time but also very hard to determine accurately. In fact, according to some conversations with people from the industry, its cost is around 1% of the 50% of the above-mentioned verification cost.

1.2 The WCET Problem

Before attempting to solve any problem it is worth questioning whether the problem is indeed solvable. From the computability point of view, when a problem does not *always* have an algorithm to solve is said to be *undecidable* [12]. Along

with decidability, it is vital to evaluate the *tractability* of a problem. The notion of tractability stems from the efficiency of the algorithm i.e., how long an arbitrary algorithm takes to process an input with size n.

An algorithm is said to be *tractable* where it exhibits a polynomial time and *intractable* otherwise [12]. In practice, intractable problems are often deemed as a deeper problem [12] than undecidable ones due to the limited hardware resources and the need to get an acceptable result in a reasonable time.

In the WCET case there are several issues that hamper its determination from a computability perspective. On the decidability side, when analyzing the WCET of a program it is assumed that the program finishes, validating such an assumption would mean solving the undecidable halting problem [13]. Likewise, the number of iterations of the loop (if they are present on the code) must be known. Calculating such a number is, in general, undecidable [14]. For this reason loops exhibited in Real-Time Systems are often written considering the need for analysability [10]. The same problem arises for recursive functions though recursion is not allowed in certain languages used in Critical Real-Time Embedded Systems [15] to avoid exceeding the memory budget [10].

Theoretically, the path containing the WCET should be feasible which means that there exists some input data that is able to exercise it. Otherwise, the results can be pessimistic. Guaranteeing path feasibility is generally undecidable [16] as well. On the complexity side, a diagram of the WCET tractability is displayed in Figure 1.2.

The outermost circle in Figure 1.2 contains the actual WCET that corresponds to *state coverage* achievement. This criterion encompasses *all plausible execution states* of a program running on a hardware platform. Considering that the execution time of a program depends on the physical environment, the underlying hardware, the state of this hardware, the interference of different components, the state of the software, the structure of the code, the input of the system and the compiler optimizations, the search space for state coverage is massive [13, 17].

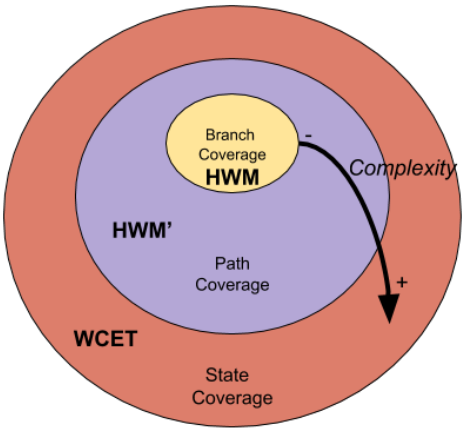


Figure 1.2: Timing estimates trades-off accuracy and complexity.

Hence, it is hard to quantify. Nonetheless, hardware specifications may help to give an approximation by analyzing the features of performance enhancement units.

After state coverage, structural code coverage metrics have been proposed for timing analysis [16, 18]. These metrics were originally devised in the *functional* testing domain and not for the timing one [19]. A striking feature of code-coverage metrics is that, under certain assumptions e.g., output of the compiler, they can be ported to different systems as they are software-based.

Path coverage is advocated to measure how many paths along the code have been traversed. For these reasons *path coverage* [13, 20] is often employed in WCET analysis. In some cases in which the execution time is mainly driven by paths, the largest observed execution time - also named *High-Water Mark* (HWM) - is arguably close the WCET. This fact is depicted in the purple circle in Figure 1.2 showing a better tractability than state coverage but worse than branch coverage. Unfortunately, although path coverage is generally more tractable than state coverage, an exponential explosion of the number paths generally occurs with the increase in software size [17].

Lastly, branch coverage consists of considering all the decisions available in the

code e.g., predicates in the control flow, loop guards. This criterion is depicted in the innermost circle in Figure 1.2 and results more tractable than path coverage. However, this criterion is generally insufficient as some *combinations* of branches i.e., a path, may raise the HWM substantially.

1.3 WCET Approaches

The former section was dedicated to explain the vicissitudes of the WCET problem. This section tackles the solutions or approaches to face it.

WCET analysis falls within a field known as *timing analysis* which is concerned with the study of the execution time bounds or estimates [13]. At the core, approaches are advocated to analyze the execution time either by *observing* execution times and select the largest one or to build *analytical* methods to compute an upper bound of the WCET. Since the WCET is generally *unknown* because of its intractability, the main objective of the analysis techniques is to give confidence we are close to it [18]. Underestimating the WCET could lead to unexpected deadlines misses. By contrast, estimations should not be overly pessimistic because it may render the system hard to schedule if such values are used.

Given the WCET absence, a confidence interval is composed by the HWM collected after testing, and, if available, a Computed WCET (CWCET) that, as we shall see, an analytical method would calculate. Regardless of the approach, during any WCET analysis the main four plausible scenarios displayed in Figure 1.3 may become apparent. Next, we explain each case in detail:

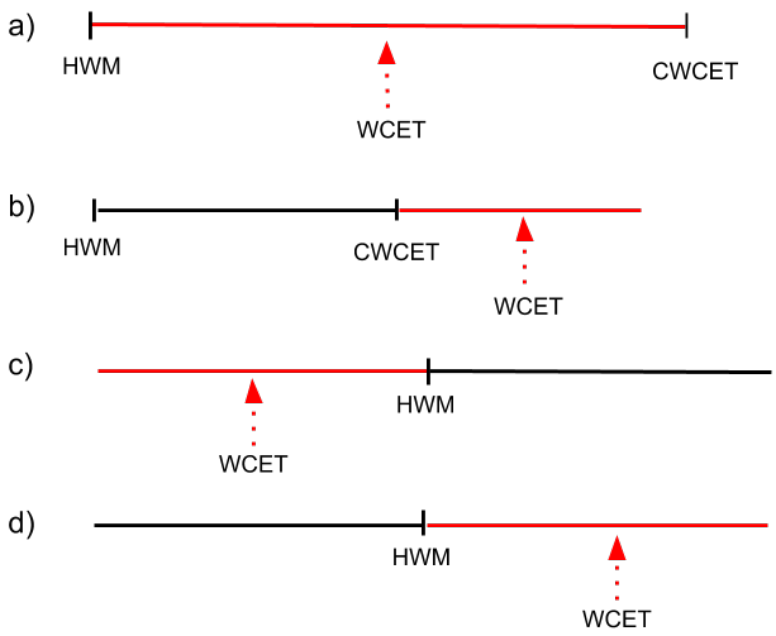


Figure 1.3: Four plausible cases of WCET analysis assuming WCET never matches either HWM or CWCET and HWM $\leq$ CWCET. Red line denotes the *uncertainty* interval where the actual WCET is.

(a) This case shows a controlled case where the WCET has a *known* confidence interval between HWM and CWCET. In some exceptional cases we may have observed the WCET when $HWM = CWCET = WCET$. This may happen in some embedded platforms whose software and execution time are very time-predictable. From an analysis perspective this is the best plausible outcome assuming that the CWCET data is reliable.

(b) In this scenario the developer is misled by the CWCET result since the WCET is greater than it. This happens when the process composing the CWCET is not reliable. Hence the definition of upper bound is violated by the CWCET. A deadline miss may occur unexpectedly on account of this wrong assumption.

(c) The third case seems to contradict the basis described at the beginning of this section since the collected HWM is greater than the WCET. This

apparent paradox can be be explained if we consider the set of inputs or events the system is tested against. An example is given in Figure 1.4.

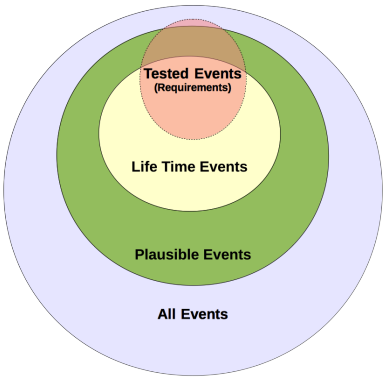


Figure 1.4: Data coverage.

From the innermost circle to the outermost we observe that: Firstly, the system is tested by a subset of events dictated by the requirements. Secondly, when the system is deployed, it will experience its own *life-time* events that will depend, in large measure, on the deployment context. Next, there is a superset of *plausible* events a system may experience followed by the superset of all events including the *impossible* ones. For the sake of applying a simple solution to the WCET problem the developer may run some tests containing input data that can not happen when deployed. Yet, the collected HWM will provide with enough confidence that the WCET of the life-time events is upper-bounded.

(d) The last situation arises when there is no process to give an upper bound of the WCET by means of a CWCET. The developer may be happy with the resulting HWM but they acknowledge that the WCET is somewhere on the right.

Irrespective of the analysis scenario from Figure 1.3, in order to calculate a CWCET or to observe a HWM there exists three main techniques [21]:

1. *Dynamic* or *Measurement-Based Timing Analysis* (MBTA) approaches collect the execution times after testing from a physical target or a repre-

sentative simulator from which the HWM is picked. They are prone to underestimate and the user must supply input data. Such an input data was traditionally generated manually, the production was expensive and it may only be available late in the development [16]. Hence, automatic techniques are sought.

2. *Static Timing Analysis* (STA) builds a model of the software and hardware to derive a CWCET. It is not concerned with the HWM, and does not require the user to provide input data. Yet, the developer may need to write some annotations in the code [10]. With the timing data of individual parts of the code along with its structure a path analysis process is able to identify the WCET path and derive a CWCET as a result. Perhaps the main limiting factor of this approach is its need to model hardware provided that modern hardware modeling is far from trivial. Let alone, hardware details may be subject to Intellectual Property clauses and thus the details may be undisclosed.

3. *Hybrid* approaches aim for combining the advantages of both methods for the sake of confidence in the results, portability i.e., extrapolation of the process to others benchmarks with different technologies and processes, or relaxation of testing requirements [22]. For example, by collecting execution times in different parts of the code (dynamic analysis) a hybrid approach would compose a CWCET from the structure of the code and thus it would apply Static Analysis techniques yielding to an approximation as if path coverage were achieved [22].

Bearing in mind that testing is seldom perfect and the WCET path may not be observed in dynamic approaches, a *safety margin* is sometimes added to the execution time data. Traditionally, an industrial practice to add a 20% to the HWM [21] was used, however this figure has an unclear justification. It used to be added when the software ran in time predictable architectures so originally it was advocated to handle imperfect path coverage.

In the recent years, the safety margin has been defined by using *probabilistic approaches* whose original objective was to compensate for the lack of *path* or *state* coverage. The upside of these methods is their ability to argue about how likely a certain execution time will be exceeded. Unfortunately, to compute the results an elaborate statistical process must be carried out and that is why automation of the calculations is sought [23]. This requirement is hard to fulfill with state-of-the-art approaches. Furthermore, when the final CWCET is estimated using a probability distribution, current methods disregard the asymptotic idiosyncrasies of the distribution used as well as common *uncertainty* in the execution time data. In sum, these inaccuracies negatively damage the confidence of the CWCET [20].

Outside the realm of probabilistic analysis for MBTA (MBPTA), some other works [24] have applied an equivalent analysis but with some advantages. These approaches use different probability distributions that enable full automation of the analysis process. A careful application of this approach may also improve the CWCET results.

1.4 Test Generators

Test data plays a central role in Hybrid and MBTA. Automatic test generation is an area dedicated to decide such test data. To give an example of its importance, Tracey in 2002 speculates that the automation of test data generation may save between £1 million and £1.5 million in jet engine controllers [16]. Some works have extrapolated these techniques to MBTA [16, 18]. However, the objective of the Test Generators differ. These objectives consist of:

1. **To test specific parts of the code so as to collect their local HWM** e.g., decisions, that are later used for a path composition algorithm to calculate a **CWCET** [25]. Obviously, this would correspond to scenarios a) and b) from Figure 1.3.

2. To traverse the paths leading to the largest execution times [18] i.e., **maximizing the HWM**. This objective is more suitable for the most common

MBPTA which is concerned with the prediction of extreme events [26].

To our knowledge, there exists 3 types of test generation applied to MBTA:

- **Random Testing** (RT) consists of assigning random values to the input data. It has a trivial implementation but is generally awkward [18]. Its implementation may be static or dynamic.
- **Search-Based Testing** (SBT) is a *dynamic* test generation embracing *meta-heuristics* which consists of informed search algorithms advocated to find a *good* solution in a *reasonable* time [27]. Metaheuristics require to be provided with proper guidance for a successful search by means of an *objective* function. In the SBT arena, the guidance is often supplied after instrumenting the Software Under Test (SUT) which generates overhead [28]. Moreover, this approach may struggle when targeting decisions whose activation depends on very specific values of the input domain [29, 30] e.g., equalities. These untested decisions may contain some loops that have a great impact on the HWM or CWCET so this approach might not be ideal for achieving a great degree of code coverage. Despite this limitation, SBT has been embraced extensively for MBTA [16, 31, 18].
- **Constraint-Based Testing** (CBT) [32] targets achieving the greatest degree of coverage as possible. It tackles test-generation as a Constraint Satisfaction Problem (CSP), another field of Artificial Intelligence advocated to find a solution considering a list of constraints.

  Sometimes a function to optimize is added in which case a constraint-optimization problem [27] is addressed. This solution has been embraced in STA when estimating a CWCET from a graph with execution time data [33]. Unfortunately, the calculation process is oblivious to infeasible paths [33] which is a source of pessimism. The same constraint solvers could be useful at identifying them. Yet, it seems that so far, this process is inefficient and can only focus on a subset of infeasible paths [34]. Infeasible paths is also a problem in some hybrid approaches [35]. Because of the way they

compose the CWCET information about the infeasibility must be provided manually.

Even though some of its underlying solving algorithms are NP-hard [27], constraint solvers have gained momentum in the recent years [36]. This is due to the significant improvements both in the solving algorithms and hardware processing power. Despite its disadvantages, CBT has been claimed to be the best test generation technique for code coverage [37]. This is because in contrast to metaheuristics, which may struggle with discrete search spaces, CSP can cope with such a non-continuous space.

Even though the WCET literature often focus on the confidence of the results [13], in an industrial setting, it is relevant to argue about the *efficiency* of the analysis applied. This is because normally any industrial project, results are due by a deadline [18]. Thus, even though a WCET analysis process is automatic, attaining results in a reasonable time is essential.

Eventually, the portability of the WCET analysis process is important to allow the application of a solution in a wider array of problems. To meet this objective software-based path composition processes are often employed [35, 32].

1.4.1 Path Explosion

A central challenge for the use of CBT is the collection of constraints. Deciding the list of constraints to formulate a CSP is a key step to maximize the effectiveness of CBT generate test vector targeting code coverage. This implies that some form of exploration of the structure and collection of values from the SUT. As a consequence, path explosion emerges as the main problem to deal with.

Several approaches have been put forward [38] such as the **use of sound Static Analysis techniques** to reduce the number of paths to be analyzed or using **heuristic techniques** to prioritize paths which may be more relevant than others.

Current CBT approaches in MBTA explores a graph model by using Depth-First Search (DFS) [39]. The resulting analysis is only able to generate branch coverage test suites. This is due to the fact that such a Test Generator (TG) is designed for functional testing but not for MBTA. Hence, an obvious objective like maximizing loop iterations with test data can not be achieved by this test generation process. Let alone, this method is not concerned with the notion of paths which is very important for WCET analysis. Another downside is that these approaches derive an inefficient graph model because it collects unnecessary information for the application of CBT for MBTA [40]. Such an inefficiency slows down the search time and increases the memory space when search algorithms are applied [39].

In summary, a path exploration process targeting relevant paths for the WCET so as to generate test data is needed. Heuristic approaches could help at targeting these sort of paths whereas search algorithms such as Best-First Search (BFS) strategies [27] may be instrumental to deal with the *efficiency* by focusing on the greatest execution times in the first place.

1.5 Research Challenges

Given its ability to achieve a good degree of code coverage, CBT results in a promising approach to be applied to MBTA. However, some challenges are identified and they are outlined as follows:

1. To identify search strategies that can be integrated with the former process so as to build paths that can potentially lead to the largest execution times first.

2. To conceive an optimal program slicing that eliminates the inefficiencies of the current CBT applied to MBTA.

3. To investigate how the use of constraint solvers applied to CBT may also help at identifying infeasible paths and thus reduce the pessimism in CWCET estimates.

Lastly, given the described issues with modern MBPTA [23] and the identification of potentially better approaches [24] the last challenge is.

4. To devise a novel MBPTA to achieve full automation and calculate more confident results.

1.6 Hypothesis

Our central hypothesis is:

> The proposed Constraint-Based Testing process provides the best test generation process in terms of increasing the largest observed execution time and collecting this result earlier than state-of-the-art approaches. The novel probabilistic analysis is able to derive safety margin with an automatic process and its results are more confident than standard approaches.

1.7 Thesis Outline

The rest of the thesis lays out as follows: Chapter 2 outlines the literature survey and provides some relevant fundamentals for the WCET analysis problem. Chapter 3 describes the suggested CBT process and tackles challenges 1,2 and 3. Chapter 4 relaxes some assumptions of the former chapter and provides additional case studies for the proposed approach. Chapter 5 addresses Challenge 4. Finally, Chapter 6 offers the conclusions of the overall thesis.

Chapter 2

Literature Survey on Worst-Case Execution Time

The first approaches to tackle the WCET problem are based on Static Analysis [13]. As stated earlier, this strand analyzes the flow of the program which is later combined with time data from analysis of the hardware. By integrating both processes with a set of rules dictated by the structure of the program, a CWCET is derived. Measurement-Based Timing Analysis (MBTA) and Hybrid have gained momentum in the recent years [17] because they do not require hardware modeling. Despite this advantage, two central challenges stand out: a) Deciding test data and b) how to handle underestimations. As for a) test generators literature is useful. Paradoxically, Static Analysis techniques are relevant for test generators and Hybrid approaches. With respect to the b) problem, a new promising approach on probabilistic analysis may improve state-of-art methods.

This chapter commences outlining Static Analysis in Section 2.1. Next, in Section 2.2 dynamic approaches are detailed. Section 2.3 explores the literature of test generators in order to face challenge a). By contrast, with the aim for providing a probabilistic solution to challenge b) Section 2.4 offer a literature survey on Measurement-Based Probabilistic Timing Analysis (MBPTA). Lastly, Section 2.5 summarizes the content of the chapter and outlines the research contributions.

2.1 Static Analysis

Static Analysis is a set of techniques whose purpose is to calculate safe approximations to the actual values computed at *run time* [41]. The reason for calculating approximations is that calculating exact computing values would raise decidability issues [41]. The accuracy of such estimations trades-off the computation time of the analysis [42].

These techniques are common in program verification and compilers since they are able to anticipate the violation of certain assumptions. Further, they are able to recognize *redundant* or *unnecessary* software statements which may help to improve the integrity and performance of the program [15]. Most notably, Static Analysis techniques for our problem domain include:

1. **Data flow** analysis consists of calculating or approximating plausible values in a specific point of a program q. It achieves that by representing the program under analysis by a Control Flow Graph (CFG) [41]. A CFG is often deemed as the standard representation of programs [6]. In a CFG, each node represents a block whereas directed edges describe the control flow to go from one node to another. A CFG also comprises an *entry* and *exit* node.

2. **Abstract interpretation** may be deemed an instance of data flow analysis. It aims at computing approximate safe values by mapping a program states set and even memory locations to a finite set of *abstract states*. This mapping is delivered by the so-called *abstraction function* [11]:

   $$\alpha : L \to M$$

   where L and M are complete lattices. By contrast, the opposite to α is the concretization function γ:

   $$\gamma : M \to L$$

   This function is in charge of returning the approximate values of the computation from the *abstract domain*. By employing this technique approximate

values in a specific program point, q, can be derived. Such a q can be an arbitrary node or edge in a CFG.

3. **Program slicing** consists of analyzing statically the program but omitting some parts that are deemed unnecessary for the analysis [11]. The heuristic is usually defined with the pair $\langle q, V \rangle$. Where q has the above definition, and V is a subset of the variable in the point q. Considering $\langle q, V \rangle$ a program slice would collect *all* statements which may have an impact of the variables in V. The program representation that enable an efficient program slicing is the *Program Dependence Graph* [11] which is often a subset of the CFG.

4. **Symbolic execution** analyzes the program by replacing input variables by symbols. Other symbols may also be added when an unknown function is called [43]. In order to deliver the execution, a *symbolic execution engine* is employed. Such an engine stores information of each control flow path about the branches traversed up until an arbitrary point q. Having reached the statement of interest the conditions to check are supplied to a constraint solver which returns, if there exist, *concrete values* to traverse that path and evaluate that property e.g., checking whether an assertion at the end of a function will be violated [43].

More precisely, Static Timing Analysis STA uses additional techniques to be able to calculate an upper bound of the WCET. The derivation of the CWCET is delivered by these three phases.

1. *Instruction Set Architecture* level infers flow information including loop bounds from the Software Under Test (SUT).

2. *Microarchitectural* analysis consists of calculating execution time in a specific hardware of basic blocks. A block is a list of instructions that does not contain any jumps [44].

3. *Path analysis* integrates information of the previous two stages to derive the longest execution path. This means that despite path coverage intractability, STA is able to consider the execution time of all paths. This is

because a linear constraint solver is able to compute that global CWCET from a graph containing the structure of the SUT along with local CWCET computed using static analysis. As said in the former chapter, the compositions of the CWCET often entails including infeasible paths which is claimed to be a source of *pessimism* [34]. These techniques are inefficient because they often add unnecessary computations to the already complex problem of path analysis. Let alone only a subset of infeasible paths is analyzed because of this issue and also some states can not join for the analysis.

Static Timing Analysis (STA) literature offers a wide array of loop analysis methods that are quite relevant for our problem domain since counting the number of iterations of loops can give us an indicator about what paths could potentially lead to the HWM.

In this respect, the seminal loop analyses relies on annotations supplied by the user [45]. Nonetheless, this method requires human intervention which in turn requires understanding of the code. This may be troublesome since code generators are often employed in the embedded industry [19]. Let alone annotations data must be acceptably right. Such surmises may not be true in reality [7]. On the contrary, STA techniques provide *automatic* techniques to calculate loop bounds [10]. Some of these techniques employ a loop pattern matching approach to calculate the maximum number of iterations [10]. Unfortunately, compiler optimizations like loop unrolling may introduce a mismatch between the source and object code. The only commercial available STA tool, aiT [46], provides with different loop patterns that are generated by different compilers to complete this verification step.

In the context of object-oriented programming, Gustafsson [42] apply abstract interpretation for timing analysis. This analysis unrolled every single loop rendering a state space quite large and thus struggling with large pieces of code with loops. As a result, for abstract interpretation to produce good results the accuracy had to be diminished towards the end of the analysis. More recently, Lokuciejewski et al. [11] uses a combination of abstract interpretation and *Ehrhart polynomials*

for loop analysis. The motivation of this method stems from the observation of a wide array of industrial real-time benchmarks display loops whose statements do not affect the computation of loop iterations. It is fair to say that this is an important prerequisite to write time-predictable software [10]. The advantages of this approach are that they are not only much faster to analyze than traditional loop analysis delivered in abstract interpretation but also it avoids pattern matching for loops in the analysis. Soon after that, Barlett and Bate [14] define a method to tightly calculate iterations in non-rectangular nested loops i.e., loops whose iterations depends on outermost loops, as standard static methods compute pessimistic results. The approach is based on inductive data which are computed by means of testing. These data are later supplied to some mathematical formulas which calculate the number of iterations. Such an analysis is able to cope with up to 8 levels of nesting that was believed to be a sound bound of nested loops in industrial code.

Given that calculation of the loop bounds is undecidable in general, the loops implemented on real-time software are restricted to the so-called **Presburger** subset of integer arithmetics. Intuitively, to calculate the number of iterations sum series are normally employed [47, 10] but this series can not be applied to every sort of arithmetics. The Presburger subset of arithmetics restricts the summation limits to: integer constants, symbols representing constants, bound variables which stand for index variables for an arbitrary loop, operators $+$, $-$ and multiplication by an integer constant [10]. It is fair to say that this is the common practice to write loops for a timing analyzable piece of software [10, 14].

A drawback of STA is that they are prone to significantly overestimate the WCET though they are able to give tight figures in specific cases due to the success at modeling the systems and all its plausible inputs. In spite of these advantages, they are very dependent on the model of the processors whose hardware design may be subject to Intellectual Property rights [48]. Further, hardware modeling may not be correct due to errors in the implementation or in its documentation. The advent of performance-enhancing hardware units such as pipelines, caches, branch prediction or speculative prediction has exacerbated the issue of hardware

modeling [21]. Recently, it has been claimed that STA has reached its limitation due to these difficulties at modeling modern and complex hardware [17, 18]. In essence, because of the nature of STA, it struggles severely with the *portability* objective.

2.2 MBTA and Hybrid Approaches

Unlike Static Analysis, MBTA requires collecting execution times from a real embedded target or a representative simulator [13]. By running the software with different input data, execution times are triggered and they serve as a empirical evidence of the performance. Due to its nature, this approach is the easiest to port. The central downside is that it normally *underestimates* the WCET which is normally deemed a worse situation than overestimation as it may produce an unexpected deadline overrun.

In the middle ground, Hybrid approaches combine the advantages of both approaches [21]. By observing execution time data hardware modeling is not necessary. Next, this data is processed with static **path analysis** techniques. To do so, they divide the SUT into different components, whether it be branches, subpaths, or others parts so as to collect execution times in all or many of these a components [49, 32]. This data is later used to derive a global CWCET. By embracing this approach the developer does not have to supply test data to traverse paths leading to the highest execution times and sometimes she may reuse test data from structural code coverage testing.

Some of these techniques are orientated to provide results as if path coverage were achieved [49] whereas others encompass both path and state coverage [32]. In the first case, the path composition may include infeasible paths and thus increase pessimism that is only reduced if the developer provides with some annotations [35]. The underlying reason is because some path composition approaches only read the structure of the code but not the data flow [50]. Perhaps the main upside of hybrid techniques is their ability to relax exhaustive testing as this path analysis compensates, to an extent, for the underestimation [51, 20].

In the second case, the SUT is divided into *"segments"* [32] to which test data maximizes the execution time [25]. The observations of these segments are collected in a multicore architecture and thus the hardware may substantially increase these local HWM or render a great deal of underestimation because the maximum has not been properly exercised with the input data. In conclusion, it is hard to argue about how realistic the CWCET is as the flow of the execution time is very sensitive to the hardware architecture whereas the flow of the program is what is normally modeled.

Unlike, state coverage, code coverage metrics are easier to quantify. The notion of **structural code coverage** or just code coverage stems from functional software testing [6]. One of the main problems traditionally in functional software testing was to systematically test software and code coverage aims to give a solution to it. Particularly, structural code coverage testing is developed to tell how much the requirements are implemented in the code [19]. It delivers results by executing test cases. The notion of test case - according to the definitions of civil aerospace safety standard or DO-178C [19] - is composed by a **test vector** (or input data), execution conditions and expected output data. A set of test cases is known as a test suite. It is worth noting that, from the certification perspective, a test case must be linked to the requirements or the specification. Otherwise, no verification would be performed on the system [52]. More precisely, code coverage testing checks whether I) requirements are complete with respect the code, II) test suite is complete, III) There is no code to be deployed that should not be there.

There exists a range of coverage criteria that indicates how much input data has traversed the SUT. Only a subset of these metrics are of interest for our WCET problem domain. In particular, *block coverage* also known as statement coverage, consists of covering all *statements* of the SUT at least once. *Decision Coverage:* Also called *branch* coverage, it is achieved when all *feasible* decisions are visited at least once. In other words, the decisions that *if-else*, *switch-case*, ternary operator *? :* , and *for, while, do-while* loops.

Code coverage test cases are sometimes imperfect or poor, which is to say some part of the code are not tested. This issue is known as *non-covered* code in the verification process [19]. Non-covered code lays bare a weakness of the MBTA and Hybrid approaches in the sense that their results are only reliable as long as good coverage is achieved. By having non-tested blocks or branches path composition algorithms may struggle and the confidence of the HWM may strain credulity. Admittedly, this is a point in favour of embracing STA techniques.

A relevant coverage metric that, to the best of our knowledge, is not addressed in the safety standards but is certainly relevant for WCET is *path coverage*. The number of paths has been claimed to grow exponentially with software size [17] however that is not always the case [53]. It is estimated that the number of atoms in the universe is 10^{82}. Having a code with a sequence of *if* conditionals would only need $82 \cdot log_2(10) = 273$ decisions with non repeated conditions in the predicates would probably have more paths than atoms the universe.

The last and arguably hardest to *control* testing achievement is state coverage. Such a coverage would entail to collect all plausible execution times and thus record all plausible execution states leading to the actual WCET. The underlying hardware the SUT runs on is the contributing factor that has the last word on deciding how big the search space is. Performance acceleration units such as caches, multi-cores with different level of caches or bus arbitration have a massive impact on the execution time [44]. A truth which must not be shirked is the fact that in some scenarios state coverage trivializes code coverage criteria as a metric to describe the execution time. This is because - as said in the introduction chapter - the execution time depends on a wide array of independent variables and the software structure is only one of them.

Apart from the hardware platform and its equipment, there exist some pieces of software (See Listing 1 in the Appendix) where the implementation contains large blocks delivering arithmetic operations. The execution time may be sensitive to the operands because of the underlying numeric algorithms [54]. In these cases,

to achieve a confident *HWM* is not only necessary to hit the block but to identify what operands or test vector maximizes the execution time of these blocks. This issue, deciding the test vector is the bulk of the work and is discussed in the next section.

2.3 Test Generation for Measurement-Based Timing Analysis

As stated in the beginning of this chapter, a central challenge of dynamic approaches is to decide test vectors to be fed. Therefore, test generation is our point of departure. Apart from Random Testing (RT) which does not really have an objective, there are only two test generation methods to be investigated for MBTA. On the one hand, Search-Based Testing (SBT) aims at finding extremes of an objective function by a process similar to trial-and-error which entails trying numerous test vectors. Unfortunately, the theory of meta-heuristics [27] is not concerned with telling whether the extreme observed is the global extreme or not. On the other hand, Constraint-Based Testing (CBT) relies on an accurate mathematical description to find, if there exists, a solution by using the constraint solver. Such a solution is indeed the test vector. It goes without saying that this approach can only be applied as long as the process can be modeled mathematically which, in the WCET problem, is arguable. More formally, a Search-Based Test Generator (SBTG) and CBTG may be defined as follows.

Definition 2.3.1. *Be I the input variables and D the domain of these variables. A SBTG, is a tuple $\langle I, D, M, F\rangle$ where M is a metaheuristic and F an optimization function. M seeks what values of I optimizes F.*

Definition 2.3.2. *A CBTG, is a tuple $\langle I, D, C, S\rangle$ where C is a tuple $C = \langle c_1, \cdots, c_k\rangle$ of constraints with $k \in \mathbb{N}$ that I must meet. S is a constraint solver that finds values, if there exists, for I considering its domain D, and a set of constraints $C' \subseteq C$.*

The rest of this section first scrutinizes the literature on SBT and then CBT.

2.3.1 Search-Based and Random Testing

When it comes to SBT, the seminal work devising search algorithms for testing Real-Time Systems harks back to 1997 by Wegener et al. [55]. In this work Wegener et al. employ Genetic Algorithms that unveil more extreme execution times - both smaller and larger - than the ones identified by manual and RT. Later on, Wegener [56] shows how *Evolutionary Testing* i.e., test vectors generated by Evolutionary Algorithms which consists of a subset of metaheuristics, was able to attain a greater HWM in comparison to the test vectors executed by the developers in an industrial setting. Conversely, Groß [57] exposes some limitations of evolutionary testing like lack of branch coverage when SUT exhibits deep nesting structures. Additionally, Groß defines a prediction metric of the *testability* i.e., how easy the software to test is, of a piece of software by means of Evolutionary Testing. Likewise in 2002, Tracey [16] embraces a Genetic Algorithm and Simulated Annealing to generate test vectors for structural code coverage and for MBTA for the control software of a jet engine. In his Ph.D. Tracey [16] defines fitness function for different predicates often found in programming language but to achieve this guidance the SUT must be instrumented.

Despite its momentum, SBT was not without its downsides. In 2004, McMinn [29] contends that SBT has some relevant limitations like their inability to trigger decisions whose predicate satisfiability depends on very specific data of the input domain e.g., equalities such as `(x - 2 * y) == 1`, deep nesting structures or the fact that the test data generated may be completely pointless. That is why the test vectors trigger a number of exceptions that probably would not be reached when the system is deployed (disjoint union of plausible and all events in Figure 1.3).

The same year, to cope with this issue Harman et al. [58] apply the so-called *Testability Transformations* which are derivations of a SUT to facilitate test generation. Yet, these transformations will produce valid test data for the original program. Results of this work give confidence that the former difficult branches are triggered, however it requires a big number of test iterations i.e, in the order

of thousands, to trigger such branches for the transformed and optimized version. In addition, some of these program transformations may not be accepted by Spark or Ada compilers as they might produce some overly long statements of conditions.

Testability transformations have also been applied to timing analysis [59], but they have different objectives and techniques. For example, one transformation is orientated to linearly reduce the execution time of the SUT. This technique still would not be valid for Critical Real-Time Systems as the analyzed system is not the same one as the one deployed. Other techniques like smoothing the landscape of execution times, may be more promising as they are less invasive with the SUT since they only alter the execution time landscape.

The advent of hardware performance-enhancement units imposes stronger requirements to test generators due to enlarging the state space as well as executing the worst-case timing behaviour of these units. In 2009, Khan and Bate [31] develop several objectives to guide the search, including hardware states e.g., maximizing cache misses. One objective of this work is to establish what optimization functions are the most appropriate depending upon the properties of the SUT. As a result, the heuristic just optimizing the execution time gives the best results generally. The main limitation of this experimental framework method is that it reads profiling performance counters that are only available in certain processors from ARM architecture [18].

More recently in 2016, Law and Bate [18] come up with another SBTG implementing a Simulated Annealing. Some of the optimization functions were crafted considering certification requirements of covering the blocks of cutting-edge jet engine control software for which a guidance of stressing the structure of the code was needed. Their main contributions are the optimization functions driven by the guidance of the only commercial available instrumentation tool for MBTA [35]. Some of these fitness functions allow, for instance, to maximize the loop counts and focus on non-tested blocks. This latter heuristic gives the best results for 25% benchmarks that happened to be industrial controllers. Aside that, the

heuristic just maximizing the execution time generally performed the best. For the majority of the benchmarks the variability across different fitness function was less than 10%. In spite of the fact that some fitness functions target achieving good coverage and hit around 90% of the instrumentation points the analysis of the results do not say how much such new functions help at triggering pathological branches like the ones identified by McMinn [29].

Lastly, RT hinges on feeding random test vectors to the SUT. This approach is the simplest to implement but it is perhaps worse than SBT at achieving a good degree of coverage. Godefroid [30] points out that this sort of testing would only hit a simple equality branch such as `x == 10` in 1 out 2^{32} having a 4 bytes integer.

2.3.2 Constraint-Based Testing

While SBT is the application of meta-heuristics to testing, Constraint-Based Testing (CBT) is the application of Constraint-Satisfaction Problem (CSP) to testing. A constraint optimization problem would also include an objective function to optimize [27] but that happens to be irrelevant for our problem domain. More formally a CSP is defined as:

Definition 2.3.3. *A CSP consists of a triple (I, D, C) in which:* $I = \langle i_1, \ldots, i_n \rangle$ *is a n-tuple of input variables,* $D = \langle d_1, \ldots, d_n \rangle$ *is also a n-tuple of domains associated to each variable in* I*. A constraint tuple* $C = \langle c_1, \ldots, c_k \rangle$ *holds a list of* k *of constraints that the input variables with their domain must meet.*

The objective of CBT is to execute as many paths from the SUT as possible by identifying appropriate test vectors [37]. The first techniques to compute test vectors based upon program analysis date back to the 70s [60]. In this work King [60] applies for the first time *symbolic execution*. The analysis was applied to small pieces of code and the processing power was a limitation. In 2002, Henzinger et al. [39] applies CBT to functional testing by using Static Analysis of the code. After parsing the code a graph known as Control Flow Automaton (CFA) is derived. Unlike standard CFG, the notion of CFA [39] has instructions on the edges rather than in the nodes. These instructions enable later to generate

a test vector to hit a specific statement [40]. More importantly, it collects *all* the statements from the SUT even though some of them are not necessary to apply CBT to achieve branch coverage. To be able to produce a test vectors set to hit branch coverage, DFS [27] is launched on the CFA. Notwithstanding, due to the great deal of states the CFA collects it explodes when the program to analyze is big.

In 2008, Holzer et al. [40] devise **FShell**, a *query-driven* test generator that provides with a layer of abstraction to the Henzinger et al.' [39] CBTG. The breakthrough was to express in the form of query the generation of test vectors to meet a coverage criteria after parsing a source code written in ANSI-C.

An inherent problem to software testing is path explosion [38]. To deal with the problem two main strands have emerged: I) **To use heuristics methods** in order to focus on a subset of paths. These approaches collect information statically from the CFG or use dynamic information from the branches traversed at run time to guide the heuristic. II) **Sound static analysis** may alleviate path explosion by synthesizing information of previous subpaths explored that is reused to explore other paths of interest. Other techniques are advocated to merge paths and thus reduce their number.

When it comes to graph search strategies like Best-First Search BFS [27] also called Branch and Bound [47] are advocated to identify the best solution or a combination (i.e., path) in graph exploration problems in the *first* place. Key algorithms in Artificial Intelligence such as A^* or Dijkstra are based on this kind of search [27]. The price to pay is to elaborate a bound of the cost of getting at the goal node [47]. This bound is used as guidance and serves to choose amongst candidate nodes to explore. Though BFS is challenging to implement, this heuristic may be instrumental in composing a path to analyze as well as attaining the results as soon as industry wishes [18].

It is important to remark that some works have integrated CBT and SBT [61, 62, 63]. For instance, Xie et al. [61] supply dynamic symbolic execution with a

path heuristic computed dynamically so as to focus on unexplored paths. Results show how this combination generally outperforms both in efficiency i.e., number of test vectors produced to meet the coverage objective, as well as the degree of coverage to random and Breadth-First Search. By contrast, in comparison to the default Dynamic Symbolic Execution, the coverage results are relatively similar whereas the greatest impact is in the optimization of the performance. Unfortunately, the evaluation does not show a comparison with a metaheuristic. The other interesting feature about this work is that this hybrid test generation heuristic may sometimes be led to an unfruitful direction which would damage its performance w.r.t random and breadth-first search.

Conversely, Baars et al. [63] aim for integrating the information supplied by symbolic execution to an enhanced fitness function to be used in SBT. The results does not generally show a significant improvement in branch coverage w.r.t standard fitness function but certainly increases the performance of the enhanced test oracle to meet the coverage objectives. Alternatively, Galeotti et al. [62] propose a dynamic switch between the symbolic execution and SBT. Though the combination details of both strategies is an open question, their evaluation shows that symbolic execution should be run for a relatively long time after a test vector improves its fitness. On average, their hybrid test generator improved branch coverage on 11% with respect to a standard Genetic Algorithm and pure dynamic symbolic execution.

In the realm of applying CBT to MBTA, in 2009, Zolda et al. [32] employ FORTAS [64], an analysis framework for the reduction of optimism i.e., underestimation, for MBTA. Such a framework is equipped Instruction Path Enumeration Technique (IPET) [65] an algorithm to derive a CWCET from timing observations along the code. FORTAS employs a divide and conquer algorithm to divide the program into *segments* [32] which are then supplied to IPET to derive a CWCET. The test-generation part of FORTAS is delivered by FShell [66] despite the fact that it is a code coverage test generator and is not concerned with the notion of paths. Since FORTAS framework uses segmentation and IPET to derive a CWCET Bünte et al. [67] believe that a good way to increase the confidence of

the CWCET was to maximize the local HWM of the segments from which the global CWCET was derived. For this reason Bünte et al. devise Balanced Path Generation I and II [67] code coverage-based rules to guide FShell to generate test vectors to increase this local HWM of these segments.

After that, Bünte et al. [25] devise FORTAS Reduction of Optimism a test generation process which combines FShell test generator with a Genetic Algorithm. To the best of our knowledge this is the only work that combines CBT and SBT for MBTA. Such a protocol guarantees, when feasible, branch coverage thanks to FShell. The resulting test vector is mixed with random test data only in the first iteration. After that, test vectors are engendered by the Genetic Algorithm. FORTAS results outperform Balanced Path Generation both in efficiency and maximization of the CWCET. However, if there is no care when mixing constraint-based test vector with the data generated by the GA the resulting test vector may struggle to hit the type of branches as displayed in Listing 3.1. Furthermore, these last cases where FShell was used they were advocated to increase the CWCET in a complex target architecture equipped with three cores [32]. This objective of increasing the CWCET and not the HWM may not be useful if probabilistic approaches are employed whose extreme results verification relies on extreme observed data [68]. Furthermore, as noted before, since the CWCET is conformed by hardware states perhaps the resulting figure may be unrealistic and overly pessimistic.

2.4 Measurement-Based Probabilistic Timing Analysis

Considering that the largest execution times may be hard to trigger by test data and in normal operation may occur infrequently, MBTA may be combined with probabilistic models which are able to justify statistically a safety margin and thus MBPTA is engendered. *Extreme Value Theory* (EVT) is a field of statistics dedicated to the predictions of extreme unobserved events [69, 70, 71]. In a

nutshell, the application of EVT consists of the following steps [72]:

1. **To test the performance of the SUT empirically and collect execution times**. Then for this data the hypothesis of identically distributed and/or independence must be checked for $(X_i)_1^n$, where $(X_i)_1^n$ is the list of observations X_i, $i = 1, 2, \ldots, n$ collected.

2. **Choosing the maximal observed data** from $(X)_1^n$ since this data is the only one of interest for the EVT fitting. Two hitherto methods of selection are Block Maxima (BM) and Peaks-over-Threshold (PoT) [73]. The former consists of partitioning the sampled data $(X)_1^n$ into equally sized blocks, whose sizes are specified beforehand b. Then the maximum of each block is picked. The latter selects *all* values (k) in $(X)_1^n$ above a certain previously defined threshold, u_n. In either case, the block size b or the threshold u_n should be supplied.

   It is worth saying that this decision is hard to automate and is the main obstacle to automate EVT analysis computations [23].

3. **Evaluate whether EVT can be applied** which is determined by examining the maxima data derived in the previous process and checking whether this distribution converges to any EVT distribution [74].

4. **Fit an EVT distribution** which is obtained by fitting extreme observed data. The resulting distribution may be a Generalised Extreme Value Distribution (GEV) which would be attained by using the BM principle or a Generalised Pareto Distribution (GPD) obtained using the PoT selection. The shape parameter (ξ) serves to differentiate the name of the GEV distribution, namely, Weibull $(\xi < 0)$, Gumbel $(\xi = 0)$ and Frechet $(\xi > 0)$.

5. **The verification of the distribution** is checked by comparing the curve of the maximal observations to the probability distribution described by the parameters (average μ, scale σ and shape ξ). Additionally, sometimes specific extreme events prediction of the tail are compared against an empirical data using *scoring* rules [75].

6. **Calculating a probabilistic WCET (*pWCET*)** from the resulting EVT distribution. This is delivered by estimating a value $q(p)$ associated with exceedance probability p such that $P\{X_i > q(p)\} = p$.

In reference to the literature survey, the seminal paper in applying EVT to timing analysis is by Edgar and Burns in 2001 [76] where they use Gumbel distribution to derive a pWCET. However, Edgar and Burns fit the distribution with raw data rather than selecting the maximal data of the sample as EVT analysis dictates [69, 70].

In 2005, Bernat et al. [77] applies another form of probabilistic analysis by using *copulas* to the execution time data collected in instrumentation points. The notion of copulas consists of the calculation of a new probability distribution from two empirical ones [77, 78]. While convolution is applicable when two distributions are independent, copulas can be considered its counterpart when these distributions are *dependent*. One of the upsides of copulas is that they enable computing a lower or upper bound from the resulting distribution. However, they are very hard to compute due to the exponential complexity of the number of distributions as input [77].

In 2009, Hansen et al. [79] revisits EVT and applies Gumbel distribution as well. Yet, Hansen et al. do not check the independence and identical distribution of the supplied data as EVT analysis mandates [69, 70]. In 2010, Griffin and Burns [80] stands on central assumption in any distribution. That is that all predicted values of the curve can be possible in reality which is not true in timing analysis. To set an example he proposes a program with few and specific execution times and shows how the Gumbel distribution underestimates and provide *unrealistic* values. Later on, Cucu-Grosjean et al. [26] investigates the EVT requirements so as to design new systems whose resulting execution times can be successfully supplied to EVT. To achieve this, they conclude that a randomization of the execution times is needed which includes the need to remove the dependence of previous execution times to meet the independent hypothesis. This occurs for instance when a cache holds data of previous execution.

To meet this objective, Kosmidis et al. [81] implements time-randomization either by software [81] or by hardware [82]. Hardware randomization consists of implementing random allocation of memory blocks in the caches or random bus arbitration in the case of multicores. On the other hand, software randomization [81] randomizes the memory layout by using special compilers and run-time allocators. The latter is unlikely to be accepted from a safety point of view since some memory layouts may not be tested when deployed [23]. Perhaps the main upside of using time-randomization is the fact that the developer can cover execution time states just by using structural test vectors. Unfortunately, this approach is advocated to indeed achieve state coverage which - as pointed out in Section 1.2 - has the worst complexity. Another downside of these architectures is the fact that same test data triggers not only one but random execution times. Such a consequence damages the *controllability* of the performance testing.

In 2012, Cucu-Grosjean et al. [26] shows a detailed statistical analysis for EVT application. Unfortunately, such a protocol process relies on the achievement of generally intractable *path coverage* assumption for its correct application. In addition, the authors employ a so-called *Exponential Tail Test* (ETT) which was meant to tell whether Gumbel distribution could be applied to the data or not. In 2015, Lesage et al. [20] develops a framework that computes a *ground truth* by calculating an actual WCET. With such a data the experimental framework checks the accuracy of the derived pWCET by using EVT. Moreover, this framework sheds light on how the lack of structural code coverage worsen this pWCET.

The same year Castillo et al. [83] delivers a case study with automotive benchmarks using the GPD distributions rather than Gumbel distribution which belongs to the GEV one. In 2016, Lima et al. [84] warns that time-randomized architectures are not enough to guarantee EVT application and additional statistical tests must be included to check for instance whether EVT can be applied to the data. In addition, they extend the application of EVT to the others GEV distributions which have different asymptotic properties [85]. This analytical consequence becomes apparent by Fedotova et al. [86] where for the same exceedance probability of 10^{-9} large differences are attained by using Gumbel and Frechet,

e.g., 7.7 ms vs 5900 ms [23]. In tandem with this observation, Lima [87] proves that resulting probabilities are very sensitive to the selection of extreme data for fitting. So as to enforce EVT application, Lima and Bate [68] formulate *Indirect Estimation in Statistical Time Analysis* (IESTA) which helps at the EVT application by padding the actual execution time observations and deriving a new distribution which is suitable for EVT application.

Considering the achieved research knowledge in EVT-based MBPTA, Gil et al. [72] outlines a list of challenges to be addressed. These challenges are split in three categories: 1) The execution time data resulting from testing that are supplied to EVT, 2) The statistical analysis of EVT itself 3) Interpretation of the EVT results and its relationship with the safety requirements of the overall system. The 1) challenge is a quite divergent one since it argues about the *representativeness* [23] of the execution time data which could entail to change the entire testing process. It's been claimed that to achieve *representativeness* the system must be tested as if deployed (also called *statistical* testing [36]). Unfortunately, this process often generates multiple Execution Time Profiles (ETPs) [23]. Another aspect is that representativeness does not necessarily imply statistically friendly distributions. In other words, it is not surprising that the data do not fit any statistical model due to its intrinsic shape or discreteness [80]. Disconcertingly, a spread idea amongst most of the MBPTA works [26, 20] is the calculation of exceedance probability in a fixed manner disregarding how much uncertainty i.e., difference between the HWM and WCET, we may have. In addition the calculation of exceedance is oblivious to the selection of maxima and the asymptotic properties of the distributions. Such a fixation hinders the possibility of using a MBPTA as a posterior statistical correction [72] and thus its usefulness is questioned.

Outside the realm of literature about the application of EVT in timing analysis, some others research works are not only stimulating but also potentially applicable. In 2000, Garrido [88] develops ETT whose actual objective is to give confidence in the Goodness-of-Fit GoF of the tail of distributions where there are no observations. The motivation behind this test is to give confidence in

structural reliability analysis where the predictions about the integrity of bridges had to be achieved by having a small number of samples (case studies are usually around 150 observations [24]). The main interesting feature of this test is that it endows with extreme events prediction power to non-EVT parametric distributions that meet certain assumptions. Hence, the entire sample rather than some few extreme observations ones are used to fit a parametric distribution. The check is done against an underlying EVT distribution whose tail predictions are compared against the parametric one. The first version of ETT requires to fulfill an assumption that enable the fit of the Gumbel distributions as an underlying EVT distribution to check the results.

Later on, Diebolt et al. [24] relaxes such an assumption and extend the analysis to the rest of the GEV distributions with the so-called *Generalized-Pareto Distribution Test* (GPDT). As a consequence, more parametric distributions can be fed to extreme events predictions. Another feature of this probabilistic approach is that, even though it is not oblivious to the selection of maxima, by using parametric distributions simulations can be carried out in order to derive a look-up table to chose a sound number of excesses [88, 24]. This piece of data depends on the employed distribution and the number of observations. Therefore, a tail test based MBPTA may achieve full automation which is actually one of the challenges of EVT-based MBPTA [72, 23].

Deciding which fitting method gives the best results and verify its implementation is another challenge in MBPTA [72, 23]. In this respect, a paper worthy of mention is submitted by Friederichs and Thorarinsdottir [75] in 2012 in which EVT distributions are fitted using Simulated Annealing algorithm maximizing the p-value of the GoF tests using data from winds. After running the experiments they observe the low variation of the shape parameter of the EVT which means that the asymptotic properties shouldn't vary much. The Search-Based Fitting (SBF) process is sometimes referred to as *maximum Goodness-of-Fit* estimation [89]. It is yet to be investigated how SBF may make a difference at meeting the prerequisites of tail tests as well as how to satisfy themselves.

2.5 Summary and Research Contributions

This chapter has surveyed a relevant part of the WCET the literature. STA stood out as the initial approach for WCET analysis but it seems that modeling modern hardware has set the limit of its application. Some of its techniques have been applied to Hybrid analysis in its attempt to predict a CWCET coming from an hypothetical path coverage [35]. Nonetheless, this estimation may be pessimistic due to the inclusion of infeasible paths, or in complex hardware architectures, the combination of the execution states predicted in the CWCET may also be unrealistic.

As MBTA circumvent the issue of modeling and achieves portability this approach has engrossed us. For this method to be effective exhaustive testing must be performed thereby automatic test generation is sought. There are only two central approaches for MBTA: I) **SBT** which is relatively simple to implement but occasionally struggles to achieve the same coverage as CBT. In particular, it shows worse performance when the satisfaction of one or more predicates depend only on very specific input of the search space [62]. Furthermore, to achieve good guidance the code need to be instrumented [16, 18] requiring an instrumentation tool and normally generating overhead in the SUT [28] II) **CBT** is deemed effective for the code coverage objective but needs a precise constraint-based model to deliver results.

Hybrid test generation combining both of the above techniques has been applied for software testing [63, 62] but not many works have been produced for MBTA [25].

So far, a strict CBT has never been applied to WCET and the state of the art CBTG applied to MBTA is designed for functional testing and has the following limitations [39, 40]:

- ☞ It is not concerned with the notion of path and thus it is not generally possible to identify the WCET path by merely delivering branch coverage.

- ☞ Important objectives such as maximizing the loop iterations are not considered since the objective is hit a loop at least once.
- ☞ The derived graph of the SUT collects unnecessary statements and consequently damages the performance of the search strategies to build path constraints.
- ☞ The search strategy applied is DFS to achieve branch coverage. Yet, by applying BFS the most promising paths could be composed first.

From these premises our **research contributions** on test generators can be split into a) CBT on its own:

1. To deliver a path composition algorithm based upon BFS that builds paths to be analyzed by using a CBTG. These paths not only lead to the largest execution times but these are computed first.
2. To evaluate the accuracy of the guidance given to the BFS in the form of cost.
3. To devise an *optimal* program slicing for MBTA which eliminates the inefficiencies of current approaches reducing the number of paths to analyze.
4. To evaluate the effects of the slicing in the run-time of a CBTG and discuss the impact on the test generation.
5. To show how this test process can be used to reduce the pessimism if an hypothetical CWCET is composed by infeasible paths by detecting such paths.

b) CBT in relation to SBT and RT.

6. To evaluate the results and performance of the proposed CBTG process against state-of-the-art SBTG and a Random Test Generator (RTG) where the main objectives are to maximize the HWM and attain results promptly.

As for the assumptions, most importantly the test generation is restricted to integer, floating and fixed points and enumerates which is claimed to be reasonable assumption for a large number of Real-Time Systems [16]. On the technical side, it is assumed that the source code of the SUT is available and there is no instrumentation or any other path-based composition process to derive a CWCET.

On the other hand, EVT has been the cornerstone of MBPTA in the recent years. Perhaps the main upside of this approach is to justify a safety margin by using a probabilistic argument when no other way of calculating a CWCET is feasible. Despite so, the lack of code coverage has already been reported to damage the pWCET results [20]. However, when looking into the details of EVT and its deployment within the context of MBPTA two main issues are worth investigating, namely, execution time data and EVT analysis itself. By approaching the execution time data problem from the testing point of view, three option arise so as to supply useful data to EVT [23].

- ☞ To test the system as if deployed in which case several distributions could be possible but *representativeness* of the data would be achieved [23].
- ☞ To generate a distribution such that the derived maxima data for fitting optimizes EVT distribution fitting [84].
- ☞ To test the system in a way to maximize the HWM so as to verify extreme events predictions of EVT using *scoring rules* [75].

From these 3 possibilities we have chosen last one for our CBTG because it provides with the greatest generality for the industry which may or may not be interested in probabilistic analysis. On the MBPTA side, the following issues have become apparent:

- ☞ Because of the maximal observation decision, EVT-based MBPTA approach struggles to be automated. This objective is important for the industry if EVT is to be embraced.
- ☞ The extrapolations of the EVT tails not only have been computed disregarding how much uncertainty there exists after the HWM but also neglecting the asymptotic properties of the tails.

- ☞ Though tail tests embrace EVT underneath, the selection of maxima can be automated.
- ☞ Search-Based Fitting (SBF) is a promising fitting method and it may be helpful at meeting tail tests assumptions.

From these issues, we draw our attention to tail tests and SBF and the following **research contributions** are to be provided:

7. To formulate a novel tail-tests-based MBPTA advocated to provide full automation and calculate tighter pWCETs.
8. To evaluate the applicability of such an analysis for the resulting *unbiased* execution time data from different test generators as well as figuring out what are reasonable exceedance probabilities using the HWM data representing *known uncertainty*.
9. To assess how SBF may help at the application of tail-tests based MBPTA.

Chapter 3

Constraint-Based Testing for Measurement-Based Timing Analysis

With the purpose of understanding a pathological case for SBT which CBT could analyze easily consider Listing 3.1. This code is advocated to represent *fault accommodation code* of industrial software where few pathological inputs trigger a checking list implemented by one or several loops. The main branch of this program would be triggered in so far as *sensor_fault* flag is activated. This in turn depends on very specific values of the three input variables. Assuming the input data takes a long range, say, $[-10^4, 10^4]$ the satisfaction of the predicate could be a burden for SBT and thus the central loop - which obviously has a big impact in the execution time - may not be executed or may take a very long time.

```
1 function check_initial_input (Signal1 : in Input_Signal; Signal2 :
    in Input_Signal; Signal3 : in Input_Signal) return Boolean
2    is
3       sensors_fault : Boolean := Signal1 - Signal2 = 10 and Signal1
    - Signal3 = 20 and Signal2 - Signal3 = 10;
4    begin
5
6       if sensors_fault then
7          for I in Array_Range'Range loop
8             Sensors_Vector (I) := RESET;
```

```
9           end loop;
10        end if;
11
12        return sensors_fault;
13    end check_initial_input;
```

Listing 3.1: *Needle in a haystack* pathological case where SBT would struggle at hitting the only the *if* decision in the code.

The underlying reason is that state-of-the-art approaches in the context of SBT [18] map the objectives of the test generator into different optimization functions to guide the search algorithm. The result of this methodology is the exploration of a *landscape* whose shape is determined by the guidance provided. Hence, this guidance is critical to generate appropriate test vectors. In this respect, as mentioned in the previous chapter, some testability transformations [58] have given some evidence about the coverage achievement. However, on the efficiency side, CBT could potentially make a big difference without any elaborate transformation of the code.

From the CBT perspective, the existing *knowledge* of the source code e.g, branch predicates, loop iterations, is not only required but also it could be used to improve the test generation results [63]. Therefore, it would be interesting to endow with this knowledge other forms of searches e.g., CSP, graph exploration algorithms, and to evaluate the results.

Returning to the needle in a haystack example, CBT would formulate this benchmark as a CSP by identifying the input data and its domain along with the constraints to be met. A sketch of the plausible representation is depicted in Listing 3.2.

```
1 Input data:
2       -10000 <= Signal1 <= 10000
3       -10000 <= Signal2 <= 10000
4       -10000 <= Signal3 <= 10000
5
6 Constraints:
7        Signal1 - Signal2 = 10
```

```
8       Signal1 - Signal3 = 20
9       Signal2 - Signal3 = 10
10
11  /*Plausible output of a constraint solver:
12      {9980, 9990, 10000};              */
```

Listing 3.2: CSP Listing 3.1.

The above description of a CSP could be provided to a constraint solver, which would determine whether there exists one or more feasible solutions, and if so it would print them. In this case a plausible solution to assign to our test vector would be $O = \langle\{9980, 9990, 10000\}\rangle$. The analysis of this code is trivial given that there are only two paths from which the WCET path is obvious. However, this test vector on its own would not achieve path coverage as the negation of the controlling variable must also be encompassed. It is fair to say that with this approach, the CSP is oblivious to what the most promising paths are from a WCET analysis perspective.

State-of-the art approaches in CBT for MBTA using FShell [40, 32] would achieve branch coverage by means of DFS with no regard on how each branch contributes to the execution time either. Yet, by including some **path heuristics** [38] such as BFS, we could point out that the first branch has a greater *cost* than the other one would be able to pick this branch in the first place. The price to pay for this advantage is to define a sound *cost*.

Another downside of current approaches would become apparent in the derived graph i.e., CFA, the block in line 8 would be on the edges and the search strategy would explore it even though it does not have an impact on the traversed branches of this software. The resulting challenge here would be to conceive an optimal program slicing. A truth that must not be shirked is that an important objective in the MBTA works where FShell was deployed [32, 67] is to maximize local HWMs collected in different points in the code. Next, these observations serve to build a global CWCET which is calculated by using the path composition IPET. By contrast, one of our objectives is to maximize the global HWM so as to give confidence in probabilistic approaches to derive a CWCET in the form of

pWCET.

Admittedly, the collection of constraints from the program analysis of this benchmark is simple, however, real-world software is generally more complex than this program. To be able to provide some automation to more examples as well as evaluate our research contributions we have implemented GenI, a test generation framework.

The research contributions of this chapter are outlined next:

(a) Contributions 1, 2, 3 and 4 from Section 2.5 which proposes **path-composition** heuristic based upon BFS and an optimal program slicing with their appropriate evaluations.

(b) Contribution 5 from Section 2.5 with four case of studies which are representative of Real-Time Software. The purpose of this evaluation is a) to evaluate of the effect of the slicing on the run-time of the CBTG and b) to assess the performance of the different test generators. The key assumptions here is that path coverage can be achieved and the constraint collection values can be collected statically.

The rest of the chapter is organized as follows: Section 3.1 addresses contribution 3 since it is the first step before applying contribution 1 which are explained in Section 3.2. contribution 2 is offered in Section 3.2.4 as well as in each case study. The *GenI* framework details for reproducibility are presented in Section 3.3. The effectiveness of contribution 4 is discussed in Subsection 3.1.4. Next, Section 3.4 presents the four case studies for contribution 5 which include data for the contributions 2 and 4. Lastly, a summary of the chapter is given in Section 3.5.

3.1 Optimal Program Slicing for Constraint-Based Testing

The notion of *testability* of a piece of software is relevant in the context of software as it anticipates the degree of difficulty of testing. Two implicit properties of the

testability are [19, 6]:

- *Observability*, which is regarded as the property of the software that allows examination of its inputs, internal variables, and outputs in order to decide whether a test passes or fails.
- *Controllability*, that determines how much the software output is driven by the input data.

Whereas observability is often achieved by debuggers, instrumentation tools or health monitoring systems [9], *controllability* is particularly relevant for CBT as we aim for identifying test vectors to achieve a great degree of coverage. Nonetheless, certain pieces of software or software constructs may not possess such a property. For example, variables declared in a program whose values are independent of the input variables e.g., constants, variables computed in a loop, are examples of non controllability. The presence of infeasible paths could also hinder the controllability as it would be hard to hit certain branches.

By contrast, certain statements in programs may not have a bearing on the flow of the program or give no relevant data for the constraints. Therefore, from CSP formulation these statements are *redundant*. To give an example of controllable and redundant statements consider the program in Algorithm 1.

Algorithm 1 Example of controllability and redundancy

```
1: procedure FOO(a)
2:     k ← 10          ▷uncontrollable statement but necessary to collect
3:     b ← a + 3                                  ▷redundant statement
4:     c ← a * 2       ▷relevant statement used in the constraint
5:     if c = k then                              ▷Constraint collection
6:         c ← ∅                                  ▷redundant statement
7:     end if
8:     c ← c + 1                                  ▷redundant statement
9: end procedure
```

A program slicing to apply CBT would collect the constraint $c_1 = \langle c = k \rangle$ and the input variable a. However the c and k are unknown variables for the CSP formulation. For this reason, we would need to collect the statement at line 2

even though it is not controllable along with the statement at number 4 because it is controllable and is the only way to exercise the constraint. The rest of the statements would be redundant as they do not have any impact on the constraint.

It is worth noting that program slicing may share common objectives with **irrelevant constraint elimination** [38]. While the latest may use constraint solver to detect redundant statement, program slicing does not necessarily computes feasibility check. Some others state-of-the-art approaches such as **amorphous slicing** [58, 59] aim for integrating in a single *flag* variable all the former variables controlling such a flag. However, to our understanding, there is not guarantee that the folded expression does not contain non-controllable variables.

Next subsection proposes some heuristics to be used for program slicing so as to collect the minimum data for a complete CSP description.

3.1.1 Program Slicing Heuristics

Before introducing the heuristics for program slicing it is worth clarifying some simple terminology often used in software testing. More accurately the notion of *definition* when a variable v is written e.g., $v = 3$ and *use* when a variable is read e.g., v is read in $x = v + 1$.

To systematically collect *the minimum constraints* of the SUT by using program slicing two central heuristics, $H1$ and $H2$, must be used: $H1$ aims for identifying the input variables, their domain and the constraints to collect. To build the constraints, $H1$ collects predicates which controls the *flow* of the program. These predicates entails, decisions in *if-else* and *switch-case* constructs as well as loop guards. On its own, $H1$ may be incomplete as it may allude to variables that are not in the input, I.

To fill this gap and connect the input variables with variables in the constraints $H2$ collects intermediate variable definitions, V, by discriminating whether a variable defined in these definitions controls subsequent constraints or they are

redundant. More formally, to understand the function of $H2$ two definitions are established.

Definition 3.1.1. *Be V the set of variables declared in a program $\mathcal{P}$, a definition of variable v_i, $Def(v_i \mid v_i, \dots v_n)$ uses variables $v_i, \dots v_n \in \{V \cup I\}$ where V are the variables declared in the program and I the set of variables from the test vector. A definition of a variable v_i, $Def(v_i \mid v_i, \dots v_n)$ is said to be **controllable**, $Cont(Def(v_i \mid v_i, \dots v_n))$ if it uses $\{I' \mid I' \subseteq I \wedge |I'| > 0\}$ or uses $\{V' \mid V' \subseteq V \wedge \{\exists\, v_j \in V',\ Def(v_j \mid I')\}\}$. Hence, $Def(v_i)$ is controllable when uses $I' \cup V'$: $Cont(Def(v_i \mid I' \cup V'))$*

Definition 3.1.2. *Be $Vc_i \in \{V \cup I\}$ the variables used in a constraint $c_i \in C$ located at a program point q_i, and be $Def(Vc_i)$ definitions of the variables in Vc_i. The heuristic, **H2**, collects $\forall c_i \in C$ and $Vc_i \in c_i$ up until point q_i the definitions which are controllable, $Cont(Def(Vc_i))$, or constants.*

A simple algorithm of this optimal program slicing would be as follows:

Algorithm 2 Optimal Program Slicing for CBT

```
1: function PROGRAM_SLICE(P)
2:                                                    ▷H1 application
3:     ⟨I, D⟩ ← CollectInputDataAndDomain(P)
4:     C ← CollectFlowStatements(P)
5:
6:     D ← H2(I, C, P)          ▷H2 application collecting relevant definitions
7:     C ← C - non-controllable(C, D)      ▷remove non-controllable constraints
8:     return ⟨I, C, D⟩
9: end function
```

Algorithm 2 describes the program slicing process. The first two statements collect in essence input data and the constraints. After that, $H2$ collects only relevant definitions for the constraints which link input data and the constraints. Lastly, in line 7, if there is any non-controllable constraint it is removed. The result is the program slice of $\mathcal{P}$ whose description is optimal to apply CBT with path coverage criterion.

With this algorithm equipped with $H1$ and $H2$ the minimum data to elaborate CSPs can be attained. However, a graph representation is still needed. This new graph-based representation is the result of the program slicing. It is worth questioning what difference does it make with respect current approaches [39]. This answer is answered from a theoretical point of view in the next two subsections. An empirical analysis comparing the effect with and without slicing is offered, wherever possible, in each case of study of this thesis.

3.1.2 Reducing Graph Complexity

Even though the program slicing has substantially synthesized the SUT we still need to build a representative model of the SUT so as to derive consistent path constraints. A good representation that enable search algorithm to be performed is a graph. State-of-the-art approaches [39] also build a graph after parsing the source code.
In the first place, to be able to understand our research contribution the notion of a **Constraint-Graph** (CG) must be defined.

Definition 3.1.3. *A CG of a program $\mathcal{P}$ is a tuple $\langle G, s_0, \delta \rangle$ where G is an acyclic graph $G = (V, E)$ with vertices V and edges $E \subseteq \{V \times V\}$. δ is the label set such that $\delta \subseteq \{V \cup E\}$. It holds the constraints, C, and definitions, D, derived by the optimal program slicing.*

To prove the difference between a CFA and a CG consider the following definitions: Given a program $\mathcal{P}$ from which a CFA, G_{CFA}, and a CG G_{CG} are derived. G_{CFA} holds *all* the statements from $\mathcal{P}$ whereas G_{CG} will only store the *minimum* to trigger the constraints because of the optimal program slice. To meet a certain coverage criteria a search must be launched inexorably. Thus, assuming the same search strategy is launched on the resulting graphs G corresponding to the exploration of G_{CFA} and G' from G_{CG} would become apparent.

A common practice to measure search strategies performance [27] is to evaluate the graph complexity of the graph they deploy. The three main variables to measure the complexity of a search are:

- **b**: **Branching factor** or the *maximum* number of successors of *any* arbitrary node.
- **d**: The **depth** of the most superficial goal node.
- **m**: The **maximum length** of any path in the state space.

The last two metrics, d and m, consider the path length. An arbitrary path length m_G in this exploration, *assuming that it contains redundant statements*, will be composed by some redundant statements r plus the minimum ones k. Hence the $m_G = r + k$ whereas the same exploring G' will only be composed by k, so $m_{G'} = k$. By operating we get:

$$\frac{m_G}{m'_G} = \frac{k+r}{k} = 1 + \frac{r}{k} \tag{3.1}$$

Hence, the length of any arbitrary path in the proposed CG will be:

$$\boxed{m_{G'} = \frac{m_G}{1 + \frac{r}{k}}}$$

In reference to the branching factor the G and G' may also be different *if the SUT contains uncontrollable flow statements*, r_b, because of the inefficient program analysis applied in CFA. The branch reduction would have a similar reduction factor between $b_{G'}$, the branching factor of the G' and b_G the branching factor of the CFA assuming r_b redundant branches and k_b minimum controllable ones.

$$\boxed{b_{G'} = \frac{b_G}{1 + \frac{r_b}{k_b}}}$$

These performance differences become apparent when the assumptions are met and we run search strategies on top of the CG and CFA graphs. As mentioned before, *FShell* runs DFS on top of the CFA [39] to achieve *branch coverage*. Such an algorithm has a complexity $O(b^m)$ in time and $O(b \cdot m)$ in space [27]. Hence when $r > 0$ or $r_b > 0$ the time and space of the search will be different and smaller for the CG' however the required data to transform deliver a CSP will be the same.

3.1.3 Graph Comparison Example

In order to better understand the application of the former theory this section is in charged with presenting an example. Firstly, a relevant function from a toy autopilot software [15] is read. It is worthy of note that this code is acyclic so the exploration graph of CFA and CG is very similar to the actual CFA and CG. Firstly, consider graphs in Figure 3.3.

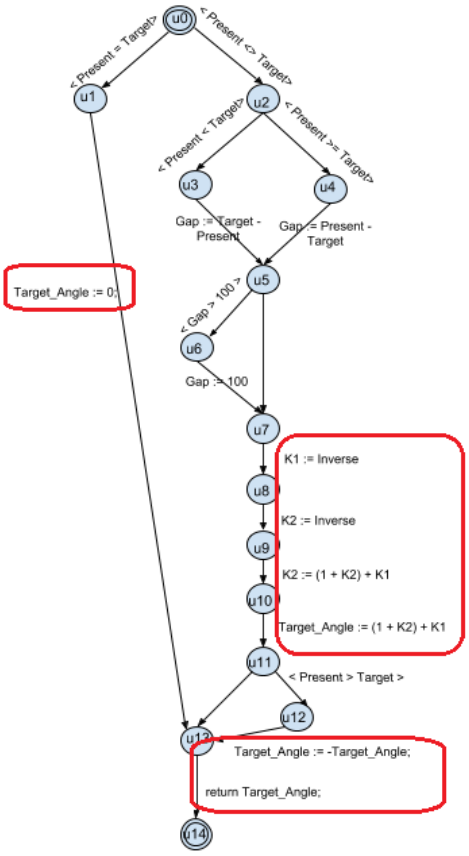


Figure 3.1: CFA example.

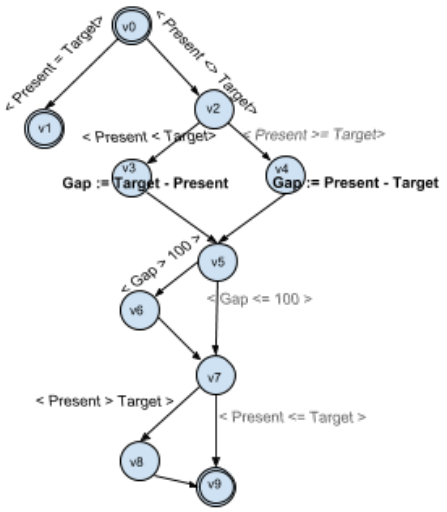


Figure 3.2: CG example

Figure 3.3: Comparison of graph models.

Figure 3.1 displays the resulting graph after analyzing the SUT by using the notion of CFA. On the other hand, Figure 3.2 depicts the resulting CG. By definition, the CFA inferred by FShell program analyzer collects all statements yielding a more complex graph. The statements surrounded by the red shapes

on graph 3.1 are redundant to trigger subsequent branches but the CFA includes them though. Unlike CFA, the CG computed only collects the minimum statements required to exercise paths.

To give some data consider the shortest path in the CFA of Figure 3.1: $\pi_{s,G} = \{u_0, u_1, u_{13}, u_{14}\}$ having just two vertices $\{u_0, u_1\}, k = 2$ that are really needed to execute the first branch (*Present = Target*). However, to reach the end point it needs to traverse $\{u_{13}, u_{14}\}$ so $r = 2$. Hence, $m_{s,G} = k + r = 4$. On the other hand the minimum path only takes two vertices: $\pi_{s,G'} = \{v_0, v_1\}$ for the CG on Figure 3.2 yielding to $m_{s,G'} = k = 2$. Replacing this number in the speed-up equation of Section 3.1.2 we get:

$$m_{s,G'} = \frac{m_{s,G}}{1 + \frac{2}{2}} = 0.5 \cdot m_{s,G} \tag{3.2}$$

So the shortest path of the CG is the half of the shortest path of CFA. Lastly, if consider one of the longest path in both examples we get: $\pi_{l,G} = \{u_0, u_2, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}, u_{11}, u_{12}, u_{13}, u_{14}\}$ where $k = 8$, $\{u_0, u_2, u_4, u_5, u_6, u_{11}, u_{12}, u_{13}\}$ and $r = 5$. On the other hand, $\pi_{l,G'} = \{v_0, v_2, v_4, v_5, v_6, v_7, v_8, u_9\}$ and again $k = 8$. By proceeding in the same way as before:

$$m_{l,G'} = \frac{m_{l,G}}{1 + \frac{5}{8}} = \frac{8}{13} \cdot m_{l,G} \tag{3.3}$$

Because this software does not contain uncontrollable decisions the branching factor is similar. This sets an example on how CFA representation is not optimal for CBT application as well as how the CG achieves an optimal representation of the SUT.

3.1.4 Effect of the Slicing on the Effectiveness of a CBTG

The effectiveness of a CBTG is understood as its ability to produce sound test vectors. Regarding the impact of the slicing on a CBTG it is worth remembering the entire purpose of the slicing is to *clear* the original program so as to build a CSP that is more likely to be feasible and thus to generate a test vector. Furthermore, it also aims for minimizing the number of computations by reducing the number of constraints to be processed.

From a CSP point of view, there is no distinction between uncontrollable predicates or redundant statements as they are both normally added as constraints. The effects of including all the statements of the default program depends on the constraint solver applied or the processing of the paths-to-constraints program.

In our framework, if no slicing were applied, two main scenarios would become apparent:

- The effect of the slicing has no effect and thus the default program would match the sliced one.
- More constraints would be added to the problem. This alternative would entail:
    1. In the CBTG from GenI an exception is launched as there are some declared variables in the constraints that are not mapped in the input data.
    2. In the constraint solver employed, SCIP [90], default range of variables is $[0, \infty)$, (∞ is encoded as the largest possible data type of SCIP variables) which may not match the variables declared along program execution. If the constraints provided intermediate variables i.e., declared variables which are not the global input data, and the constraints allow a feasible solution in the default range, then a feasible solution is returned. Otherwise, the problem is ticked as infeasible. This is demonstrated in the experiment below.

3. Even though the actual range of the uncontrollable or redundant variables was given, when assigning the resulting values to test vectors, run time debugging errors could pop up. That is why those variables SCIP provides are obviously not in the context or their range overflows the declared range in the program.

In order to evaluate the default values of SCIP to undefined variables, we are creating two CSPs one which contains a redundant variable, `t_0`, whose constraint is consistent with the default value (Listing 3.3) and another one (Listing 3.4) in which its constraint is inconsistent with the default range.

```
1  Minimize
2  0
3  Subject To
4   c1: i_0 + i_1 > 11
5   c2: i_0 - 2 i_1 > -1
6   c3: t_0 > 0
7  Bounds
8   0 <= i_0 <= 1000
9   0 <= i_1 <= 1000
10 End
11
12 solution status: optimal solution found
13 objective value:        0
14 i_0                     1000     (obj:0)
15 i_1                     500      (obj:0)
16 t_0                     100000   (obj:0)
17
```

Listing 3.3: CSP with an undefined variable in a constraint.

As described on Listing 3.3, SCIP makes the range assumption described above and prints a feasible solution. On the contrary, if we impose to `t_0` a negative value as portrayed in Listing 3.4, the value of `t_0` is not printed, yet the defined ones do.

```
1      Minimize
2      0
3      Subject To
```

```
4         c1: i_0 + i_1 > 11
5         c2: i_0 - 2 i_1 > -1
6         c3: t_0 < 0
7       Bounds
8         0 <= i_0 <= 1000
9         0 <= i_1 <= 1000
10      End
11
12      solution status: optimal solution found
13      objective value:        0
14      i_0                     1000      (obj:0)
15      i_1                     500       (obj:0)
16
```

Listing 3.4: Another CSP with an undefined variable in a constraint.

This experiment has demonstrated that the specified range of undeclared variables in SCIP, which are equivalent to uncontrollable or redundant statements, matches the observed one. As a consequence of this default setting, the resulting value of these variables may be unsound as the default range is unlikely to match the actual range declared in the program.

3.2 Path Construction Using Best-First Search

3.2.1 Path Coverage Complexity

For a constraint solver to generate test vectors for a path all the defining constraints of the path must be included. Adding constraints manually is normally unmanageably complex. Let alone, the path explosion because of the combinatorics of **path coverage**. To provide insight about the complexities of the former, some works on MBTA [17] contend that the number of paths increase exponentially with the software size. Notwithstanding, the path explosion combinatorics is more studied by Bang et al. [53].

The conclusion of this work is shown in Figure 3.4. The path explosion is generally exponential in a sequence (K) of conditional structures, a sequence of loops and nested loops. The only exception, is when we have a sequence of nested con-

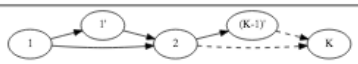
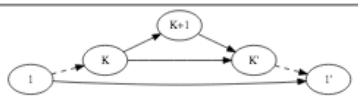
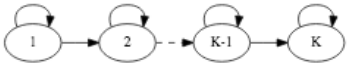
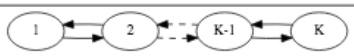
Pattern	Control Flow Graph	Cyclomatic Complexity	NPATH Complexity	Asymptotic Path Complexity
K If-Else in sequence		$K+1$	2^K	2^K
K If-Else nested		$K+1$	$K+1$	$K+1$
K Loop in sequence		$K+1$	2^K	$\Theta(n^K)$
K Loop nested		$K+1$	$K+1$	$\Theta(b^n)$

Figure 3.4: Path complexity depending on the underlying program structure. Source [53, Table 2]. K denotes either the number of elements in a sequence or the depth of nesting, n designates the depth bound and b is a constant number which depends on K.

ditional statements in which case the path explosion is linear. Yet, if we examine the third case with a sequence of loops and K as a constant and changing n - the depth of the loop bounds - the path explosion would endure a polynomial explosion. Bearing in mind these restriction the modelling of path coverage is generally *intractable* and therefore challenging.

3.2.2 Best-First Search

So far we have a program slicing technique to collect the constraints. By reading this information and building a graph we need a way to systematically derive constraints which represents a path. There are three challenges to consider here.

1. To define a search strategy to build path constraints.
2. To be aware that in most cases path coverage can't be achieved and a partial path coverage is sought.
3. To give confidence that the search provides with the most promising paths first to meet the efficiency objective of the industry.

4. To evaluate the accuracy of the guidance i.e., cost, to evaluate the confidence of point 3.

The objective of identifying the paths leading to the largest execution time trades-off with our requirement of devising a portable approach since we are appealing to software for such a calculation. For this reason, the entire CBTG most probably give good results when the execution time is aligned with the guidance of the search strategy which is often hard to do by looking at the software only. Nevertheless, some evaluation is provided in this respect at subsection 3.2.4.

After reviewing search strategies theory [27] and graph exploration algorithms [47], BFS [27] emerges as a promising strategy since it seeks to explore the assumed most promising paths of an arbitrary graph in the first place. At the core, BFS is based upon holding a *priority queue* where the nodes are inserted considering a *cost estimation function*, $f(n)$, of the cost of exploring candidate nodes. The values of this function are also called *optimality hypothesis* [47]. Since the priority queue order depends on $f(n)$ it is instrumental that $f(n)$ computes either lower or upper bounds depending on the problem at hand. If the $f(n)$ results are *inconsistent* the search may be *misdirected*. Popular search algorithms like A* or Dijkstra [27] are based on BFS but unlike this problem their objective is to *minimize* the cost. BFS is often called *Branch and Bound* search [47] but its working principles are the same. This algorithm is depicted in Algorithm 3.

In Algorithm 3 the priority queue is named *frontier*. The front node of the queue is explored first and in the event that it is a final node, the search is concluded (lines 12 to 14). Otherwise, more children nodes are explored as long as they are not explored yet and they are not in the priority queue (lines 16 to 18). The estimated bounds may be updated as the search progresses (lines 19 to 21) for the sake of accuracy. With time the algorithm inserts and deletes node in *frontier* whereas the set of *explored* nodes is filled up rendering less nodes to explore.

To give an example about the execution consider Figure 3.5. On the left side a standard CG derived from an arbitrary SUT is displayed. Constraints are encoded with B+ and NB+. Each branch has a node with an attached cost but

Algorithm 3 Best First Algorithm. Source [27]

```
1: function BEST_FIRST_SEARCH(problem)
2:     node ← {problem.Initial_State, Path_Cost = 0}
3:     frontier ← ∅
4:     frontier.push(node)                          ▷Priority queue ordered by f(n)
5:     explored ← ∅                                 ▷Set of the explored nodes
6:     while true do
7:         if frontier = ∅ then                     ▷Queue is empty
8:             return failure
9:         end if
10:        node ← frontier.top()
11:        frontier.pop()                           ▷Remove the front of the queue
12:        if Goal_Achieved(node) then
13:            return node;
14:        end if
15:        explored.push(node)
16:        for all child ∈ node do
17:            if child ∉ {explored ∪ frontier} then
18:                frontier.push(child)
19:            else if child ∈ frontier and Child.Path_Cost is greater than
20:                the estimated in the queue then
21:                frontier.update_cost(child)      ▷Update the estimated cost with
   the actual cost
22:            end if
23:        end for
24:    end while
25: end function
```

B1 and NB1, which are assumed to be 0 for the sake of simplicity. On the right side, the maximum accumulated cost of each branch is computed. B1 cost results from B1.1 + NB1.2 = 17 + 7 = 24 whereas NB1 is B1.3 + NB1.4 = 10 + 10 = 20. On the first stage, the search would pick up B1 so the path to test would hold $\pi = \{B1\}$. On the second stage it would take B1.1 since NB1.1 offers only an accumulated cost of 22 in contrast to 24 offered by B1. So now $\pi = \{B1, B1.1\}$. The same comparison would be computed between B1.2 and NB1.2 selecting again the nodes leading to the greatest cost. Since NB1.2 is the most costly node and happens to be a leave i.e., goal node, the search would be concluded composing the path to test as $\pi = \{B1, B1.1, NB1.2\}$.

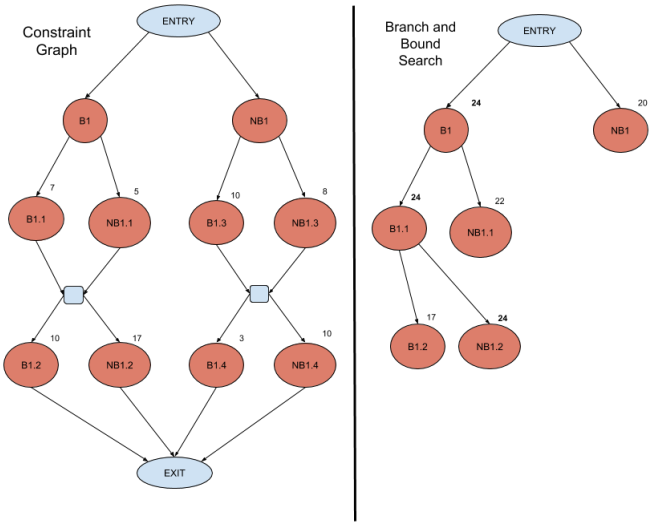


Figure 3.5: BB search guided by the costs of the constraints.

The core of this search strategy is the cost assignment of the constraints. Next section discusses this decision.

3.2.3 Constraint Cost Assignment

Ideally, the cost assigned to constraints should be proportional to the actual execution time which would entail to figure out how the idiosyncrasies of the compiled code as well as the clock cycles taken by each instruction. Let alone the impact of potential hardware performance acceleration units. Unfortunately, this relationship is hard to predict, if at all possible, by reading source code only. By assuming that a wide array of test vectors can be generated as a result of a great number of feasible paths, the accuracy of costs would result, to some extent not that important.

For the sake of providing with the simplest solution the cost is decided by counting the statements plus some overhead - typically 0.5 - of previous evaluation of the decisions if that is the case of the constraints at hand. It is important to remark that *this cost assignment is applied when the default program is analyzed* event though its final representation may be sliced. Therefore, the cost assignment is not impacted by program slicing.

In the case of function calls some overhead can be added on account of the instructions involved in the execution. 0.5 was added as well in the case studies. When it comes to loops the number of iterations can be really useful to determine their costs. In this respect STA works on counting loop statements [7, 14] are really useful but restrict the loops analysis to those which meet the Presburger arithmetic requirements. Such a requirement is one of the assumption of our contributions.

Lastly, it is worth noting that the proposed heuristic-driven path composition are independent of the *way* costs are assigned to constraints. In other words, this constraint-assignment decisions can subject to future improvement and both test generation and path construction delivered by BFS would still work.

3.2.4 Accuracy Evaluation of the Cost Function

This final subsection is dedicated to assess the precision of the cost function. To achieve this, the hypothesis *the cost derived from counting the statements generates statistically similar execution times across different programs* is established.

To evaluate this hypothesis we have used the execution time in clock cycles as a metric because it provides a great accuracy of the intended observed effect. Three types of benchmarks are employed because they arguably represent the three kind of programs we may find in the real world [91]. Programs containing arithmetic (Arith.) operations such as sums, subtraction and multiplication, load and store (L/S) such as assignment to variables or arrays and input output (I/O) to different modules such as watchdog timer or communications. It is worth saying that sometimes I/O operations are *memory mapped* [91] i.e., the system address space is shared. So the internal implementation of the L/S operations may be very similar to the I/O ones.

These benchmarks provide fair and representative examples of operations of Real-Time Systems as their statements were picked directly from the Real-Time software of the case studies. They are composed by a single block of code containing statements from the same above-mentioned operations. The number of statements for each benchmark is $[2, 4, 6, 8, 10]$ (which would match the same cost) as this is a reasonable range observed in these benchmarks. Additionally, as the execution time may be influenced by the input we establish an additional category for the benchmarks.

As for hardware platform the STM32F429 board equipped with STM32F429 microcontroller with 64-Mbit SDRAM with 1024 bytes of instruction cache memory and 128 bytes for a data cache. The processor installed in the microcontroller is a single-core ARM Cortex M4 [92]. In other words, it is a relatively simple and time predictable hardware with the burden of small cache for the WCET analysis. Memory-mapped I/O is implemented in the chip.

The benchmarks were run on top of a Spark/Ada run-time for ARM-based micro-controllers [93]. The availability of this free run-time motivated the choice of the board. Moreover, the decision of running the benchmarks on an actual hardware instead of a simulator is to provide a realistic setting to the experiments.

Regarding the results for the constant input, they are depicted in Figures 3.6 and 3.7. Figure 3.6 starts with a small difference of around 2.8 times between I/O and L/S in the cost 2 case. The difference becomes more significant from cost 4 to cost 10 where it reaches a maximum of 14 times greater again between I/O and L/S. This is because of the inclusion of more expensive I/O operations related to the communication modules e.g., Universal Asynchronous Receiver Transmitter (UART). An interesting feature of the constant-input benchmarks is the fact that the execution time shows no variability in each case which evidences the time-determinism of the hardware platform.

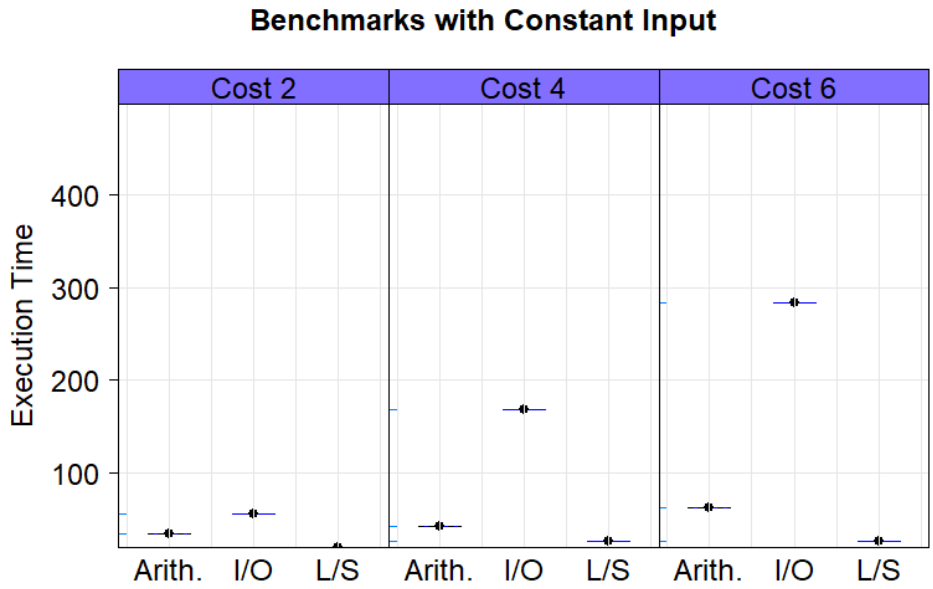


Figure 3.6: Block-based benchmarks ranging from from 2 to 6 with constant input.

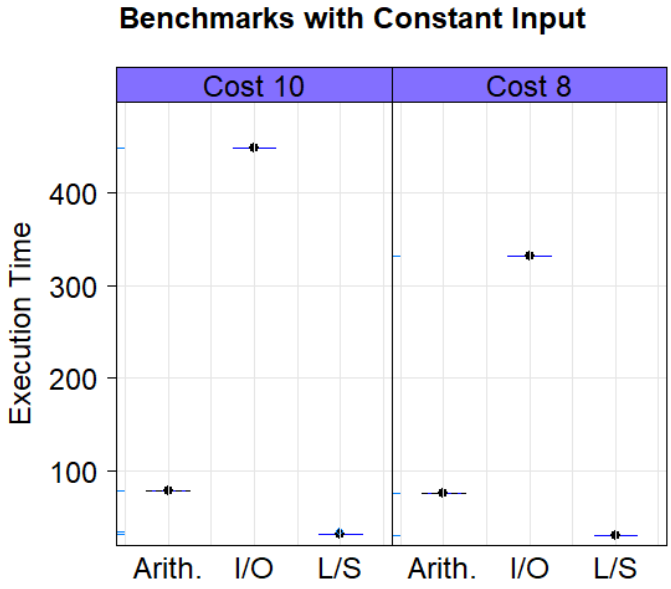


Figure 3.7: Block-based benchmarks ranging from from 10 to 8 with constant input.

On the other hand, the benchmarks which were provided a random test vector for each of their statements are depicted in Figures 3.8 and 3.9. In this case, the cost 2 case shows at slightly smaller greatest difference of approximately 2.63 times. Whereas, at the end of the experiments on cost 10 this difference is around 13.16 times greater.

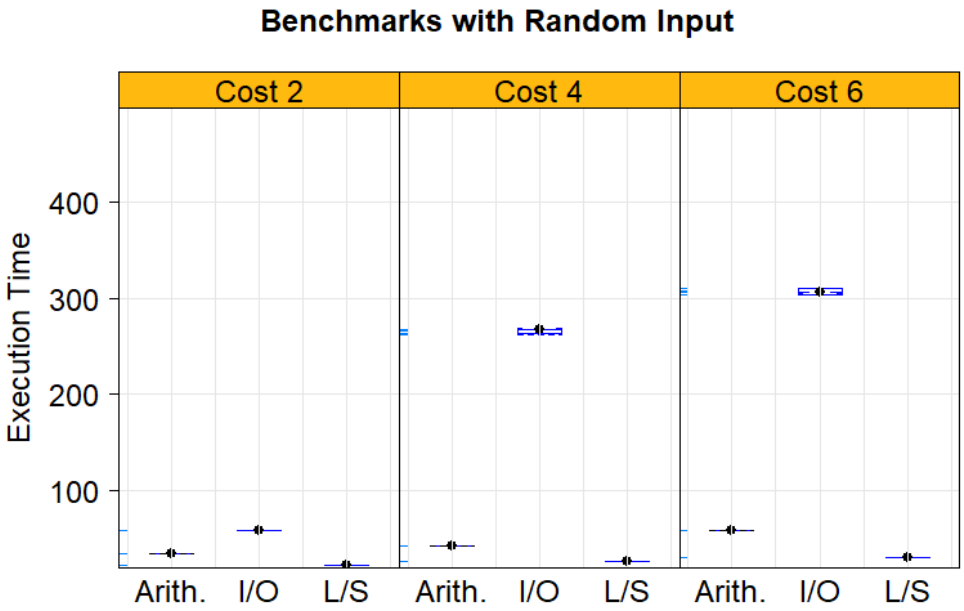


Figure 3.8: Block-based benchmarks ranging from from 2 to 6 with random input.

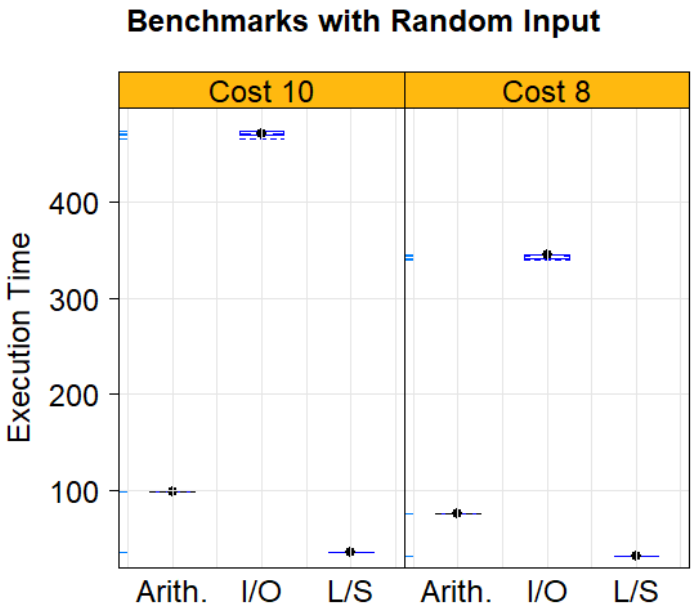


Figure 3.9: Block-based benchmarks ranging from from 10 to 8 with random input.

Perhaps the most interesting data analysis of this experiment is to evaluate the statistical significance of the Friedmann test. The reason for using this test is its ability to evaluate the degree of similarity or difference i.e., statistical significance, of all the empirical samples. This test is used to detect differences in treatments across multiple test attempts. The procedure involves ranking each row (or block) together, then considering the values of ranks by columns. *E.g., n wine judges each rate k different wines. Are any of the k wines ranked consistently higher or lower than the others?*.

The results of Friedmann test applied to the resulting data of the three type of benchmark are displayed on Table 3.1. In all the cases statistical significance is shown, meaning that the estimated cost and the actual execution time is significantly different. In conclusion, in light of the data of the block-based benchmarks *the formulated hypothesis at the beginning of this section is false.*

Lastly, it is important to discuss the threats to validity of the conclusion. In this

Cost	2	4	6	8	10
Random Input	Yes	Yes	Yes	Yes	Yes
Constant Input	Yes	Yes	Yes	Yes	Yes

Table 3.1: Friedmann test. Statistically significant with $\alpha = 0.05$.

work we consider two threats of validity inspired by some similar works on test generation [63].

- **Threats to internal validity** argues about potential bias in the design of the experiments. An example of such a bias comes from the kind of benchmarks employed which are restricted to the category of operations above described. In practice, some real-time benchmarks e.g., control systems [94], show a mix of the above operations. Another source of bias may come from the fact a Random Test Generator was employed whereas the cost is advocated to be used for the CBTG. The reason for this choice is to have a more realistic observation of execution time.

- **Threats to external validity** is deemed as arguing about how the conclusions can be extrapolated to more general cases i.e., different embedded architecture and different SUT. The most obvious threat is the computer architecture employed which is relatively time predictable one with memory mapped I/O. As we have extensively discussed, the execution time is highly sensitive to the computer architecture the software runs on. Hence, this conclusion may or may not be applicable to similar benchmarks running on another microcontrollers. Moreover, because we have not included benchmarks with branches and loops we could not observe how the execution time behaves in these cases. Nevertheless, the benchmarks presented in this thesis include such a data.

3.3 GenI Test Generation Framework

So far, the main theory of program slicing and path building by BFS has been outlined. To give some compelling evidence about the proposed CBTG an implementation was carried out in a tool named *GenI*. This framework's objective

is the cross comparison of different test generators for MBTA.

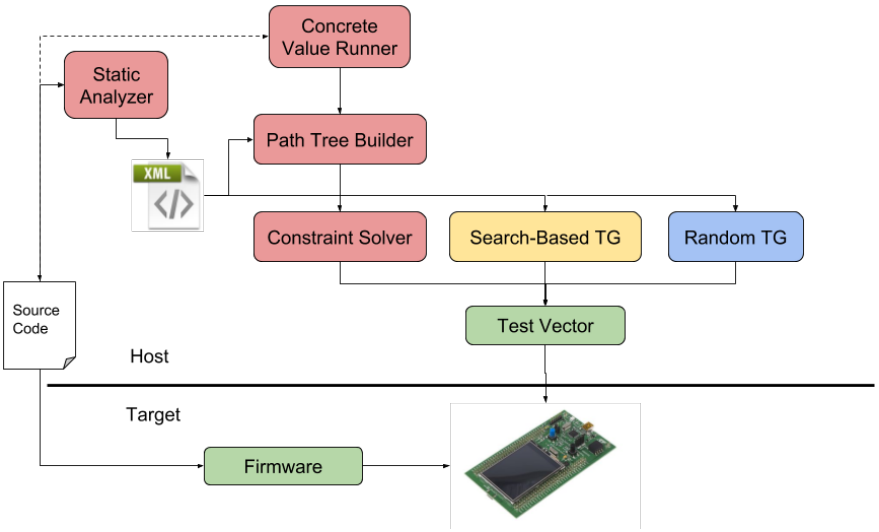


Figure 3.10: GenI code-driven test generator framework overview.

To begin with, Figure 3.10 displays an overview of the different components of *GenI*. Red color indicates the components of the CBTG, yellow component refers to the SBTG and blue to the random one. Additionally, green components denotes the elements interacting with the embedded target. The main input for the framework is a source code or SUT. As these test generators read only the source code and not a specification, this type of test generation is often called *code-driven* [95]. The first step consists of analyzing the SUT seeking the subprogram to test. This information is specified by the developer. Next, the static analyzer reads the structure of the subprogram SUT to gather constraints using the program slicing algorithms described in Section 3.1. Once this process is completed, all the relevant information is backed up into an XML file which contains the results of the program slicing.

For some programs where at least one of the required constraints is subject to values computed at run time, this static analyzer won't be able to collect all the required information to produce the constraints. That's why in a *concrete value* runner, which consists of an equivalent SUT, is executed in the host with random test vector as input. This feature will be tackled on the next chapter.

When all constraints are properly collected, along with data of their level of nesting a CG is built. The algorithms describing this process are outlined in Subsection 3.3.4. Then this CG is transformed in a tree from which the BFS is launched. The latter process is detailed in Subsection 3.3.5. Finally the encoded constraints of the path are processed and transformed to some understandable constraints by the constraint solver. Such an encoding and processing is explained in Subsection 3.3.2.

Regarding the constraint solver SCIP [90] was used. The underlying reason for using this constraint solver is that it is one of the most popular constraint solvers, it is free for academic purposes, it provides functionality to analyze a wide array of constraints: linear, logical, polynomials, etc... and it is easy to interface with our test generator framework. Finally, if the path is feasible a test vector is generated and executed on-target and thus an execution time is measured.

3.3.1 Applying BFS to our Problem Domain

Though the theory provided in the books may look simple the matter of the fact is that the implementation of BFS is certainly challenging [47]. In addition we must consider the specifics of our problem. For example, some heuristic problems using BFS finishes when a goal node is found. Notwithstanding more paths need to be analyzed to further new searches must be launched. Particularly, when integrating and implementing the theory in our problem domain the following problems become apparent.

1. To define a constraint notation.
2. Build a CG from a program description.
3. Build a path tree from previous CG.
4. Deciding an upper bound to drive the search: $f(n)$.

5. Handle cost consistency and node constructions when adding or deleting nodes.

Next section deals with problem 1. Problem 2 is tackled in Section 3.3.4 which is preceded by Section 3.3.3 explaining the involved data structures and addresses problem 5. Problem 3 and 4 are addressed in Section 3.3.5. Section 3.3.6 outlines how build a path by running BFS. Finally, Section 3.3.7 offers the details of the SBTG and RTG.

3.3.2 Constraint Encoding Notation

In order to compose a consistent CG from the description of the SUT not only the definitions of the constraints are needed but also the nesting level and the cost. To meet this demand a **Path Constraints Notation** (PCN) must be established Since the decision of cost has already been argued, the information regarding the structure of the code, including the level of nesting is tackled.

A *Constraint ID* has a single B to denote the first predicate of a structure. EB is employed to denote a predicate between the first and last decision of that structure, or NB to denote the last decision of such a structure (often related to *else* or *default* tokens in switch statements). Along with the alphabetical part, a number is used to denote its order with respect the root function and its nesting level. Assuming that root function has a 0 nesting level each constraint ID belonging to the first nesting level would be assigned to a natural number. If a nesting level is greater than 1 the dots notation often used in text processors is embraced. A problem with the proposed notation becomes apparent when several *elsif* or *switch-case* in the code as all of them would be named $EBX.Y$. For this reason a number *before* the alphabetical part is included. So the first *elsif* would be $1EBX.Y$ the second, $2EBX.Y$ and so on and so forth.

Once a path π is selected and is encoded as a sequence of constraint IDs, there are two more decisions to carry out: a) replace those IDs by the proper constraints and occasionally b) to process those constraints to be supplied in the proper way for a constraint solver. For instance, assume a path is encoded as

$\pi = \{NB1, B1.1, B1.2, \} = \{\textbf{Present} \neq \textbf{Target}, \text{Present} < \text{Target}, \text{Present} > 10\}$. To be able to hit *NB1* we need to negate B1 constraint. This requirement demands that the test generator implements a negation function of the operators that most programming languages exhibit.

Operator / Boolean Expression	Negated Operator / Expression
$=$	$\neq$
$<$	$\geq$
$>$	$\leq$
x	$\neg x$
$x \wedge y$	$\bar{x} \vee \bar{y}$
$x \vee y$	$\bar{x} \wedge \bar{y}$
$x \oplus y$	$x \equiv y$
$x \equiv y$	$x \oplus y$

Table 3.2: Table of operators and boolean expressions often found in programming languages. $\oplus$ denotes XOR operation and $\equiv$ XAND one. Some combinations omitted for simplicity.

Perhaps the simplest way to do that is just to look up a negation table like the one displayed at Table 3.2. Depending on the length of the conditional structure it may require to negate several constraints in the list. After having a negation function we show a simple algorithm to transform a path of IDs to a path of a list of constraints. Such an algorithm is displayed in Algorithm 4. The simplest case happens when the ID to transform is the first decision in a conditional structure since there is no further negation to consider in that structure. In this case, we only need to load the corresponding constraints of that branch. In the algorithm this is decided in line 4. The other cases are a bit more elaborated since they have to not only load the constraints of its decision in particular but also the negation of the previous decisions of the same conditional structure. In the latter case a list of constraints is returned. This is described from line 6 onwards. The function `negate` takes into account Table 3.2 definitions.
Lastly, regarding this processing to transform a path to a CSP, unless the SUT is quite simple we normally would encounter definitions of variables in the code. These intermediate variables are not input data nor constraints in the branches

Algorithm 4 Algorithm to transform a path defined as a list of IDs to a path of constraints. $\oplus$ stands for concatenation.

```
 1: function CONVERT_ID_TO_CONSTRAINTS(ID, ID_list)
 2:
 3:
 4:     if (ID_starts_with_B(ID)) then
 5:         return get_constraints(ID);
 6:     else
 7:
 8:         constraint_list ← get_constraints(ID)
 9:         previous_IDs ← get_previous_IDs(ID, ID_list)
10:
11:         for each ID' ∈ previous_IDs do
12:             constraint_list ← constraint_list ⊕
13:                         negate(get_constraints(ID' ))
14:         end for
15:
16:         return constraint_list;
17:     end if
18:
19: end function
```

and would be unrecognized by the solver. If the SUT is controllable they should depend on the input data. For this reason their definitions are collected. To be able to adapt the path constraint to a CSP one of the following two solutions must be given: a) Process the constraints so that all the variables that appear in the constraints depend upon the input variables or b) If the solver provides the proper functionality, these temporary variables can be added as some extra variables but when the constraints are solved there must be a way to filter the temporary variables from the input ones. Solution b) is probably the simplest and the best one as the other would entail a more elaborate programming.

3.3.3 Data Structures and Bounds Consistency

In order to understand the algorithm of the path composition we must define the data structures employed. The decision of choosing these data structures and their final state is founded on two objectives: 1) To access relevant nodes first

i.e., most costly 2) Manage easily the deletion or addition of leaves so as to keep consistency of the overall cost estimation.

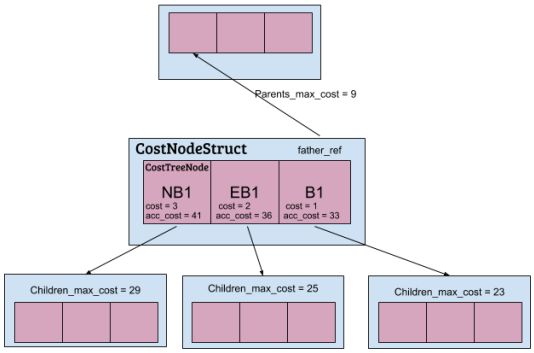


Figure 3.11: Simplified structure to implement BFS search

Figure 3.11 contains a diagram of the main structures implemented in the building of the path tree. To start with, *CostNodeStruct* holds a sorted vector containing the nodes data i.e., *cost_tree_node* (CTN). The fields of CTN will be explained in detail later. Each *CostNodeStruct* also holds a reference to a specific CTN that is supposed to be its father. Likewise, each node points to its proper children implemented in a *CostNodeStruct*. A codified and more extended definition of these structures is depicted in Algorithm 5.

The smallest data structures from which the CG or search tree is derived consists of the following. First one is *cost_tree_node* (CTN) structure, which holds a single constraint ID along with its *cost*, *children_max_cost*, *parents_max_cost* and *accumulated_cost* that are numeric and null by default. *Cost* field corresponds to the constraint cost. This data structure is embedded in a vector which in turn is contained in *CostNodeStruct* (CNS) structure. Such a vector must remain sorted to facilitate the access to the most costly nodes so the insertion operation must meet this requirement. Each of the parameters of CTN are mathematically defined as follows.

Be $CostNodeStruct_i$ in a level i of a tree, a structure be A_i the set containing the CNS ancestors of $CostNodeStruct_i$ and a_i the set of CTN ancestors to

Algorithm 5 Main data structures used for BB search

```
1: procedure TREE_NODES_DATA_TYPES
2:
3:     Structure cost_tree_node
4:     ID ← ∅                          ▷String type. It denotes the ID of the node
5:     parent_max_cost, cost, children_max_cost, accumulated_cost ← 0.0
6:     bb_father, bfs_child ← ∅                          ▷Reference to CostNodeStruct
7:     cg_child_vector ← ∅                ▷vector of references to CostNodeStruct
8:     end Structure
9:
10:    Structure CostNodeStruct
11:    node_vector ← ∅                                   ▷vector of cost_tree_node
12:    father_ref ← ∅          ▷Reference to cost_tree_node containing the father
13:    nodes_to_update ← ∅                          ▷vector of NodesToUpdateStruct
14:    end Structure
15:
16:    Structure NodesToUpdateStruct
17:    father_node ← ∅                                   ▷reference to father node
18:    ID ← ∅                                                    ▷ID of the father
19:    cfg_to_build_node ← ∅    ▷reference to the node in the CG from which to
   build BB tree
20:    end Structure
21:
22:    Structure NodeFatherToPick        ▷Structure used in the main priority
   queue of the BB algorithm
23:    ID ← ∅
24:    accumulated_code ← 0.0              ▷data to assign priority in the queue
25:    cost_node_ref ← ∅                          ▷Reference to CostNodeStruct
26:    end Structure
27: end procedure
```

$CostNodeStruct_i$, $parents\_max\_cost$ is defined as follows:

$$parents\_max\_cost_i = \sum_{j=i-1}^{1} max(\{n \in CostNodeStruct_j\}) \; ; \; \forall j \in A_i \wedge \forall n \in a_i$$

Be S_k the set of successors of an arbitrary non-empty CTN k stored in $CostNodeStruct_i$ and be m the depth of the tree.

$$children\_max\_cost_k = \sum_{j=i+1}^{m} max(\{n \in CostNodeStruct_j\}) \; ; \; \forall j \in S_k$$

From these definitions we have that an arbitrary CTN k with a local cost ($cost_k$) has an $accumulated\_cost\_k$:

$$accumulated\_cost_k = children\_max\_cost_k + cost_k + parents\_max\_cost_i$$

The data for the corner cases are $parents\_max\_cost_0 = 0$ since that is parents cost of the root and $children\_max\_cost_l = 0$ of any arbitrary leave $l <= m$ denoting the cost of non-existent children.

Resuming the structure description in Algorithm 5, apart from the cost data in CTN structure we also encounter an ID of the constraint in question, a reference to a CNS both for father and child along with a vector of children pointing to the relevant children to build from the CG. Aside *node_vector*, CNS contains *father_ref* which is a reference to the CTN father and *nodes_to_update*. Last one is a vector composed by the last structure in Algorithm 5. This vector is quite relevant when keeping the cost consistency when the a single or specific fathers need to be updated rather than all available in the node_vector of the father. This case may be deemed especial or pathological.

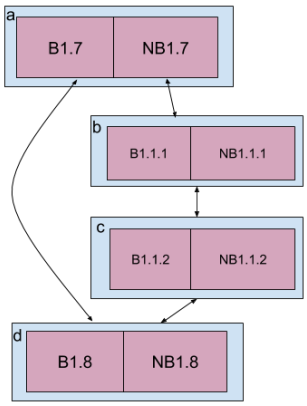


Figure 3.12: Pathological nesting case in a CG.

An example of such a pathological pattern is illustrated in Figure 3.12. In such a figure when structure d is added as a successor of c it needs to update *all* the nodes in c and b but when it reaches node a it only needs to update *NB1.7*. In reality, the reason why *NB1.7* would only be updated is that nodes in b and c would only be reached if *NB1.7* is triggered e.g., nested conditional. On the other hand, when structure d is assigned as a child of *B1.7* it only needs to update this CTN in particular. In summary, *nodes_to_update* is used when we need to update a single father node rather than the entire CNS.

NodesToUpdateStruct structure is dedicated to hold information of the only father to update as well as a reference in the last statement to build children if it was needed. *NodeFatherToPick* has a similar structure to *NodesToUpdateStruct* but it has a copy of the accumulated cost that is used for the priority queue order.

Regarding the algorithm to keep costs updated, it is listed in Algorithm 6. An important requirement to run this algorithm is that it should only be called from the leaves once these are added or deleted. This is because the update procedure runs from the leaves to the root. This procedure ought to access to the data in the CNS holding the leaves. First part of the algorithm spanning from line 3 to 8 discriminates whether the father of the node we are analyzing is the root or not since the root doesn't have parents whose cost to add. If it is not the case of a root (else from line 9 to 35) the algorithm discriminates whether that node has to update a single CTN by asking whether *nodes_to_update* is empty or not (line 9).

Finally, if it must update the entire father CNS, the else from lines 25 to 32 is taken. On account of the update in the accumulated cost it may occur that the sorted vector of the father is *inconsistent*, therefore a sort routine is called (line 33). This routine may trigger further updates of the nodes in the tree when the cost of the first element is different from the one before the sort was executed.

Algorithm 6 Algorithm to update costs. → denotes access to a data of the structure.

```
1: procedure UPDATE_ACCUMULATED_COST(new_cost)
2:     calling_node→node_vector.begin().ID        ▷Must exists a way to access node_vector and its links
3:     if father_node ≠ ∅ then
4:         if father_node→ID = root then
5:             father_node→children_max_cost ← new_cost
6:             father_node→accumulated_cost ← father_node→cost +
7:             father_node→children_max_cost
8:         else
9:             if nodes_to_update ≠ ∅ then
10:                for all item ∈ nodes_to_update do
11:                    if item.father_node = father_node then
12:                        if item.ID ≠ ∅ then
13:                            for all jt ∈ father_node→node_vector do
14:                                jt.children_max_cost ← new_cost
15:                                jt.accumulated_cost ← jt.parents_max_cost +
16:                                jt.cost + jt.children_max_cost
17:                                if item.cg_to_build_node ≠ ∅ then
18:                                    jt.cg_child_vector.push_back(item.cg_to_build_node)
19:                                end if
20:                            end for
21:                        end if
22:                    end if
23:                end for
24:            else
25:                for all jt ∈ father_node→node_vector do
26:                    if father_node→ID = jt.ID then
27:                        jt.children_max_cost ← new_cost
28:                        jt.accumulated_cost ← jt.parents_max_cost + jt.cost +
jt.children_max_cost
29:                    end if
30:                end for
31:            end if
32:            sort_node_vector(father_node)    ▷sort node vector of the father to pick best node first
33:        end if
34:    end if
```

35: **end procedure**

3.3.4 From SUT description to CG

Bearing in mind that our final objective is to build a tree to apply BFS search we need some algorithms to transforms the description of the SUT into such a tree. The reason why a CG *per se* is not a proper graph to explore stems from when we consider the nesting level of the SUT and the path building as a search. This issue is portrayed in Figures 3.13 and 3.14.

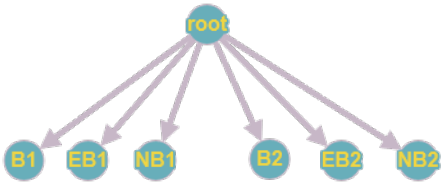


Figure 3.13: Path building preserving the nesting level of the code.

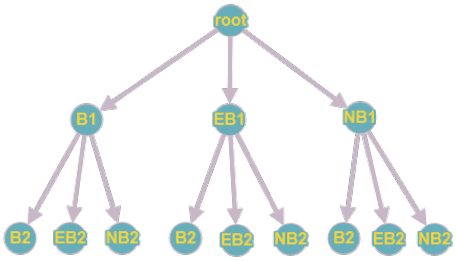


Figure 3.14: Path building as a search tree.

In the representation of Figure 3.13, the nesting level of the original code is preserved so a path composition algorithm must pick two nodes of the same tree level. Second representation on Figure 3.14 represents search tree where the possibility of picking each node from B2 is expanded in each B1 node. However second one increases one level of nesting. Second representation is more suitable to deliver a BFS search but we need a way to transform left one to right one when building the tree structure. With respect to building a path tree we need to solve to transform the SUT description into a CG and then to convert the CG into a tree.

The first issue is addressed in this section. Before declaring the algorithm we need to be familiar with the global variables used in Algorithm 7.
In the first place, *cg_root* stands for the root of the CG. Secondly, *NodeStructMap* consists of a hash table that will help us to build the links between the nodes in different procedure calls. Hash tables *last_number_map* and *prev_last_number_map*

Algorithm 7 Global data structures used in Algorithm 8.

```
1: procedure BFS_Global_Data_Types
2:
3:     cg_root ← CostNodeStruct(cost_tree_node(root, 0.0))  ▷Build the root of the CG
4:
5:     NodeStructMap  ▷HashMapVector whose key is an unsigned integer and the value a reference to CostNodeStruct;
6:
7:     last_number_map, prev_last_number_map  ▷HashMap whose key is an unsigned and the value a string
8:     to_paste_node ← ∅  ▷Reference to CostNodeStruct
9:
10:    father_ref_map ← ∅ ▷Hash table whose key in an unsigned and the value is a references to cost_tree_node
11: end procedure
```

are maps that help processing the numeric part of the constraint ID so as to infer nesting transition. *To_paste_node* is a reference holding an entire path tree structure from a constraint ID starting 1 level below the root. The purpose of this reference is to update the leaves of previous conditionals in level 1 of the tree. Eventually, *father_ref_map* stores references of parents node that are used to link the structure properly.
After initializing the input variables, Algorithm 8 implements a solution to transform a SUT description to a CG. In other words, given a description like the one displayed in Figure 3.13, Algorithm 8 transforms it to a structure similar to the one displayed in Figure 3.14.

This procedure is called *every time* a new constraint is added. *Push_Back_Constraint()* receives three input parameters, namely, *branch_pair* containing the ID of the branch along with its cost, *transition* which indicates whether the flow of the constraints goes down the graph, remains in the same level or goes up. *New_CID_in_root* indicates when a new conditional structure has been added to the root. This event is relevant because all leaves of previous level will need to build this subgraph when deploying BFS search. The first two variables defined in lines 3 and 4 are

Algorithm 8 Algorithm to make build a CG from SUT constraints description.

```
1: procedure PUSH_BACK_CONSTRAINT(branch_pair, nesting_level, transition,
   new_CID_in_root)
2:
3:     next_level ← nesting_level + 1
4:     prev_level ← nesting_level - 1
5:     last_node_ref ← ∅                ▷Static variable saving the last node added
6:     last_number ← get_last_number(branch_pair.ID)      ▷Gets the numerical
   part of a constraint ID
7:     different_last_number ← false
8:     last_number_map[nesting_level] ← last_number
9:
10:    if exists_key_in_map(prev_last_number_map, nesting_level) or
11:        last_number_map[nesting_level] ≠ prev_last_number_map[nesting_level]
   then
12:        different_last_number ← true
13:    end if
14:
15:    prev_last_number_map[nesting_level] ← last_number
16:
17:    if new_conditional_structure_in_root then
18:        add_to_root ← false
19:        if to_paste_node ≠ ∅ then
20:            join_paste_node_to_leaves(cg_root)
21:        end if
22:    end if
```

employed to describe adjacent levels. *Last_node_ref* is a static variable saving the reference of the last node added. *Last_number* stores the numeric part of the constraint ID e.g., 1.1.2 from the constraint ID B1.1.2. *Different_last_number* indicates whether last number e.g., 2 in previous example, has changed to detect a transition. Finally, *last_number_map* and *prev_last_number_map* are hash tables used to infer these shifts in the numeration. This is done in decision at line 12 in Algorithm 8.

Algorithm 9 Algorithm to paste new branch structure in root to previous leaves

```
1: procedure JOIN_PASTE_NODE_TO_LEAVES(refToCostNodeStruct block_ref)
2:     for all elem ∈ block_ref→node_vector do
3:         temp_block ← to_paste_node→node_vector.begin()→cg_child_vector.end()
4:         if elem→has_child_cg() and
5:             elem ← block_ref→node_vector.begin()  then
6:             for all child_ref ∈ elem→cg_child_vector do
7:                 join_paste_node_to_leaves (child_ref)
8:             end for
9:         else if not elem→has_child_cg() and block_ref ≠ temp_block then
10:                                                  ▷actual assignment:
11:            elem→cg_child_vector.push_back(temp_block)
12:        end if
13:    end for
14: end procedure
```

The next decisions declared in line 20 holds a statement that calls to procedure *join_paste_node-_to_leaves()*. This procedure is in charge of linking, when we have more than one constraint structure in the first level, to the leaves so as to facilitate a consistent search. Algorithm 9 describes a procedure in order to paste *to_paste_node* to the leaves in the form of a reference from which to build the proper subgraph. Such an algorithm implements a standard *preorder* tree traverse [96]. An example of the effect of this algorithm is displayed in Figure 3.15.

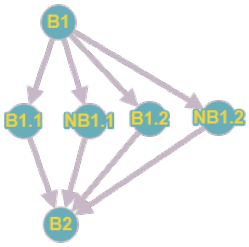


Figure 3.15: Example of the resulting CG after applying Algorithm 9. B2 in the root is linked to the leaves of previous structure B1.

Resuming *push_back_constraint* algorithm we observe that the first decision depicted from line 24 until 45 implements the decisions when traverse goes down. The decisions from line 24 to 39 discriminate when the assignment of the proper links must be done including the cases where links must be established in the root of the tree. After such a control flow in lines from 41 to 43 the proper father_node i.e., CTN, along with a reference is assigned in *NodeStructMap* so that we may access later.

The next and last part of the algorithm implements actions when the transition

```
23:     if transition = going_down then
24:         if new_conditional_structure_in_root then
25:             father_node ← to_paste_node
26:             father_ref ← to_paste_node→node_vector.begin()
27:             to_paste_node→node_vector.begin()
28:                     →cg_child_vector.push_back(NodeStructMap[1].end())
29:         else if nesting_level = 1 and add_to_root then
30:             father_node ← cg_root
31:             father_ref ← cg_root→node_vector.begin()
32:             cg_root→node_vector.begin()
33:                     →cg_child_vector.push_back(NodeStructMap[1].end())
34:         else if nesting_level > 1 then
35:             father_node ← NodeStructMap[prev_level].begin()
36:             father_ref ← last_node_it
37:             NodeStructMap[nesting_level].end()→father_ref
38:             →cg_child_vector.push_back(NodeStructMap[nesting_level].end())
39:         end if
40:
41:         NodeStructMap[nesting_level].begin()→father_node ← father_node
42:         NodeStructMap[nesting_level].begin()→father_ref ← father_ref
43:         father_ref_map[nesting_level] ← father_ref
```

goes down or same level. This is deployed in the branch between 44 and 58. A new CNS is constructed from which proper father references are assigned. Last decision depicted on line 60 removes the *next_level* so as to keep a consistent *NodeStructMap* when exploring the path tree supplied. Finally *last_node_ref* definition in line 65 holds the record for the next hypothetical call to the procedure.

```
44:     else if transition = same_level or transition = going_up then
45:         if different_last_number then          ▷New Branch sequence in the
46: same level e.g., NB1.1 .. B1.2
47:             nodeStruct ← ∅                              ▷new CostNodeStruct
48:             nodeStruct→father_node ← NodeStructMap[prev_level].begin()
49:             nodeStruct→father_ref ← father_ref_map[nesting_level]
50:             NodeStructMap[nesting_level].push_back(nodeStruct)
51:             nodeStruct→father_ref ← father_ref_map[nesting_level]
52:             NodeStructMap[nesting_level].push_back(nodeStruct)
53:         end if
54:         father_ref_map[nesting_level].cg_child_vector.push_back
55:                 (NodeStructMap[nesting_level].end())
56:     end if
57:
58:     if transition = going_up and exists_next_level(NodeStructMap, next_level)
    then
59:         NodeStructMap.clear(next_level)        ▷Erase key entry and all values
60:     end if
61:     last_node_ref ← NodeStructMap[nesting_level].end()→push_back
62:     (cost_tree_node(branch_pair.ID, branch_pair.cost));
63: end procedure
```

3.3.5 From a CG to a BFS Tree

By definition, a tree is particular case of a graph which doesn't contain any cycles and each node can't have more than one father excluding the root [96]. Our notion of a BFS consists of a standard tree where each node contains the necessary cost data to successfully carry out a BFS search. Once an initial CG has been built and subsequent branches in level 1 references have been pasted to the leaves, one more function must be executed. A function must transform the plausible multiple children of the CG into a BFS tree. This shift in the structure is displayed in Figure 3.16 that would be the result of applying such a transformation in Figure 3.15. Figure 3.16 repre-

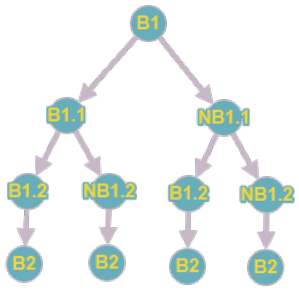


Figure 3.16: Tree structure changing the original nesting level where it's easier to apply BFS.

sents proper nesting level and a tree structure where BFS can be easily applied.

Algorithm 10 Algorithm that builds a subtree from the CG references

```
1: function BUILD_SUB_TREE (itr_ref, father_cost, father_ID, recursive_call,
   RefTo_leave_nodes_to_call_cost_update)
2:     build_children_from_cg ← itr_ref→node_vector.begin().cg_child_vector ≠ ∅
3:     cg_refs_to_build ← ∅                    ▷Vector of references to CostNodeStruct
4:     prio_queue ← ∅                          ▷Priority queue composed by
   NodeFatherToPick structure. The criterion for the order of the queue is the
   comparison of the accumulated cost
5:     sub_level_created ← update_nodes_different_level ← false
6:     first_queue_item_added ← false
7:
8:     if build_children_from_cg then
9:         if itr_ref→node_vector.begin().apply_construction_with_filter then
10:            return_father_ptr ← build_sub_bfs_tree_with_filter (itr_ref,
11:            father_cost, father_ID)
12:            return return_father_ref;
13:        end if
14:        if recursive_call > 0 then
15:            for all node ∈ itr_ref→node_vector do
16:                if node.ID = father_ID then
17:                    cg_refs_to_build ← node.cg_child_vector
18:                    break
19:                end if
20:            end for
21:        else
22:            cg_refs_to_build ← itr_ref→node_vector.begin().cg_child_vector
23:        end if
```

The Algorithm 10 implements the building function of nodes when BFS search prescribes so. *Build_sub_tree* function returns a reference of the node from which a subtree has been built. It takes 4 input parameters, namely, *itr_ref* is a reference of the node that needs to be built, *father_cost* is the accumulated cost of the father node, *father_ID* is the constraint ID of the father, *recursive_call* is a counter to know the recursion level, *RefTo_leave_nodes_to_call_cost_update* is a reference to a priority queue that is later used to update the costs of the tree.

With respect to local variables implemented from lines 2 to 4 we get that: *build_children_from_cg* is a boolean variable that corroborates that a children needs to be built, *cg_refs_to_build* is a vector of references of nodes that are going to be copied for the building, *prio_queue* is a priority queue that are going to be later used to write *nodes_to_update* in a sensible order. After that some boolean variables are defined to be later used. The decision between lines 8 and 12 calls to *build_sub_tree_with_filter()* an equivalent function that we will investigate later. Next control flow stated between lines 14 and 22 initializes the vector of references to CG whose child are going to copied and explored.

```
24:     for all cg_children_vector_ref ∈ cg_refs_to_build do
25:         level_d ← copy_node (cg_child_ref)          ▷Building new node
26:         return_father_ref ← level_d
27:         if update_nodes_different_level then
28:             item ← prio_queue.top()
29:             level_d→father_node ← item.cost_node_ref;
```

The main loop, which is expanded from line 24 line 77, explores the children. In this loop a copy of the node is created from the reference of the CG (line 25). Conditional from line 27 to 49 is executed when a created node needs to be aware of the fact it must update only specific nodes of its father. In this code section the content of the *prio_queue* is backed up in while loop in line 34, giving priority to those parents with the greatest accumulated cost to keep consistency of the accumulated cost along the tree. If there is no need to update nodes specific nodes proper links are assigned in line 50 after the construction. *Prio_queue* is cleared before being written in line 52.

```
30:             for all node ∈ level_d→node_vector do
31:                 node.bfs_father ← item.cost_node_ref
32:             end for
33:             first_queue_item_added ← false
34:             while prio_queue ≠ ∅ do
35:                 node_update ← ∅
36:                 if not first_queue_item_added then
37:                     node_update ← {prio_queue.top().cost_node_ref,
38:                                    prio_queue.top(), ID, ∅}
39:                 else
40:                     node_update ← {prio_queue.top().cost_node_ref,
41:                                    prio_queue.top(), ID, cg_child_ref}
42:                 end if
43:                 level_d→nodes_to_update.push_back(node_update)
44:                 prio_queue.pop()
45:                 first_queue_item_added ← true
46:             end while
47:             update_nodes_different_level ← false
48:             assign_child(level_d→father_node, level_d, cg_child_ref)
49:         else
50:             assign_father_and_children (level_d, father_ref)
51:         end if
52:         prio_queue ← ∅
```

After a new node is created (*level_d*) the children references are explored in loop spanning from line 53 to 75. The first part of the loop between 55 and 61 is dedicated to update the costs. Line 65 interrogates whether the iterating node has a child in the CG but it does not have in the BFS tree. In such a case it needs to be built by delivering a recursive call to another function. Line 74 adds the new *NodeFatherToPick* created. A key data in the initialization is the presence or absence of the father ID ($\emptyset$ or jt.ID respectively). This data later is used to tell whether all father nodes or some specific ones must be updated as the reader may observe in line 12 in Algorithm 6.

Line 76 updates proper link. Finally, if the last node is a leaf (*not sub_level_created*) this is added in the priority queue *leave_nodes_to_call_cost_update* to launch the update later. Should *build_sub_tree* is called the first time we need to build the

```
53:             for all node ∈ level_d→node_vector do
54:                 sub_level_create ← false                    ▷Updating costs
55:                 if cg_child_ref = cg_refs_to_build.begin() then
56:                     node.parents_max_cost ← father_cost
57:                 else
58:                     node.parents_max_cost ← node.bb_father→begin().parents_max_cost
59:                             + node.bfs_father→node_vector.front().cost
60:                 end if
61:                 node.accumulated_cost ← node.cost + node.parents_max_cost
62:                 build_children_from_cg ← node.cg_child_vector ≠ ∅
63:                 no_bfs_children ← not node.has_child_bfs()
64:                 new_node ← ∅                    ▷new NodeFatherToPick created
65:                 if no_bfs_children and build_children_from_cg then
66:                     sub_level_create ← update_nodes_different_level ← true
67:                     father_ref ← build_sub_tree (level_d,        ▷Recursive call
68:                             node.accumulated_cost, node.ID, recursive_call + 1)
69:                     new_node ← {∅,
70:                     father_ref → node_vector.begin().accumulated_cost, father_ref}
71:                 else
72:                     new_node ← {jt.ID, jt.accumulated_cost, level_d}
73:                 end if
74:                 prio_queue.push(new_node)
75:             end for
76:             father_ref ← level_d
77:         end for
78:     end if
79:
80:     if not sub_level_created then
81:         leave_nodes_to_call_cost_update→push(return_father_ref)
82:     end if
83:
84:     return return_father_ref
85:
86: end function
```

tree, it often occurs that we need to partially build the tree as a consequence of the deletion of nodes. So for example, in Figure 3.12 assuming that π' is the same subpath and we build two paths $\pi_1 = \{\pi' \cup \{\text{NB1.1.2, B1.8}\}\}$ and $\pi_2 = \{\pi' \cup \{\text{NB1.1.2, NB1.8}\}\}$ we would delete {NB1.1.2, B1.8, NB1.8} however

we would like to form combinations with B1.1.2 for which we would need to build again {B1.8, NB1.8}. In this case we would need to search for node B1.1.2 and build from there on. For this reason *build_sub_tree_with_filter* is implemented.

Algorithm 11 Global variables used by build_sub_bfs_tree_with_filter

```
1: node_ID_sought ← ∅
2: apply_filter ← false
```

Before explaining *build_sub_tree_with_filter* it is worth presenting the two main global variables used in this algorithm displayed in Algorithm 11. *Node_ID_sought* is indeed the ID of the node after which to build a subtree. *Apply_filter* will be used to trigger building actions.

Algorithm 12 Algorithm that builds a subtree when the search has been partially built but it needs additional building because of a deletion

```
1: function BUILD_SUB_BFS_TREE_WITH_FILTER(itr_ref, father_cost, father_ID,
   initialize_data ← true, actual_father ← ∅)
2:     father_ref ← itr_ref
3:     return_father_ref ← temp_father_ptr ← ∅
4:     vector_to_iterate ← ∅          ▷Vector of references to CostNodeStruct
5:     prio_queue ← ∅                 ▷same use build build_sub_tree
```

The header and implementation are very similar to *build_sub_tree*. The three first parameters has the same meaning of *build_sub_tree*. *Initialize_data* is initialized by default to true and triggers action to initialize search data. *Actual_father* is a reference to *CostNodeStruct*. From the three local variables defined in the body of the function *vector_to_iterate* consists of a vector of references to be explored later. The first decision spanning from line 8 to 19 initializes search data when appropriate or just initialized the the *vector_to_iterate* the constraint ID of the father matches.

Likewise lines from 20 to 25 initializes search data. The main decision of the algorithm is satisfied when there is a need to build by comparing with the CG structure. This decision is implemented between lines 27 and 92. The main loop

```
6:    update_nodes_different_level ← first_queue_item_added
7:    ← children_explored_recursively ← false
8:    if initialize_data then
9:        node_ID_sought ← itr_ptr←node_vector.begin().cg_child_vector.end()
10:       actual_father ← father_ref
11:       itr_ref ← cg_root→node_vector.end().cg_child_vector.begin().cg_child_vector
12:       vector_to_iterate ← itr_ref←node_vector.begin().cg_child_vector
13:   else
14:       for all it ∈ itr_ref→node_vector do
15:           if it.ID = father_ID then
16:               vector_to_iterate → it.cg_child_vector
17:           end if
18:       end for
19:   end if
20:   if vector_to_iterate ≠ ∅ then
21:       vector_to_iterate ← itr_ref→node_vector.begin().cg_child_vector
22:   end if
23:   if not initialize_data and actual_father ≠ ∅ then
24:       father_ref ← actual_father
25:   end if
26:
27:   build_children_from_cg ← itr_ref→node_vector.begin().cg_child_vector ≠ ∅
28:   if build_children_from_cg then
29:       for all cg_child_ref ∈ vector_to_iterate do
30:           children_explored_recursively ← true
31:           if apply_filter then
32:               if cg_child_ref→node_vector.begin().ID = node_ID_sought then
33:                   apply_filter ← false
34:               end if
35:               build_children_from_cg ← cg_child_ref→node_vector.begin()
36:               cg_child_vector ≠ ∅
37:               if temp_father_ref ≠ ∅ then
38:                   father_ref ← temp_father_ref
39:               end if
40:               children_explored_recursively ← true
41:           end if
42:           if not apply_filter and not children_explored_recursively then
43:               level_d←copy_node(cg_child_ref)          ▷Actual construction
44:               return_father_ref ← level_d
```

explore the children from line 29 to 94. The first decision from line 31 to 40 prepares the links and variables when the sought ID is found in line 32. The actual construction is delivered in line 42. The conditional *updates_nodes_different_level* and the rest of the algorithm is equivalent to *build_sub_tree* from line 44 to the end.

```
45:             if update_nodes_different_level then
46:                 item ← prio_queue.top();
47:                 level_d←father_node ← item.cost_node_ref
48:                 for all it ∈ level_d→node_vector do
49:                     it.bfs_father ← item.cost_node_ref
50:                 end for
51:                 first_queue_item_added ← false
52:                 while prio_queue ≠ ∅ do
53:                     if not first_queue_item_added then
54:                         level_d→nodes_to_update.push_back({
55:                         prio_queue.top().cost_node_ref,
56:                         prio_queue.top.ID, ∅ })
57:                     else
58:                         level_d→nodes_to_update.push_back({
59:                         prio_queue.top().cost_node_ref,
60:                         prio_queue.top.ID, cg_child_ref})
61:                     end if
62:                     prio_queue.pop()                    ▷Remove last element
63:                     first_queue_item_added ← true
64:                 end while
65:                 assign_bfs_child(father_ref, level_d, cg_child_ref)
66:             else
67:                 assign_father_and_children (level_d, father_ref,
68:                 father_ID)
69:             end if
70:             prio_queue ← ∅
71:             for all jt ∈ level_d→node_vector do
72:                 if cg_child_ref = vector_to_iterate.begin() then
73:                     jt.parents_max_cost = father_cost
74:                 else
75:                     jt.parents_max_cost =
76:                     jt.bfs_father→node_vector.begin().parents_max_cost
77:                     + jt.bfs_father→node_vector.begin().cost
78:                 end if
```

```
79:                     jt.accumulated_cost ← jt.cost + jt.parents_max_cost
80:                     build_children_from_cg ← jt.cg_child_vector ≠ ∅
81:                     if build_children_from_cg then
82:                         father_ref ← build_sub_bfs_tree_with_filter (
83:                         level_d, jt.accumulated_cost, jt.ID, false, ∅)
84:                         update_nodes_different_level ← true
85:                         prio_queue.push(∅,
86:                         father_ref→node_vector.begin().accumulated_cost,
87:                         father_ref)
88:                     else
89:
90:                         prio_queue.push(jt.ID , jt. accumulated_cost, level_d)
91:                     end if
92:                 end for
93:                 father_ref ← level_d
94:             end if
95:         end for
96:     end if
97:     if initialize_data then
98:         apply_filter ← false;
99:         leave_nodes_to_call_cost_update.push(return_father_ref)
100:     end if
101:     return return_father_ref
102: end function
```

3.3.6 Path Composition

So far we have described algorithms advocated to give a consistent cost bound as well as how build CG and BFS tree. The last bit to build encoded constraint paths is to declare our BFS algorithm that will compose the paths leading to the greatest cost. Such an algorithm is described in Algorithm 13.

Algorithm 13 Algorithm that builds a search tree from CG and run a BFS. It returns a vector with all paths after running this search

```
1: function BEST_FIRST_SEARCH_SEARCH(limit ← 20000)
2:
3:     bfs_graph_root ← copy_node (cg_root);
4:     level_1 ← copy_node (cg_root→node_vector.begin().cg_child_vector.end())
5:     assign_father_and_children (level_1, bfs_graph)  ▷Assign proper references
6:     itr_ref ← bfs_graph.node_vector.begin().bfs_child
7:     has_potential_bfs_children ← false
8:     path_vector ← ∅                          ▷vector containing the paths computed
9:     leave_nodes_to_call_cost_update ← ∅          ▷queue of references to
   CostNodeStruct
```

Firstly, this algorithm accepts a limit of the path vector in case a complete search could not be achieved. The limit by default is set to 20000. There is no rationale behind this decision other that a "*finger in the air*" number that emerged during experimentation. This is approximately the numbers of paths that could be checked in two hours. Regarding the body of the algorithm the first 3 statements between line 3 and 5 build a copy of the CG graph root into a BFS tree root (*bfs_graph_root*) starting from the root and building the first level. *Itr_ref* is the key reference that is going to be used to explore the constructed path tree. After that, in lines 7 and 8, *has_potential_bfs_children* boolean variable is defined to indicate whether a node must build some children from the data of the CG. *Path_vector* in line 8 is a vector containing all paths to be constructed. In line 9 *leave_nodes_to_call_cost_update* consists of a queue of nodes that to update the costs of the entire tree. Such a variable also appears in former algorithms.

```
10:
11:     do
12:         is_bfs_leaf ← has_potential_bfs_children ← false
13:         itr_ref ← bfs_graph.node_vector.begin().bfs_child
14:
15:         if itr_ref ≠ ∅ or itr_ref→node_vector ≠ ∅ then
16:             finished_bfs_search ← true
17:             bfs_graph ← ∅
18:         else
19:             finished_bfs_search ← false
20:         end if
21:
22:         path ← ∅
23:
24:         while not is_bfs_leaf and limit > 0 and not finished_bfs_search do
25:
26:             if itr_ref→node_vector ≠ ∅  then
27:                 is_bfs_leaf ← has_potential_bfs_children ← true
28:             else
29:                 has_potential_bfs_children ← not itr_ref→node_vector.begin().
30:                 cg_child_vector.empty()
31:             end if
32:                                                 ▷⨁ denotes concatenation
33:             path ← path ⨁ itr_ref→node_vector.begin().ID
34:
35:             if is_bfs_leaf and not has_potential_bfs_children then
36:                 if not itr_ref→node_vector ≠ ∅ then
37:                                                 ▷last costs statements
38:                     itr_ref→delete_node()
39:                 end if
```

Inside the main do while, spanning from line 11 to 58, we have some others initialization including *is_bfs_leave* in charge of telling when the node explored is a leaf and thus a *necessary* condition to build new nodes would be met. The conditional structure from line 15 to 20 determines when the BFS is concluded and if so, the structure of the root is cleared. *Path* in line 22 is an important string variable that will store each constraint ID and thus conforming the encoded path.

```
40:             else
41:                 if not itr_ref→node_vector.begin().has_child_bfs()
42:                     and has_potential_bfs_children then
43:                     build_sub_bfs_tree (itr_ref, itr_ref→node_vector.begin().cost,
44:                             itr_ref→node_vector.begin().ID,
45:                             ref_to(leave_nodes_to_call_cost_update))
46:
47:                     Update_Cost_From_Leaves (ref_to(leave_nodes_to_call_cost_update))
48:                 end if
49:                 if itr_ref ≠ ∅ then
50:                     itr_ref ← itr_ref→node_vector.begin().bfs_child
51:                 end if
52:             end if
53:         end while
54:         if path ≠ ∅ then
55:             path_vector.push_back(path)
56:             limit ← limit - 1;
57:         end if
58:     while limit > 0 and not finished_bfs_search
59:
60:     return path_vector
61:
62: end function
```

The most relevant loop within Algorithm 13 is the *while* loop deployed between line 24 and 53. The *path* variable in line 33 adds a new constraint ID in each iteration. The two main decisions in such a loop are defined in decisions 35 and 40. First one is able to identify when a leaf of a tree is reached. That leaf node is deleted from the tree and thus cost must be updated. The latter is done in *delete()* procedure. The *if* structure from line 41 to 48 takes over when BFS search needs to build a new node. As explained earlier *build_sub_bb_tree* function assigns memory and constructs subsequent nodes. Next procedure *Update_Cost_From_Leaves* delivers proper calling from the leaf nodes to keep consistency of the cost in the tree. This procedure will be explained later. The reference *itr_ref* is updated to the next element to explore in line 48. Lastly, the path is added to the *path_vector* in line 55.

Algorithm 14 Algorithm to trigger the costs update in the tree

```
1: procedure UPDATE_COST_FROM_LEAVES (leave_nodes_to_call_cost_update
   ▷reference input data)
2:     while leave_nodes_to_call_cost_update ≠ ∅ do
3:         if not leave_nodes_to_call_cost_update.begin()→node_vector.begin()
4:                                         .has_child_bfs() then
5:                             ▷Start with the node with the greatest accumulated cost
6:             leave_cost ← leave_nodes_to_call_cost_update.node_vector.begin()
7:             leave_nodes_to_call_cost_update.begin()→update_accumulated_cost
8:                                                                 (leave_cost)
9:         end if
10:        leave_nodes_to_call_cost_update.pop()
11:    end while
12: end procedure
```

Lastly, *Update_Cost_From_Leaves* in Algorithm 14 just backs up the queue and call *update_accumulated_cost* from leaves so as modify the cost of the entire tree and thus undertake a consistent search.

So far the most relevant algorithms of the CBTG of GenI are outlined. Next, SBTG and RTG implementation are addressed.

3.3.7 Search-Based and Random Test Generator

The objective of the SBTG implementation is to reproduce Law and Bate implementation [18] as it represents the state-of-the art in SBT for MBPTA. In this work they implemented the Simulated Annealing and focus on those fitness functions that target execution time. Nonetheless, the actual Simulated Annealing algorithm version was taken from [27, Figure 4.5] as the branches implemented in this version enable us to discern whether a solution is an actual improvement, whether it is accepted randomly or if it is rejected. This data is quite relevant for the verification of the algorithm.

Because of our assumption of the absence of the instrumentation tool only two fitness functions are implementable from Law and Bate work [18]. The first one

is $Fitness_{ET}$ is advocated to identify the largest execution times:

$$Fitness_{ET} = \frac{Current\_Time - Previous\_Time - 1}{Previous\_Time} \tag{3.4}$$

Each new observed execution time ($Current\_Time$) is compared to $Previous\_Time$, which in turn is defined as the observed HWM [18] up to that point in the execution. $Current\_Time$ is indeed the observed execution time in the concrete iteration where the fitness function is called. The function is normalized by using the observed HWM. In summary, this heuristic is advocated to find greater execution time each time. Because of the arithmetics of this fitness function, a negative feedback is returned in the event of $Previous\_Time \geq Current\_Time$ and thus that solution would be either accepted randomly or rejected. $Fitness_{ET}$ gave good results despite its simplicity [18].

The other optimization function named *Unique Execution Times* targets triggering different paths by observing changes in the execution time. Unfortunately this function did not perform well in the case studies presented in [18] and that's why it is omitted.

Temperature of the Simulated Annealing ranges from $[0.1, 0.01]$. These and the rest of Simulated Annealing data are decided following Law and Bate prescriptions [18]. Furthermore, the cooling temperature function is described in Algorithm 15 (implemented from the specification of the paper [18] and internal communications). This function records the temperature every time there is an improvement in the fitness function i.e., first decision. In the event of rejecting more than 200 test vectors the temperature is reheated so as to avoid a premature stagnation of the search [18] . After that, the new cooled temperature spanning from line 8 to 14 is calculated by using a linear equation. The *slope* is defined in line 11 and the *intercept* in the next line. The slope takes into account the maximum number of iterations of the SA that is equivalent to the number of test vectors generated. Such a limit is set to 10000 which allegedly suffices to analyze jet engine controller software [18]. Finally, in reference to the stop criterion the Simulated Annealing stops if more than 1000 solutions are rejected, the temper-

ature is less than 0.01 or watchdog timer alerts of spending more than 12 hours of execution.

Algorithm 15 Algorithm to trigger the costs update in the tree

```
1: function CALCULATE_NEW_TEMPERATURE(iteration, temperature, no_solutions_accepted)
2:     if no_solutions_accepted = 0 then
3:         last_temperature ← temperature                    ▷static variable
4:     end if
5:
6:     if no_solutions_accepted ≥ 200 then
7:         temperature ← last_temperature
8:     else
9:                                          ▷maximum test vectors generated
10:        maximum_number_iterations ← 10000
11:        m ← -0.09 / (maximum_number_iteration × 0.9999)
12:        n ← 0.1001
13:        temperature ← iteration × m + n;
14:    end if
15:    return temperature;
16: end function
```

Eventually, RTG is included and it has a trivial implementation. The purpose of this test generator is to have a random basis against which to compare the results to test statistical significance. Both SBTG and RTG were implemented *pseudorandomly*. This means that both of these TG algorithms are deterministic in their output but they look random [54]. The different output in each trial stems from using a deterministic random value test generator that takes different seeds as input. The motivation of this decision is to fulfil repeatability and reproducibility requirements demanded by certification authorities [19].

3.4 Evaluation and Case Studies

To reiterate, the purpose of the evaluation described is to assess Contribution 5 from Section 2.5. The empirical evaluation of the test generators implemented in GenI encompasses different research questions. Research questions 1 and 3 are mainly concerned with the central hypothesis of this work.

- **Research Question 1 - Effect of the test generators on the HWM**. What is the test generation method that achieves the global HWM? This is measured by comparing the HWMs as this data is instrumental for MBTA. This data is measured in clock cycles in the target architecture as this measure is very accurate and best records the effect under study. In the next equation, j and i denote different test generators.

$$\text{Relative HWM}(\%) = 100 \times \frac{HWM_i - HWM_j}{HWM_i} \;\; ; \;\; \forall j \neq i$$

- **Research Question 2 - Effect of the test generators on the execution time distribution**. Based upon the resulting execution time distributions, which test generator provide significantly different results? This is measured by the statistical tests described below.

- **Research Question 3 - Effect of the test generators on the wall time**. When the HWM is observed during test generators execution? This metric considers the time taken for test generation and initialization of the test vector on-target, the execution on the SUT and the collection of execution time. Despite the fact that results are contributed by these three factors, this metric is advocated to measure the effect of the test generation time since the rest is virtually the same as they are independent modules.

- **Research Question 4 - Effect of the test generators on the efficiency (η)** which is deemed the number of test vectors during the wall time. How many test vectors are generated per time unit? This metric is advocated to measure how quick each test generator produce test vectors. Because the on-target testing time is similar across different test generator this difference will come from the test generation time mainly.

$$\eta = \frac{\#Test\_Vectors}{Wall\_Time} = \frac{\#Execution\_Times}{Wall\_Time}$$

$$\text{Relative Efficiency}(\%) = \eta_{\Delta} = 100 \times \frac{\eta_i - \eta_j}{\eta_i} \;\; ; \;\; \forall j \neq i$$

- **Research Question 5 - Effect of the test generators on RAM memory usage.** How much RAM memory each test generator uses? The aim of this metric is to observe the differences of the static processing of the different test generators since CBTG delivers a greater static analysis in contrast to SBTG or RTG each time. As part of the framework, the Working Set Size of the test generator process is sampled every time a test vector was generated. A working set of a process is "*the set of pages in the virtual address space of the process that are currently resident in physical memory*" [97]. The sampled data is expressed in bytes [98] so it will be converted to megabytes in the results. Relative RAM use has a similar definition to HWM.

- **Research Question 6 - Effects of the slicing of the CBTG on the run-time**. What is the difference in the run-time between running a program with and without slicing? The run-time of the application of the BFS is measured in the host computer in seconds as this metric effectively records the intended effect in the performance. Again, 100 run-times were collected to have a good statistical power. The host computer is equipped with an Intel Core i5-7200U processor running at 2.5GHz. The calculation of the percentage of slice removal is calculated comparing the size of the list of constraints and definitions resulting from the program with and without slicing. This comparison is advocated to show the difference in the slicing effect.

- **Research Question 7 - Accuracy of the cost in the resulting execution time**. How representative is the derived cost w.r.t the actual execution time? Even though, experiments in Subsection 3.2.4 concluded that the cost is not accurate, some data about a larger scale application of

the cost and the actual execution time is shown. This metric is designed to assess our contribution on BFS as a way to build paths. This approach uses the notion of cost.

When the resulting evidence is collected, it is normally checked using statistical significance tests so as to enhance the confidence of the conclusions. These tests are applied evaluating the empirical data and they are non-parametric. The actual reason for using them was because they were used by Law and Bate [18] in a similar evaluation.

1. **Friedmann test** as described in Subsection 3.2.4.
2. **Wilcoxon-Nemenyi-McDonald-Thompson Test** (WNMT) compares every possible combination of the results of the different methods to figure out where the differences come from.

It is worthy of mention that these tests require the same number of observations as input. This is certainly an issue bearing in mind the idiosyncrasies of each test generator. For these reasons these tests were applied only from the first observation until minimum size of the sample of each testing session.

To recap, the tests generators employed are the ones implemented in GenI.

- **Constraint-Based Test Generator** implementing the program slicing, BFS algorithm to build paths and a constraint solver [90] to solve the path constraints. This method represents our contribution.
- **Search-Based Test Generator** implementing the Simulated Annealing described in the former section whose fitness function maximizes the execution time. This method is advocated to be representative of the state-of-the-art in MBTA [18].
- **Random Test Generator** which aims for providing random evidence against which to verify the significance of the results.

For the sake of confidence each test session was repeated 15 times changing the seeds. SBTG was provided with the stop criterion described above for a fair comparison. The time spent at the test generation for the test generation is set by the run time of the SBTG. Nonetheless, CBTG automatically stops if path coverage is achieved or a limit of 20000 paths is surpassed. This limit is set taking into accounts that the program may run out of memory as a consequence of paths explosion. Sometimes I am using the term *trial* to refer to a test session.

Unless another specification is stated, the hardware platform and drivers library are the same one used in Subsection 3.2.4. The reason for choosing this setting is their availability of the drivers for our Ada-Run Time as well the ARM architecture is often used in real-time embedded systems.

Eventually, the current section shows four case studies by using four benchmarks. They aim to provide examples of Real-Time software and have different characteristics such as the domain they are applied or their inner software structure. More details about these programs can be found in each individual case study.

(a) The first one consists of the pathological example for SBT as displayed in Listing 3.1.

(b) Second one consists of a more complex example extracted from an Spark book [15] and implements an autopilot navigation system. The aim of selecting this example is to have a representative Real-Time benchmark employing a variety of programming constructs.

(c) Third example, Certyflie [99] consists of a stabilizer control loop function implementing a PID control system. Unlike, autopilot, this is part of software that has actually been applied for a drone control. Again, the aim is to have another example of Real-Time benchmark implementing a control system for an aerospace system.

(d) Fourth example, RC Car [100] benchmark implements a controller of a radio controlled car. The purpose of this benchmark is to have an example from a car, another type of vehicle using Real-Time software.

It is worth remembering that apart from the features discussed the above, all these benchmarks have constraints that can be collected statically and path coverage can be achieved. In the next chapter, we will be looking at relaxing these assumptions. The latest subsection is concerned with discussing the threats to validity.

3.4.1 Needle in a Haystack

Given that the peculiarities of this benchmark were laid out at the beginning of this chapter, the current subsection is restricted to display and discuss the results of the GenI framework.

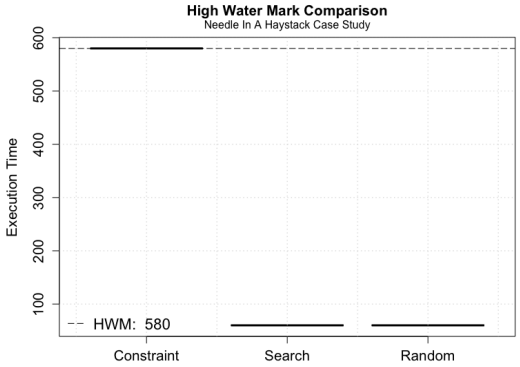


Figure 3.17: Resulting HWM boxplot.

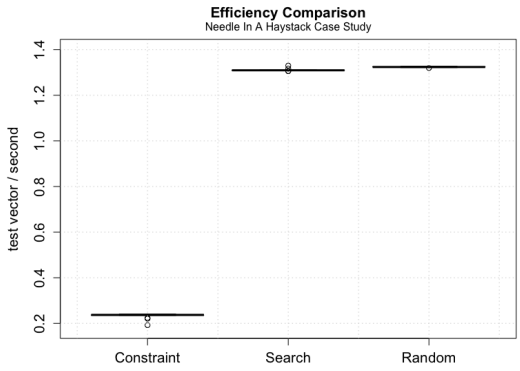


Figure 3.18: Test vectors per second generated in each test generator.

Figure 3.17 displays the HWM in *boxplot* format of each test generator as is advocated to respond research question 1. The results gave two execution times 580 and 60 (89.65% difference) corresponding to the only two paths this benchmark contains. Since there are only two paths the HWM could be the WCET or be really close to it. Only CBTG is able to hit the relevant branch and thus to trigger the HWM of all test generators. By contrast, the SBTG does not hit main branch in all of its test sessions which is aligned with the reported limitation of SBT [29]. In reference to research question 3 about the efficiency, which is illustrated in Figure 3.18, as it could be expected CBTG provides the lowest one

in comparison to SBTG (around −5.65 times) and RTG. RTG exhibits the best efficiency given the simplicity of its code but such an efficiency is not that distant from SBTG.

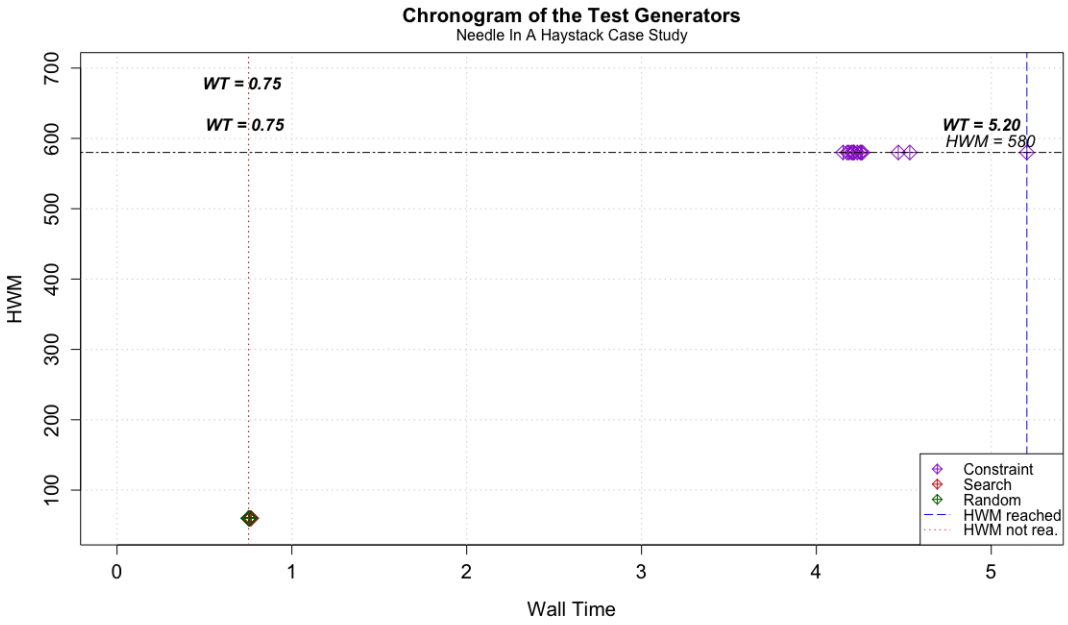


Figure 3.19: Chronogram of the analysis of Listing 3.1. WT denotes Wall Time measured in seconds.

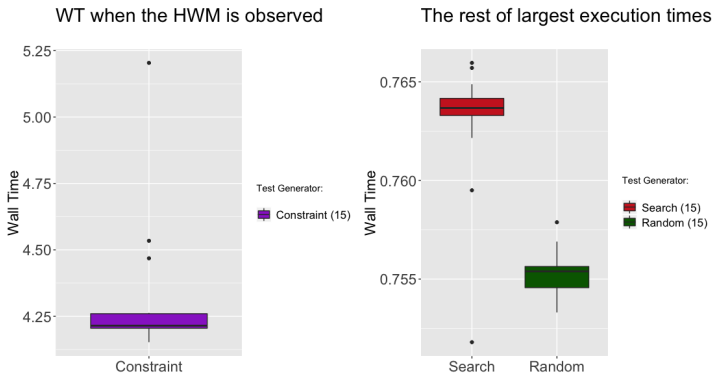


Figure 3.20: Boxplot of the observed largest execution times of the chronogram. Numbers in the parenthesis stand for the number of observations in each boxplot.

Figure 3.19 displays a **chronogram**, whose purpose is to depict the results of research question 3. Each symbol denotes the HWM in each trial by each test generator. The horizontal grey line represents the absolute HWM or the greatest observed execution time of *all* trials. Results show that although RTG and SBTG hit their HWM very soon they are very far from the observed HWM.

Alongside the chronogram, Figure 3.20 depicts the wall time of when the HWM was hit (on the left) and when the rest of largest execution time of a specific session was hit (on the right). Because the HWM data unveiled by the Constraint method and the other two generators differs so much a statistical comparison of wall times would not make much sense an that is why we decided to skip it. Regarding HWM and efficiency given the number of execution time data as a result of the simplicity of the benchmark we decided not to apply any further statistical tests. Admittedly, this case study was rather simple but shows some key differences amongst different test generators. However, the static processing of this benchmark is trivial.

3.4.2 Autopilot Case Study

This case study employs an open-source benchmark implementing a navigation system of a plane where the input is provided by some sensors in the cockpit. The simulated output are the actuators of the ailerons of the airplane. The original benchmark was modified to meet our needs regarding the compatibility of the static analyzer as well as focusing on those parts that may be of interest for a MBTA case study. The original root functions call two control procedures, namely, one dedicated to control the *altitude* and the other dedicated to control the *heading*. After inspecting visually the structure of the altitude control procedure, it was deemed to be more interesting for our analysis and that's why it was sliced for our benchmark. This resulting benchmark holds an array of states to deliver a control function.

To make the SUT controllable the writing of this array was enabled from the

test harness. Unfortunately, after all these changes the benchmark was still not fit for timing analysis after observing its timing profile. The main reason is that the cardinality of the set of execution times was really low regardless of the test generator used. To solve this issue an additional branch triggering a loop was implemented. The satisfaction of this predicate tries to simulate another time of fault accommodation code decision when comparing the difference of three input signals measuring the same magnitude. This modification was inspired on Triple Modular Redundancy software management often found in avionics systems [94]. After this change the resulting execution time profile was more interesting whereby a timing analysis case study was performed. These signals were added as input data.

In a nutshell this benchmark has 1 array composed by 10 integers, 8 integers variables and 2 enumerates. GenI tool reported that this benchmark has 4052 paths from which only 293 are feasible (7.231 %). This latter data could only be reported thanks to the CBTG. In reference to the results these are displayed next.

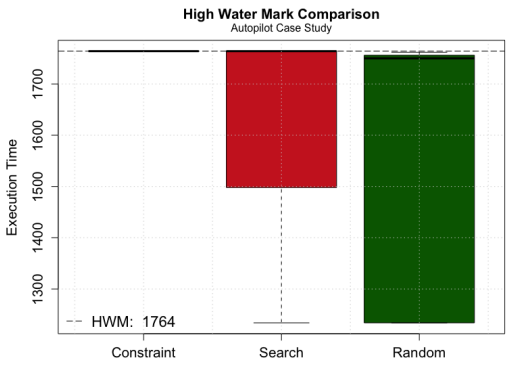


Figure 3.21: HWM Boxplot Comparison

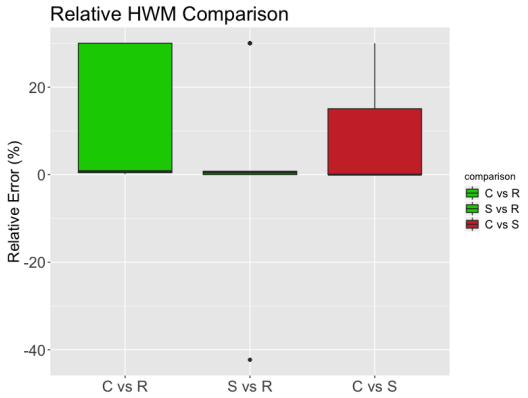


Figure 3.22: Comparison of the *difference* in HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Figure 3.21 shows a *boxplot* of the HWM of each trial and each test generator. It

is dedicated to show evidence for research question 1. This figure portrays that CBTG's HWM are pretty much the same and hit the absolute HWM. SBTG exhibits more deviated data than CBTG but closer to the HWM than RTG. RTG offers some dispersed execution times. On the right side, Figure 3.22 displays the relative and the statistical significance tests w.r.t the HWM distributions showing that both CBTG and SBTG are statistically significant with respect RTG techniques. CBTG vs SBTG results were not claimed to be significant due to the fact that most HWM of the SBTG are the absolute HWM. On average, the difference across the unveiled HWM is close to 0%.

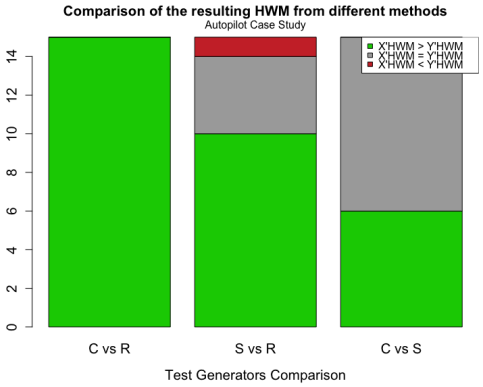


Figure 3.23: HWM Comparison

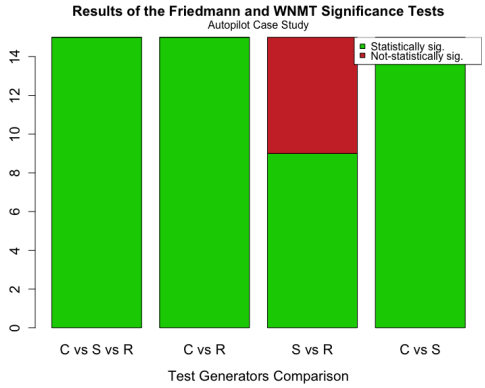


Figure 3.24: Friedmann and WNMT Test results.

Figure 3.23 shows the exact comparison of HWM across different trials and test generators. CBTG method always beats RTG, it beats SBTG in 6 out 15 cases (40%) and draws in the rest. Likewise, SBTG outperforms RTG in 10 out 15 cases (66.67%), it draws in 4 and is underperformed in one single case.

On the right side, in Figure 3.24 displays the significance tests statistical tests applied to initial execution times. This graph targets research question 2. The Friedmann test contends that the three test generators results are different. By analyzing pair-wise distributions the vast majority are different between each other (3 last columns on the right). CBTG vs RTG and CBTG vs SBTG are always concluded to be different methods. SBTG vs RTG are different in 9 cases

(60%) and the same in the rest. This could be explained since SA behaves *"more randomly"* in the beginning due to the big temperature.

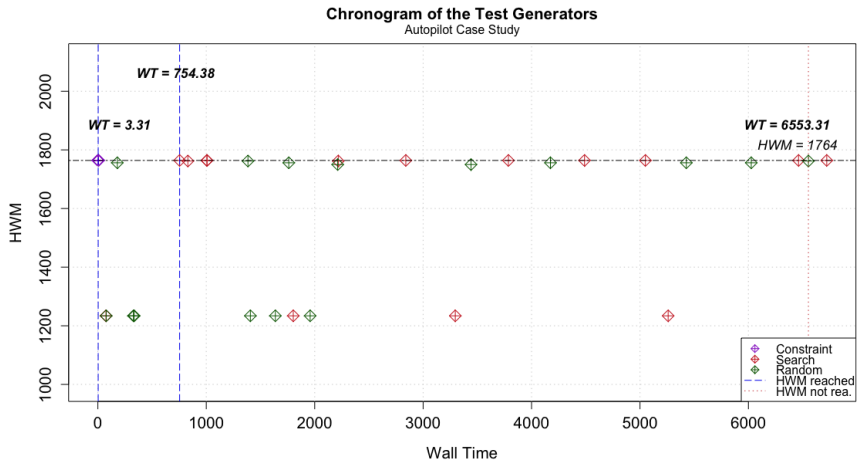


Figure 3.25: Chronogram. Wall time expressed in seconds. Stripped vertical lines denote the first observations of the largest execution times of each test generator.

Figure 3.25 displays a chronogram of the autopilot benchmark where the promptness of reaching the largest execution times of each test generator is exhibited. CBTG collects the HWM the first one in around 3 seconds and never underestimates this data. The second performing best is SBTG whose earliest collection of HWM is after 12 minutes and 30 seconds approximately. Lastly, RTG never actually hits the HWM.

Aside chronogram, boxplots in Figure 3.26 shows when the HWM is reached on the left and the rest of the largest execution times on the right. All these graphs show evidence for research question 3. As results testify, CBT approach rapidly reaches the HWM in contrast to SBTG. Additionally, SBTG only hits the HWM in 9 cases. In most cases, it takes on average around 3800 seconds (around 1 hour and 3 minutes). As unveiled in the chronogram, RTG never hits the HWM but the average of hitting its own largest execution time is close to 2000 seconds (around 33 minutes).

Because the described statistical tests require the same number of samples, this

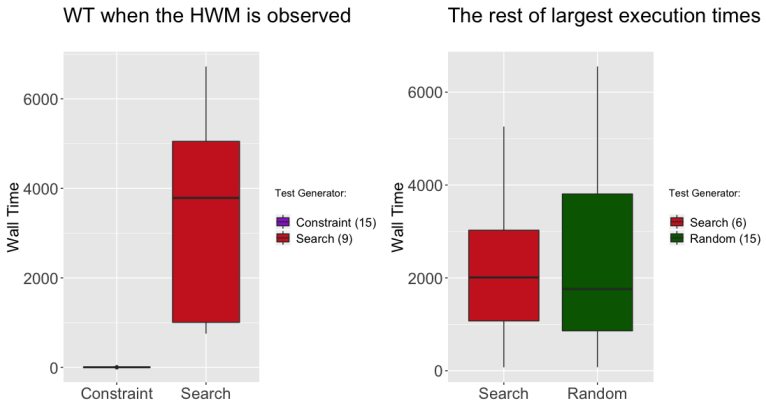


Figure 3.26: Boxplot of the observed largest execution times of the chronogram. Numbers in the parenthesis stand for the number of observations in each boxplot.

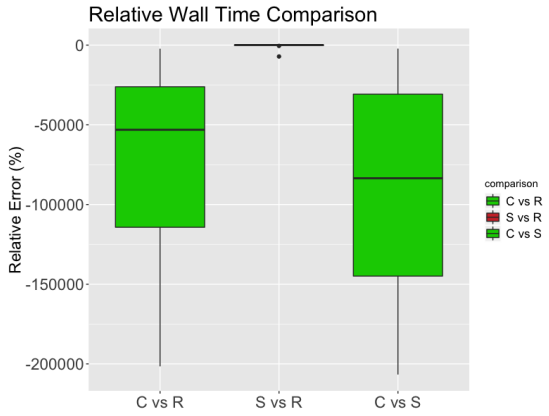


Figure 3.27: WNMT test applied to the wall time distributions.

separation is problematic for these tests. Despite so, integrating the 15 data of each trial WNMT test was applied and is depicted in Figure 3.27. The wall time of CBTG is always significant whereas the SBTG vs RTG is not. This difference between CBTG and SBTG is on average close to -500 times different. This difference is even greater (around -750 times between CBTG and RTG. By contrast, the difference between SBTG and RTG is close to 0% in the scale of the boxplot

When it comes to the efficiency, which is tackled in research question 4, Figure 3.28 exhibits the test vector generated per time unit. As expected, CBTG

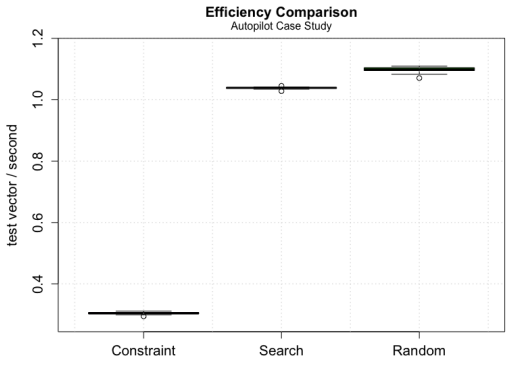


Figure 3.28: Efficiency results.

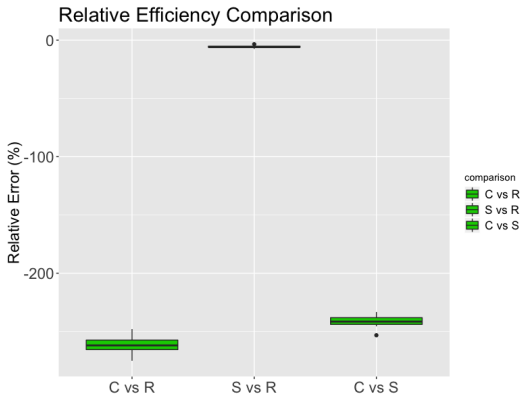


Figure 3.29: Comparison of the *difference* in efficiency distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

gives a low efficiency in comparison to SBTG and RTG. RTG results in the best performance which is not surprising. In spite of the fact that SBTG and RTG may have a similar efficiency in Figure 3.28, all methods show statistical significance in every combination according to the results displayed in Figure 3.29. On average, CBTG is -3.75 times smaller than RTG and around -3.3 times lower than SBTG. Conversely, the efficiency between RTG and SBTG is close to 0% with the scale of Figure 3.29.

The graphs depicted in Figures 3.30 and 3.31 display the data of the RAM usage. They aim to respond to research question 5. As expected, CBTG is more memory-consuming than the rest of test generators. In particular, it uses around 65% more memory than RTG and 55% more than SBTG. This is not surprising considering the amount of static processing both for path analysis and constraint solver. Again given its simplicity memory footprint RTG is the lowest one followed relatively closed by SBTG. All tests conclude that RAM usage are significant across test generators as reported in Figure 3.31.

When applying the program slicing the sliced version was 7% smaller than the

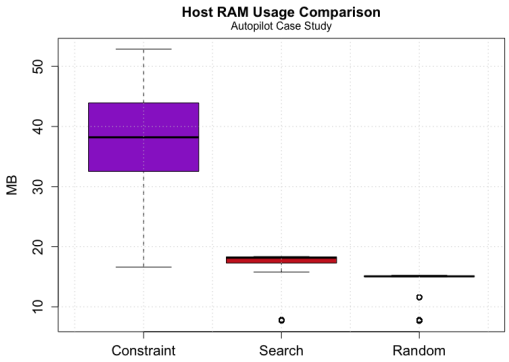


Figure 3.30: RAM usage.

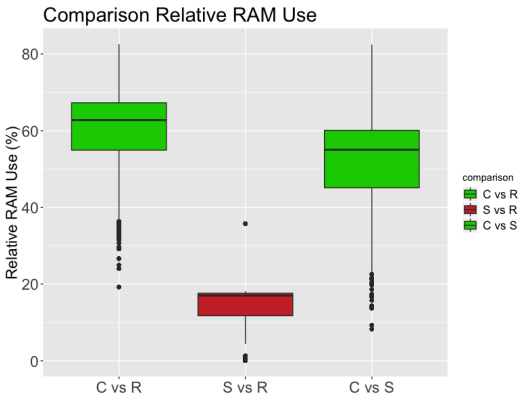


Figure 3.31: Comparison of the *difference* in RAM usage distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

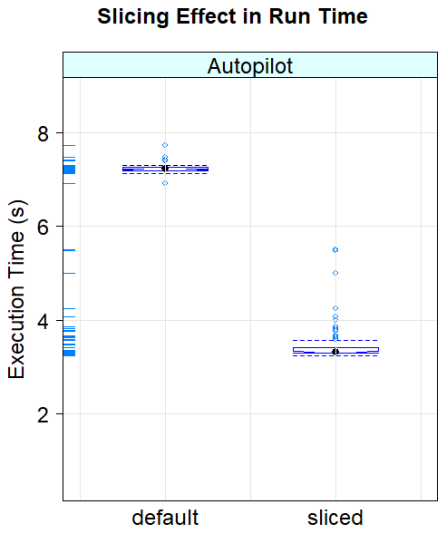


Figure 3.32: Slicing effect in the run-time of the CBTG

default one. The resulting execution time after running the described BFS is displayed in Figure 3.32, which aims for replying research question 6. On average, the difference of the execution time was around 55% smaller in the sliced version. Friedmann tests showed statistical significance for $\alpha = 0.05$. It is worth remembering here that our version of BFS performs a full of exploration of the

paths, unlike other versions of the same algorithm which stops after finding an optimal solution traversing one path.

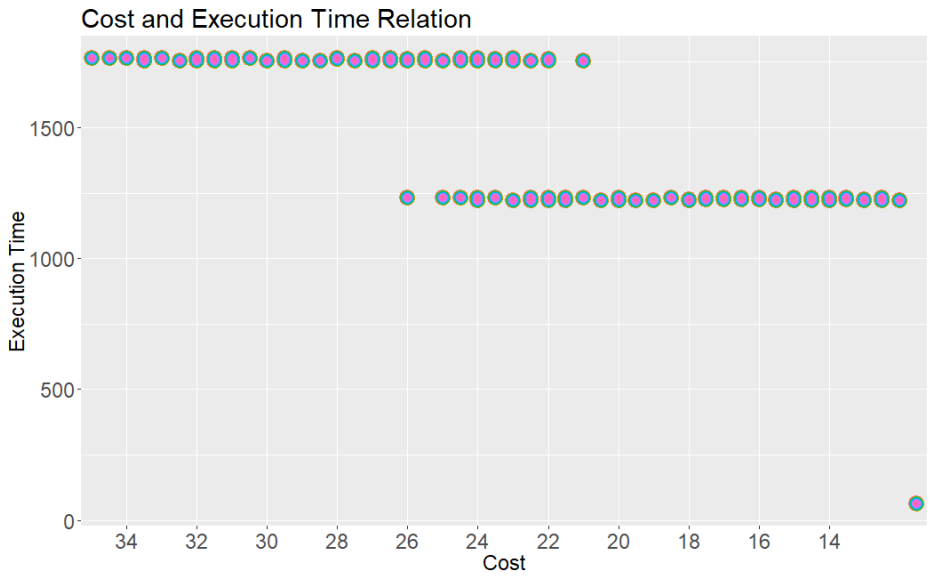


Figure 3.33: Cost and Execution Time. Each colour and shape denote a different trial.

Finally, the results of the cost and execution time, which are mentioned in research question 7, are displayed in Figure 3.33. This graph sets out to portray the degree of accuracy of the estimated cost w.r.t the actual execution time. A strongly correlated function of the cost would display a monotonically decreasing function with no overlap.

In this case the computed cost data displays some overlap in the middle area and, based on the data, this benchmark does not seem to offer a large number of different execution times. To evaluate the correlation more accurately we applied the Pearson correlation coefficient whose values are between -1 and 1. These values indicate, -1 anti-correlation, 0 no correlation and 1 correlation.

In the case of the Figure 3.33, the value of this coefficient is only 0.17. Therefore, it is not safe conclude that the cost function accurately estimates the execution

time.

3.4.3 Certyflie

The third case study examines a flight controller from a software called Certyflie [99] which is applied actually to the drone on Figure 3.34. This software was picked as it is a representative benchmark for a flight control software and its technical aspects such as board libraries and programming language was compatible with GenI. Despite so, some similar changes to previous case study were made to the software to achieve full compatibility with our framework and contribute to the validity of the experiment.

Figure 3.34: Crazyflie drone running Certyflie software. Source [101]

The SUT consists of PID (Proportional Integral Derivate) controller with 4 inputs: Pitch, Roll, Yaw and Thrust [94] to control the flight. GenI reported that this benchmark 2594 paths from which only 197 were feasible (7.59%). The reader can map the rest of the graphs presented in this and the others case studies to research questions from the mapping in previous case study.

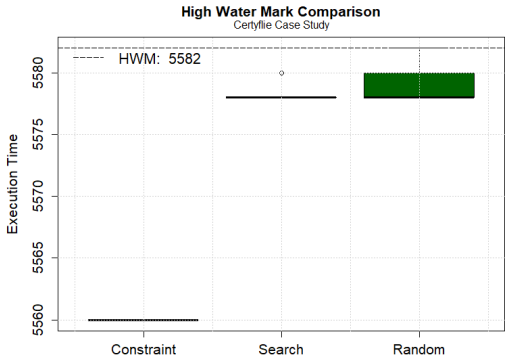


Figure 3.35: HWM Boxplot Comparison

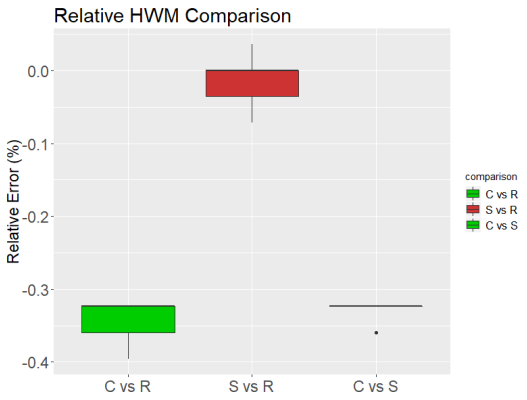


Figure 3.36: Comparison of the *difference* in HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

The firsts results are displayed in Figures 3.35 and 3.36. Figure 3.35 illustrates how the HWM is only achieved by the RTG which shows a similar performance to the SBTG. This idea is later confirmed by the WNMT test showing no statistical significance between SBTG and RTG. By contrast, CBTG did not spot the HWM. In spite of the fact that the error of the difference in the HWMs is relatively small according to Figure 3.36 the statistical test argues that there is a significant difference between the CBTG and the other two.

After inspecting the code along with the execution time results we drew the conclusion that the reason why CBTG did not perform as well as the other two generators is because there was some arithmetic code for the output of the motors to which mere coverage test vector did not suffice to unveil its local HWM. By contrast, SBTG and RTG generated more convenient test vectors which in turn generated operands that maximized the execution time of these code more successfully.

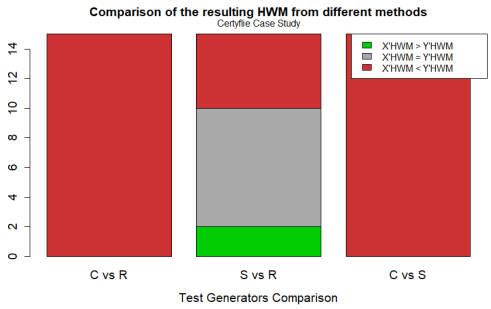


Figure 3.37: HWM Comparison

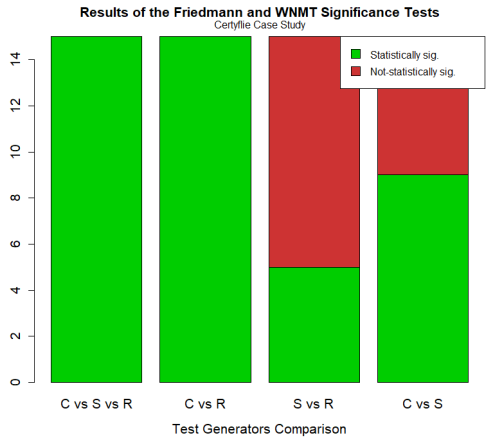


Figure 3.38: Friedmann and WNMT Test results.

Figure 3.37 displays the data comparison of the HWM where only SBTG and RTG methods unveiled some different HWMs. Statistical tests on Figure 3.38 concludes that the three methods were different according the Friedmann test. WNMT test contends that the CBTG results are normally significant different for RTG and SBTG. However it also estimates that in 10 out 15 cases results of the SBTG and RTG are similar and thus not statistically significant.

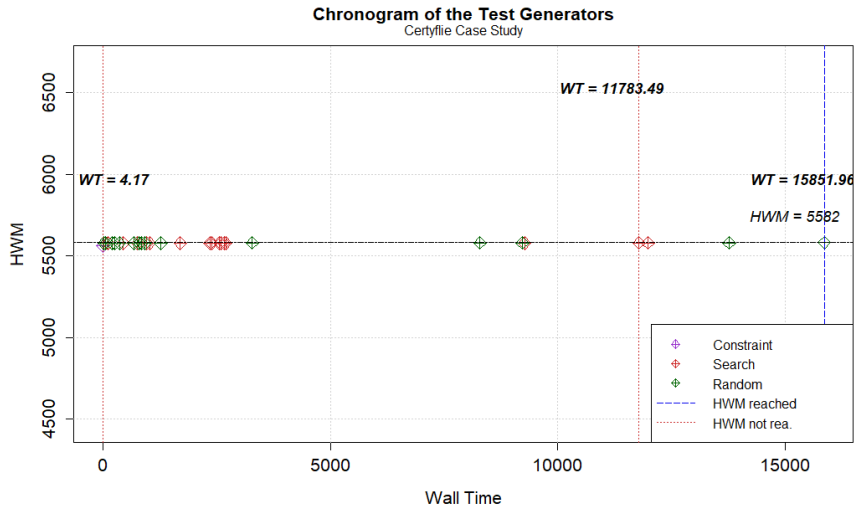


Figure 3.39: Chronogram. Wall time expressed in seconds. Stripped vertical lines denote the first observations of the largest execution times of each test generator.

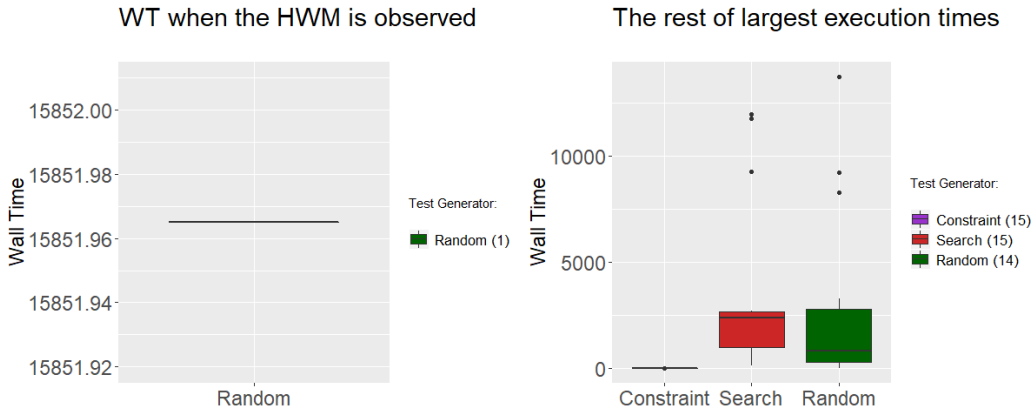


Figure 3.40: WNMT test applied to the wall time distributions

Another interesting feature of this case study is the result of the wall time, which are depicted on Figures 3.39 and 3.40. The HWM unveiled only for the RTG is attained at the latest time, around 4 hours and 24 minutes. The local HWM of the SBTG was only 0.0358% smaller than the one unveiled by the RTG and it was spotted around 3 hours and 16 minutes. The local HWM was spotted around 4 seconds and is 0.39% smaller than the global HWM.

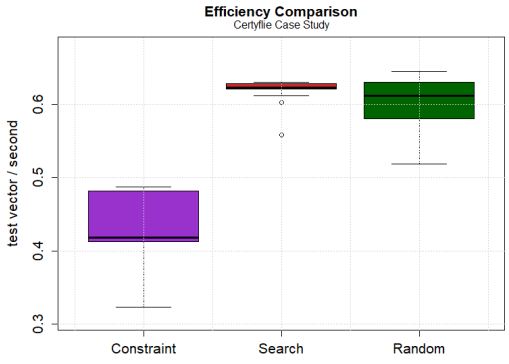


Figure 3.41: Efficiency results.

Figure 3.42: Comparison of the *difference* in efficiency distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Figures 3.41 and 3.42 are concerned with showing data for the efficiency. In summary, the efficiency of the SBTG and RTG are similar to the extent of not showing statistical significance on Figure 3.42. As expected, CBTG exhibits a significantly worse efficiency with respect the other two TGs but this difference is smaller in comparison to the previous case study. An explanation for this observation could be the computational load as a result of the constraints and input data.

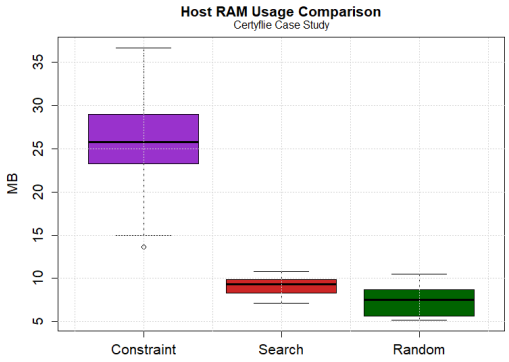


Figure 3.43: RAM usage.

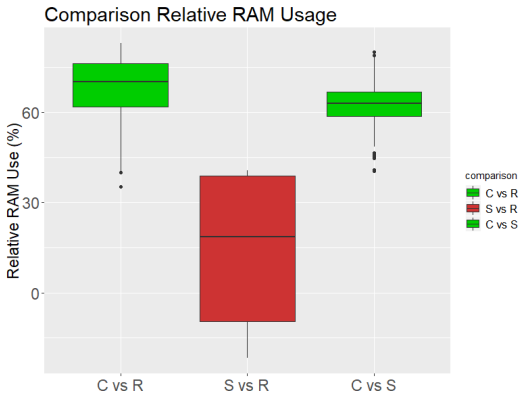


Figure 3.44: Comparison of the *difference* in RAM usage distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Figures 3.43 and 3.44 displays data of the wall time. The results are as expected. CBTG shows a statistically significant usage of RAM memory on account the static analysis in place whereas SBTG and RTG show an equivalent consumption which is not concluded to be statistically significant.

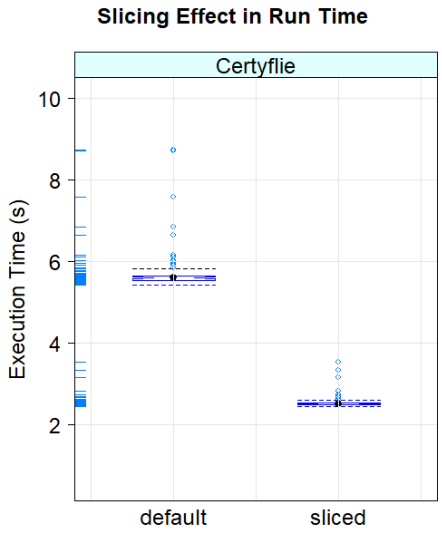


Figure 3.45: Slicing effect in the run-time of the CBTG.

Figure 3.45 displays the effect of the slicing on the run time. In this case, the effect of the slicing removed around 22% which statements that had to do with code interfacing with hardware units. It had an impact of around 62% in the run-time. Friedmann test concluded that such a difference was statistically significant.

Figure 3.46 depicts the data of the cost and actual execution time. This time the first and greatest cost did not match the local HWM achieved by the TG. Data shows how the same cost triggered different execution time. However there is a slight trend of proportionality between the cost and the execution time.

Pearson correlation coefficient returned a value of 0.28 which concludes there is a poor correlation in the data.

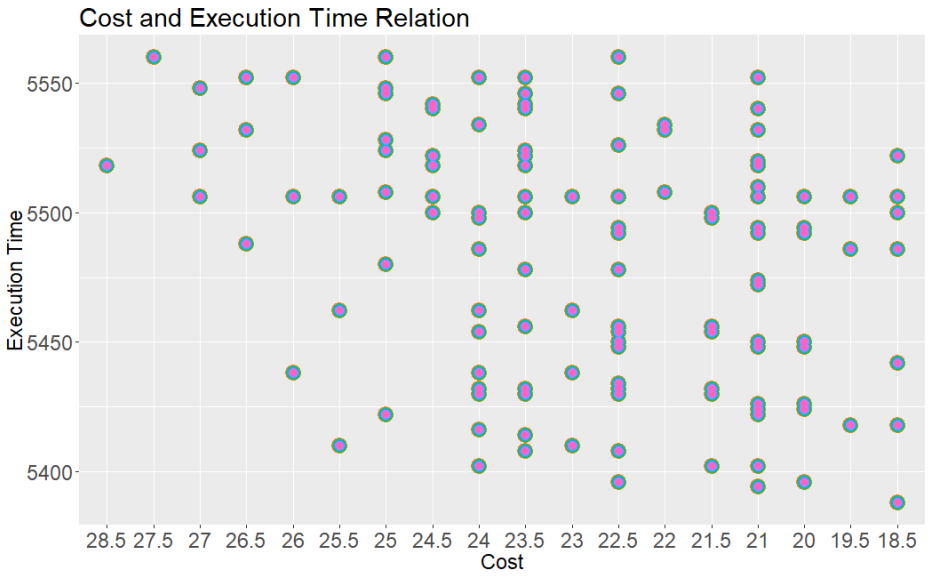


Figure 3.46: Cost and Execution Time. Each colour and shape denote a different trial.

3.4.4 RC Car

The fourth and last example of this chapter tackles a Real-Time software applied to control a Radio Controlled (RC) car which is depicted on Figure 3.47. The reason for choosing this software is to evaluate a representative Real-Time software applied in a car vehicle.

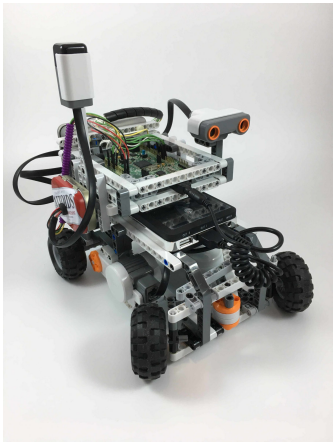


Figure 3.47: RC Car running Robotics With Ada Software. Source [102].

This vehicle implemented a Robotics With Ada library which can be found at [100]. From this software, we analyzed a control function which accepts control commands, it checks for a collision detection and performs the appropriate control actions. This benchmark contained 6 input data from which 5 were enumerates indicating states and 1 integer indicating the speed. GenI reported this benchmark contained 109 paths from which 21 were feasible (19.26%).

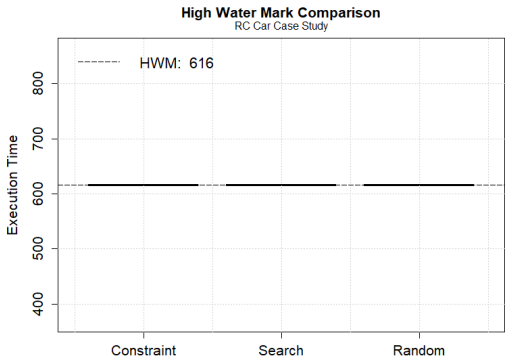


Figure 3.48: HWM Boxplot Comparison

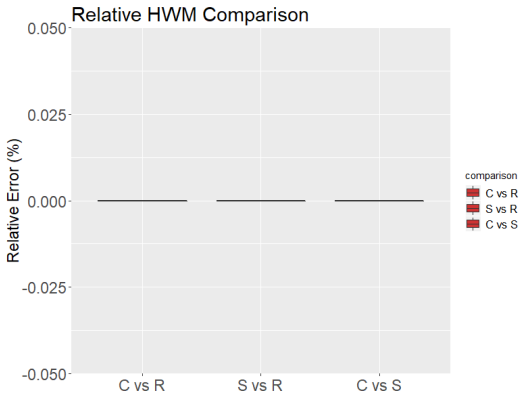


Figure 3.49: Comparison of the *difference* in HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

When it comes to the HWM results, these are depicted in Figures 3.48, 3.49 and 3.50. The HWM spotted is the same in all TGs and cases rendering no statistical significance as a consequence according to the statistical results in Figure 3.49.

Perhaps the most interesting results are displayed in Figure 3.51 where the methods are claimed to be significantly different in 9 out 15 cases according to Friedmann test. However, CBTG is mostly claimed not to be significantly different to RTG in 13 out 15 trials and similarly to SBTG in 11 cases. Similarly, SBTG and RTG are only concluded to give significantly different results in 7 cases.

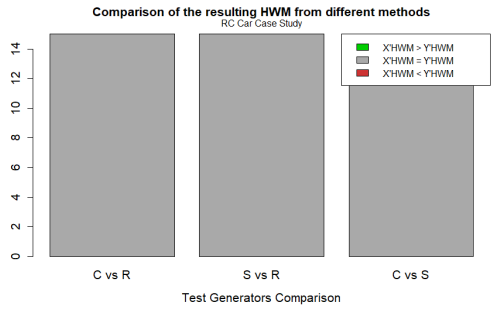


Figure 3.50: HWM Comparison

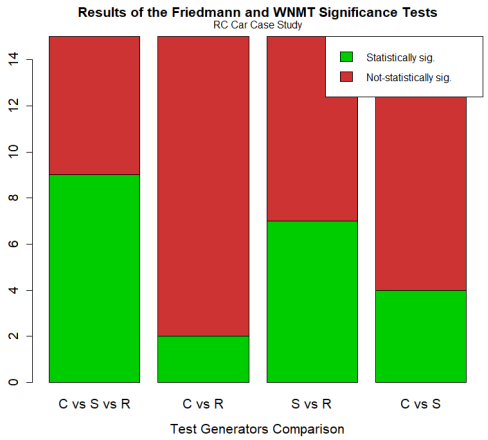


Figure 3.51: Friedmann and WNMT Test results.

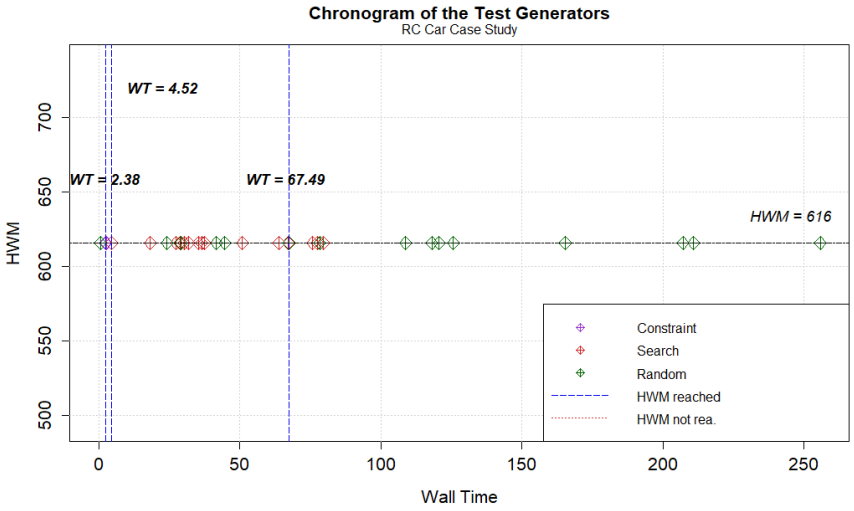


Figure 3.52: Chronogram. Wall time expressed in seconds. Stripped vertical lines denote the first observations of the largest execution times of each test generator.

Regarding the wall time, results are depicted in the Figure 3.52, where the first HWM is spotted by the CBTG at only 2.38 seconds followed by SBTG which spotted it at 4.52 in the best case. RTG took a minute and 7 seconds to achieve the same results. The resulting distributions of the wall time are displayed in Figure 3.53. Results show how CBTG performed the best. The wall times generated by the CBTG were concluded to be statistically significant by the WNMT

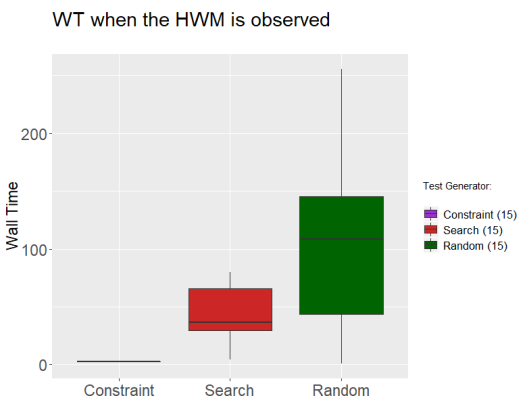


Figure 3.53: Wall time distribution.

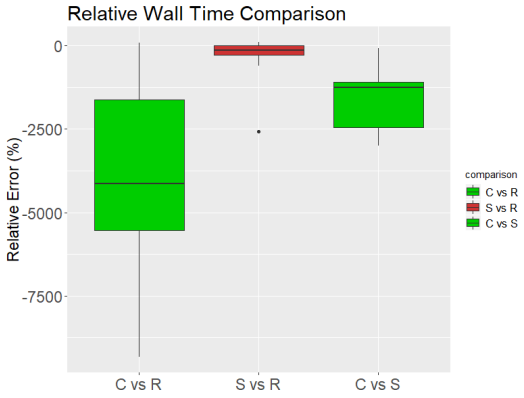


Figure 3.54: WNMT Test results to wall time.

test. However, the same data was not claimed to be significantly different for the SBTG and RTG.

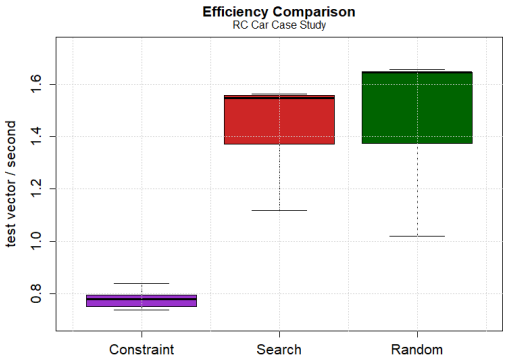


Figure 3.55: Efficiency results.

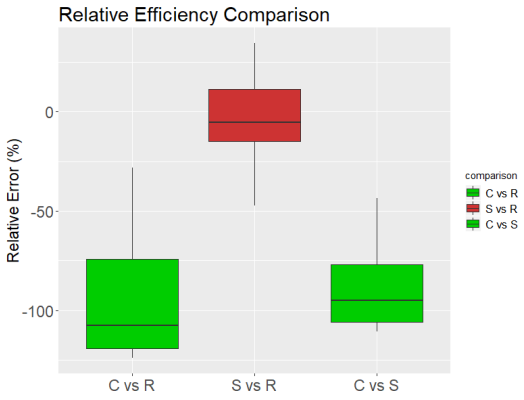


Figure 3.56: Comparison of the *difference* in efficiency distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Results of the efficiency are depicted in Figures 3.55 and 3.56. CBTG showed a significantly worse efficiency - around 2 times worse - than SBTG and RTG. By contrast, SBTG and RTG which exhibited a similar efficiency with around -7% difference.

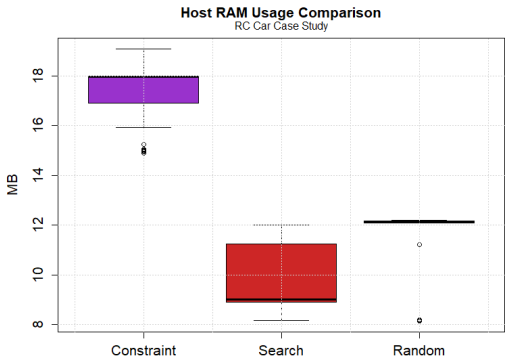


Figure 3.57: RAM usage.

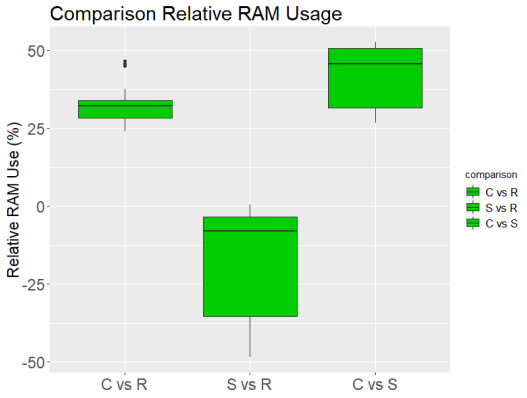


Figure 3.58: Comparison of the *difference* in RAM usage distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Figures 3.57 and 3.58 depicts the RAM memory usage. Once more, CBTG employed the greatest amount of RAM memory because of the static analysis. However this time, SBTG and RTG showed a significantly different memory usage. We believe the underlying motivation for this result was some sort of inactivation to free the space of the dynamic memory as SBTG and RTG may have had some memory dependencies between their successive execution.

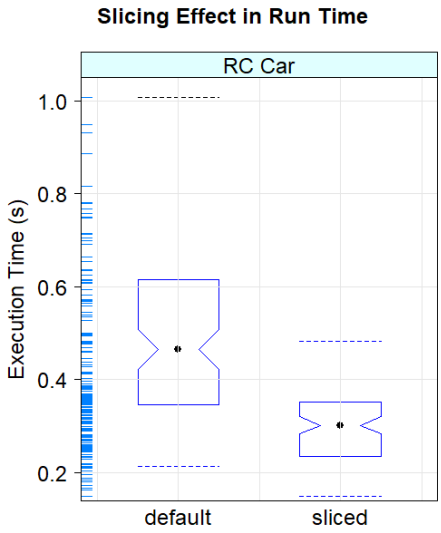


Figure 3.59: Slicing effect in the run-time of the CBTG

Figure 3.60 depicts the results of the slicing. This time only 5.8% of the slicing was removed which incurred in reducing the execution time around a 33%. Friedmann test concluded that this reduction was statistically significant.

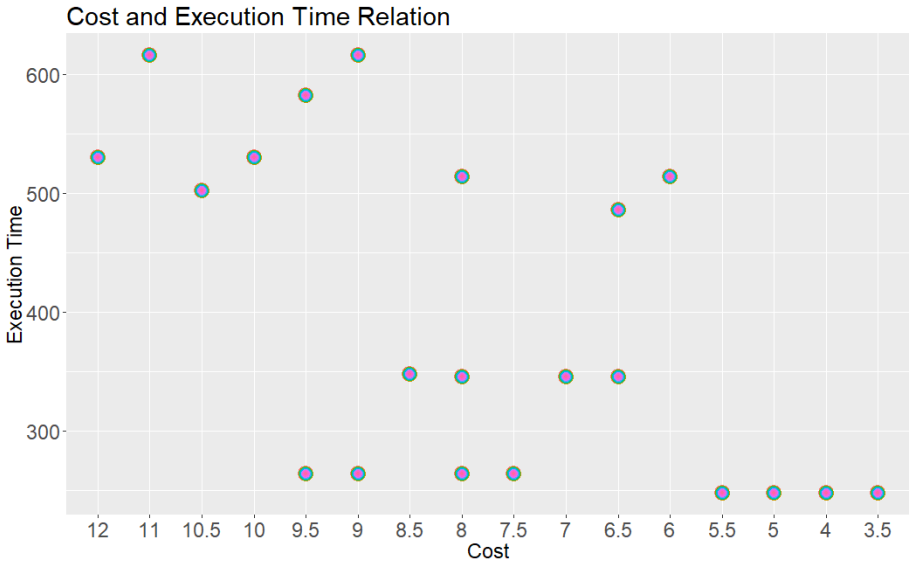


Figure 3.60: Cost and Execution Time. Each colour and shape denote a different trial.

Lastly, Figure 3.60 portrays the cost and execution time relationship. Whereas, the first two estimation of the cost were relatively reasonable the data show an asymmetric and inaccurate pattern.

Still, the Pearson correlation coefficient returned a 0.6 value suggesting that there is a slight correlation despite the apparent dispersion of points in the Figure 3.60.

3.4.5 Threats to validity

Threats to validity normally become apparent in any empirical study and the one presented is not an exception to the rule. Two strands of threats can be laid out [63]:

- **Threats to internal validity:** One potential bias comes from the reproduction of the state-of-the art in SBT. In the case of the Needle-in-Haystack, there exists some existing work in SBT [58] which could arguably trigger branches like the one presented by using testability transformations. The programming language employed in this work impose severe restrictions in the compilation check which may render the testability transformations not applicable. These testability transformation also violates the principle *"the system you analyze is the system you deploy"* [19] often used Safety-Critical Systems. Nonetheless, based on the efficiency results in [58] we would argue that the conclusions of the efficiency would hold.

  The second threat comes from measuring the run-time of program slicing in the host computer. As the execution time is highly influenced by the hardware, we believe that taking into account the hardware architecture of the host computer with several levels of cache may add a substantial measurement noise to the run-time results. Particularly, when the effects to be measured has a run-time close to 0.

  The third threat comes from the features of the statistical tests which require the same number of observation for the evaluation. This is a conflictive requirement as the CBTG normally generates a lower number of

observations. As a consequence, a small number of observations can be analyzed which negatively impacts the confidence of the conclusions for research question 2.

Lastly, a threat stems from a potential dependence at preserving data in memory when running sequentially the SBTG and RTG. Such a effect becomes apparent at evaluating the impact of memory RAM usage as argued in RC Car case study.

- **Threats to external validity** to other embedded targets or benchmarks include:

  One threat comes from the benchmarks employed which are open source. Despite the fact industrial benchmark are representative examples of Real-Time Systems, their availability is highly restricted. Their complexity and features may differ from the examples provided. As a consequence, caution is advised when extrapolating these conclusions.

  Aside that, the assumption of achieving path coverage as a way to compensate for the deficiencies of the BFS heuristic on account of the inaccurate cost function, may not be applicable in all cases because a) we cannot always achieve path coverage and b) we may have other sources for underestimation. The latest include arithmetic operations as discussed in Certyflie case study or hardware features such as caches which create some dependencies about how the paths are executed.

  The second threat comes from the embedded architecture employed. While the one used is a relatively simple and time-predictable, industry is moving toward multicore chips which places a significant challenge for timing analysis. The cost function as defined here is most likely not comparable as the source code does not contain information on events such as bus in-

terferences which could inflate the HWM very significantly [103].

A hard to spot limitation of our approach stem from the way the path constraints are built. Programs containing functions with multiple return statements may incur in a inconsistent path constraints. Having a single return statement is a reasonable assumption for Spark programs [15] but not necessarily for C programs. In addition, input data such a pointers, or elaborate data structures are not claimed to be compatible with our approach.

Lastly, an existing threat comes from the programming language used in the benchmarks i.e., Spark and Ada. A striking feature of this language is the frequent declaration of tailored data types for the implementation. As a result, the range of the user-defined data types is frequently smaller than primitive data types. This feature gives a competitive advantage to SBT and RT as the search space is frequently reduced. In other words, the results of the evaluation may not be applicable to other C-based benchmarks often used in embedded systems as they often use data types with a larger range. However, the CBTG methods described here could be perfectly applicable.

3.5 Summary

This chapter was advocated to introduce some contributions in the realm of CBT and evaluate their impact. The first contribution consisted of an optimal program slicing. The objective of this process is to synthesize only the parts of the program that can be tested and maximize the chance of obtaining a useful test vector i.e., maximizing the effectiveness of the CBTG, produced by a constraint solver. Unlike state-of-the-art approaches, constraint solvers are not involved at selecting the statements to be sliced and some heuristics are used.

The proposed heuristics map the input data and the controlling predicates of the flow constructs in order to determine which statements to collect. In the event

of not using this slicing and include all the statements of the default program the effectiveness of the CBTG would be damaged and their effects would be a) reducing the number of test vectors produced as a consequence of the infeasibility and b) inexistent variables of the test vector would be produced with a plausible unsound value steming from the default range of variables of the solver.

The resulting graph representation of the sliced program is later traversed by a tailored BFS algorithm. Unlike state-of-the art approaches, this algorithm targets, in the first place, what is assumed to be the most promising paths so as to trigger the largest execution times leading to the HWM. This prioritization of paths targets the efficiency of the overall approach as ideally the HWM - which is the key data of interest for the WCET - would be collected sooner than state-of-the-art approaches.

The price to pay to implement this heuristic is the inclusion of the notion of *cost*. In essence, this cost is calculated by counting the number of statements of the original program. The evaluation of the accuracy of this cost consisted of simple block-based benchmarks having different operations of load/store, input/output and arithmetic and yet the same estimated cost. The results of the experiments has led to the conclusion that the cost function is *inaccurate*.

As the program slicing argues about the efficiency of the resulting graph exploration, the effects of the run-time using the sliced and default program are included as part of the case studies. The results show how the slicing typically removed 7%, 22% and 5.8% of the uncontrollable or redundant statements. This removal corresponded to code that read data from hardware units or consisted of internal arrays which stored states for control functions. The effect on the run-time had a statistically significant impact of 55%, 62% and 33% on average respectively. Such results are generated after applying the BFS to generate the path constraints. These results show how even small amount of slicing can have a significant impact in the overall search efficiency.

The latest part of the chapter was concerned with the comparison of the resulting

techniques w.r.t state-of-the-art SBT and RT. Four case studies were addressed. All of them exhibited statically collectible constraints and path coverage could be achieved. In addition, the last three were more advocated to be representative examples of Real-Time System including a navigation system of an autopilot, a PID controller of a drone and a control function of a radio controlled-car. Their most striking results could be synthesized as:

1. **Needle in a Haystack** case study has shown how a pathological case for SBTG and RTG can be easily handled by CBTG which has given the best results at unveiling the global HWM. However some existing techniques for SBTG i.e., testability transformations, might provide better results by creating a more convenient guidance for these cases.

2. **The Autopilot** case study has not exhibited much difference in terms of the HWM but it has shown the benefits in the wall time of our CBTG, which obtained the results the earliest. The low proportion of feasible paths i.e., around 7% of all paths, has reduced the number of testable paths. In addition, the predicates of the branches were reachable by every test generator.

3. **Certyflie** case study shows how only RTG and SBTG unveiled the HWM. The interesting bit is how the success of attaining HWM may also depend upon arithmetic operation inside some blocks of code to which the CBTG is oblivious and consequently fails to generate operands that trigger larger execution times for these blocks of code.

4. **RC Car** case study has demonstrated how all test generators are able to produce the same HWM. However, the interesting result is that our CBTG was able to achieve this data earlier i.e., wall-time, than the other two counterparts. Unlike SBTG - which normally needs several iterations to reach the objective - the CBTG does not need an iterative process and it makes the most of the accuracy of constraint solvers.

In the last two cases the cost function was not shown to be very accurate yet this hypothesis was already falsified.

Lastly, the threats to validity argue, amongst other things, about: the partial reproduction of the state-of-the-art in SBTG, the limitation of the experiments to open-source benchmarks, the relatively simple embedded architecture employed and the competitive advantage of SBTG and RTG as a result of the programming language employed which normally uses a reduced data types range.

Chapter 4

Dynamic Constraints Analysis and Infeasible Path Detection

Along with facing path explosion, one of the greatest challenges to embrace CBT is obtaining the constraints from the SUT. A limitation of the program slicing described in the former chapter is its inability to attain constraint values which are generated *dynamically* by reading the source code. Collecting these values would entail either *parsing* the source code, *running* the program or applying elaborate *static analysis techniques*.

The other assumption in the former chapter is the achievement of path coverage as part of the testing process which is not normally true. This chapter is concerned with reviewing and including *dynamic* constraint collection as well evaluating the performance of the CBTG when path coverage can not be generally achieved.

With respect to the evaluation, our goal is to examine the behaviour of the CBTG when dynamic variables are to be collected and a path explosion occurs. To meet this objective three Mälardalen benchmarks [104] - which are popular benchmarks for WCET analysis - are taken as case studies. Apart from sorting algorithms, search algorithms also showed some interesting features for the WCET analysis and they have been also employed in similar research works [16]. For this reason, linear search and binary search were included. An additional constraint has been

imposed so that the element to be found must be in the search space. The reason for that is that normally search algorithms reveal their worst-case performance when the element to be searched is not in the search space.

Additionally, a hash-function benchmark was also included as it is a relevant problem in symbolic execution.
Coincidentally, the benchmarks of the previous chapters have integer and enumerates as input data whereas these benchmarks have fixed point and integers.

Some of these benchmarks - particularly the sort routines - write on the test vector by swapping the elements which create a mismatch between the constraints added as global input to and the ones collected in the constraints. This chapter discusses the solution applied in GenI for this problem.

Another upside of the proposed CBT approach is that some good side effects become apparent when applying it. In particular I) the potential reduction of pessimism in path-based estimations of CWCET that are not equipped with feasible path detection and II) To gain some *observability* as part of matching the constraints solved with the branches or loops in the code.

In a nutshell the contributions of this chapter are:

(a) Contribution 3 from Section 2.5 on showing the cost and execution time relationship.

(b) Contribution 4 from Section 2.5 on evaluating the effects of the slicing in the run-time in those cases where it is possible.

(c) Contribution from Section 2.5 on assessing the TGs in general and CBTG in particular when path coverage can not necessarily be achieved, and dynamic constraints need to be collected.

(d) Contribution 5 from Section 2.5 on detecting infeasible paths of CWCET to reduce the pessimism.

(e) To provide a case study of the former contribution.

The rest of the chapter is organized as follows. Section 4.1 discusses the challenges of non-static constraints. Section 4.2 provides 6 case studies advocated to show contribution (a), (b) and (c) Section 4.3 outlines contributions (d) and (e) chronologically. Finally, Section 4.4 summarizes the content of this chapter.

4.1 Constraint Collection of Dynamic Values

In order to understand the challenge of applying CBT to programs with dynamic constraints consider Listing 4.1.

```
1   for I in A'First + 1 .. A'Last loop
2
3      Value := A (I);
4
5      J := I - 1;
6
7      while J >= A'First and then A (J) > Value loop
8
9         A (J + 1) := A (J);
10
11        J := J - 1;
12
13     end loop;
14
15     A (J + 1) := Value;
16
17  end loop;
```

Listing 4.1: Insertion Sort benchmark with Ada syntax.

Listing 4.1 displays the central loop of the Insertion Sort procedure from Mälardalen benchmarks [104]. According to the premises of our testing protocol the innermost loop should be exercised so the only controllable branch must be collected i.e., *A(J) > A(I)* after replacing *Value*. Unfortunately, *the indexes of the array is subject to values that change on each iteration.*

According to the literature surveyed in Chapter 2, there exists some **Static Analysis** techniques to derive variables whose value is dynamic:

- Pure Abstract Interpretation [42] which may only offer approximate values.
- Enhanced abstract interpretation like the method developed by Lokuciejewski et al. [11] which also uses *Ehrhart polynomial* to calculate loop indexes. The latter offer more precision and efficiency than the first one.
- Static Symbolic Execution [95] which has shown its limitation when the constraints to analyze are beyond the theory of the constraint solver.

As an alternative, Dynamic Symbolic Execution [30] - also called *Concolic Testing* [61] or - have gained momentum because of its ability to collect the required values from the actual executions. Another variant is known as *Directed Automated Random Testing* which combines model checking methods so as to test *all* feasible paths. However, this approach struggles with scalability due its handling path explosion [95].

Concrete Value Execution is often used as part of Dynamic Symbolic Execution [95] so as to collect concrete values of a constraint rather than symbolic values since the last ones can be *imprecise*. To record these values the SUT is normally instrument and a random test vector is provided to start with. By embracing this technique hard-to-analyze functions, such as hash functions, are treated as a black box [30] increasing the effectiveness of the CBTG.

In our framework, *concrete value execution* has been embraced for the sake of simplicity. However, it is worth noting that the actual execution is done in the host computer and not on-target. The other difference with Dynamic Symbolic Execution is that we are not using symbolic execution as we are not using symbols to replace input data. Instead, we are using concrete values collected from the instrumentation. The path construction is not necessarily built by negating path constraints but applying BFS strategy described in the previous chapter.

Hence, in our framework we manually insert instrumentation points in a equivalent program of the SUT to collect the required data. An obvious downside

of this approach is that in the absence of automatic code translator we have to find a way to integrate the concrete value execution with GenI framework. This entails to rewrite the code.

4.1.1 Keeping Consistency in the Test Vector

The other troublesome aspect from Listing 4.1 is the fact that the test vector is not only *read* but also *written*. This may cause inconsistencies when reading the constraints of the test vector after being written since the collected index may refer to another index in the original test vector. To solve this issue we need a trace mechanism that is able to compute sound values and is called every time a value in the test vector is swapped. The following algorithms analyzes swap function used in *quick sort* and *insertion sort*.

Algorithm 16 Global Variables.

```
1: input_variable_written_set ← ∅          ▷empty set
2: input_writing_history ← ∅               ▷Vector of hash maps
```

Algorithm 16 displays global variables used for these algorithms. First one consists of a set which indicates what indexes were recorded. The reason for having this check is to cover the corner case when asking for an index that has not been written and thus the collected index is correct. In addition, *input_writing_history* holds a map of all the records of index swapping.

Algorithm 17 records the index of the variables swapped in a temporary map. Next, this map is added to the written set of variables and the history map. This algorithm must be called every time a swap is executed and is not claimed to work beyond the scope of swapping elements of a test vector.

Finally, Algorithm 18 seeks in the record history the initial and correct value of the index variable to withdraw a sound variable. If it happens that the variable is not written in the history (first if statement) the same index is returned. This

Algorithm 17 Procedure to record the swapping of the input vector.

```
1: procedure RECORD_NEW_DEFINITION_IN_TEST_VECTOR(index1, index2)
2:
3:     if index1 ≠ index2 then
4:         index_map ← ∅                                  ▷map of indexes of vector
5:         index_map[index1] ← index2
6:         index_map[index2] ← index1
7:         input_variable_written_set ← input_variable_written_set ∪ index1
8:                                                 ∪ index2
9:         input_writing_history.push_back (index_map)
10:    end if
11: end procedure
```

Algorithm 18 to get sound indexes when a constraint data is recorded.

```
1: function GET_ACTUAL_INDEX(recorded_index)
2:     if recorded_index ∉ input_variable_written_set then
3:         return recorded_index
4:     else
5:         sought_index ← recorded_index
6:         for all map ∈ input_writing_history do
7:             if map.key = sought_index then
8:                 sought_index ← map[sought_index] ▷Entry of the sought_index
9:             end if
10:        end for
11:        return sought_index
12:    end if
13: end function
```

function is called every time a constraint is added or read, in a instrumentation point and contains at least one dynamic variable that can be subject to exchange.

In summary, as a result of the swap, the indexes of the test vector and their contents are exchanged rendering the initial declaration of global and unsound input. The Algorithm 17 holds the history of the swaps whereas Algorithm 18 provides the sound indices by looking at the record of the previous algorithm in order to be added to the constraints.

4.2 Evaluation and Case Studies

This section is dedicated to present 6 case studies with the same research questions as Section 3.4. The case studies are advocated to provide a broad subset of examples often used in WCET analysis and their internal structure and type of dynamic constraints vary.

Three case studies are from the Mälardalen benchmarks which were chosen by Law and Bate [18] to show the advantages of the SBT. From the benchmarks presented in [18] only *qurt* is omitted. The reason for that exclusion is that such a benchmark would entail adding polynomials constraints. Even though [90] constraint solver is able to cope with them, our GenI framework only supports linear constraints. Adding polynomial constraints would entail adding a switch functionality between linear and polynomial which in turn would entail more time-consuming software development for our GenI framework.

An advantage of using these benchmarks is that it is claimed that the test vector provided triggers the WCET path. In order to evaluate how our test generators can maximize the HWM in comparison with the assumed test vector achieving so, the following research question is considered.

- **Research Question 8 - Effect of the test vector from the Mälardalen benchmarks in the HWM.** Is the default test vector from the Mälardalen benchmarks able to hit the global HWM? This effect is compared against the HWMs triggered by the other test generators.

Along with sorting, which are considered in the above benchmarks, search algorithms are instrumental in computer science. For these reason two popular algorithms are included in the cases studies. These algorithms have also being used in similar test generation works for WCET [16]. The requirement that the element to be searched is in the search space because has been imposed since, intuitively speaking, search algorithms normally hit the WCET when they have exhausted the search space or reached the stop criterion [27].

Lastly, hash functions [95] set an example of problematic functions where Dynamic Symbolic Execution has made a positive impact by treating them as a black box [30]. They can be considered another example where dynamic constraints are necessary and our approach could deal with them.

Because the SUT was translated and instrumented statically for the collection of dynamic values, the harvest of constraints can only be performed as long as the test vector reaches the instrumentation point. Thus, different path trees may have different costs. To maximize the coverage and maximize the chance of generating different path trees *random test vectors* were fed in the concrete execution every x iterations.

The benchmarks employed are:

1. **Select k largest**, picks the number of an unsorted numeric vector that is kth if it were sorted. It contains three levels of loop nesting and the number of iterations is defined within certain decisions inside the loops.

2. **Quick Sort** is a popular sort algorithm that exhibits the worst-case time complexity when the test vector is already sorted [96]. There exists iterative and recursive versions. We used the recursive one as is the only example of this type in this work. In addition, to our knowledge, C-written real-time systems support recursion even though its behaviour should be bounded to enable WCET analysis [10].

3. **Insertion Sort** sorts the test vector in an iterative fashion rather than using recursion.

4. **Linear Search** aims for identifying an element in an array by analyzing every element from beginning to end.

5. **Binary Search** has the same objective as the search before but it subsequently halves the search space by assuming that the array is sorted.

6. **Hash Function** implements an example of this modular function which is often used in command parser or language processors [30].

The test vector size of the first 5 benchmarks has a direct impact on the path explosion. For this reason, its original size was reduced to the maximum accepted by the CBTG. The default test vectors size were adapted accordingly, though in the Select k Largest an experiment was carried out with the actual size of the benchmark. Given that originally the Mälardalen were executed in the sliced version and hit the memory limit because of the exponential complexity of the BFS, the non-sliced default version could no be included. By contrast, in the last three benchmarks the effect of the slicing in the run-time is included.

4.2.1 Select K Largest

The first one of the case studies is perhaps the most challenging given the control flow it exhibits. Controllable parts of the code are only encountered inside the loop. In particular, only the nested branches and the third most nested loop are controllable. To add more difficulty, it contains a great deal of infeasible paths e.g., it has two consecutive branches with the decisions: $j \leq k$, $j \geq k$.

To assess research question 8 we employed SBTG or RTG for comparison. To do so, the actual size of the test vector (20 items plus k variable) was preserved and the benchmark was translated to Ada. As expected, due to the path explosion of BFS that had to be stored in the CBTG this program ended up out of memory for a test vector of this size. Hence, CBTG could not be considered for the default test vector size. That's why only SBTG and RTG took part in this experiment.

Results are illustrated in Figure 4.1. In this figure, we can observe that SBTG identifies a test vector that slightly outperform the one provided by the Mälardalen benchmark and sets an example about how difficult to predict execution time is alluding to the code only. The underestimation is only around -0.53%. In conclusion, given that SBTG outperforms the HWM generated by the default test vector, it can be claimed that this input data does not guarantee to trigger neither the WCET nor the global HWM.

To make the benchmark analyzable for the CBTG the size of the test vector

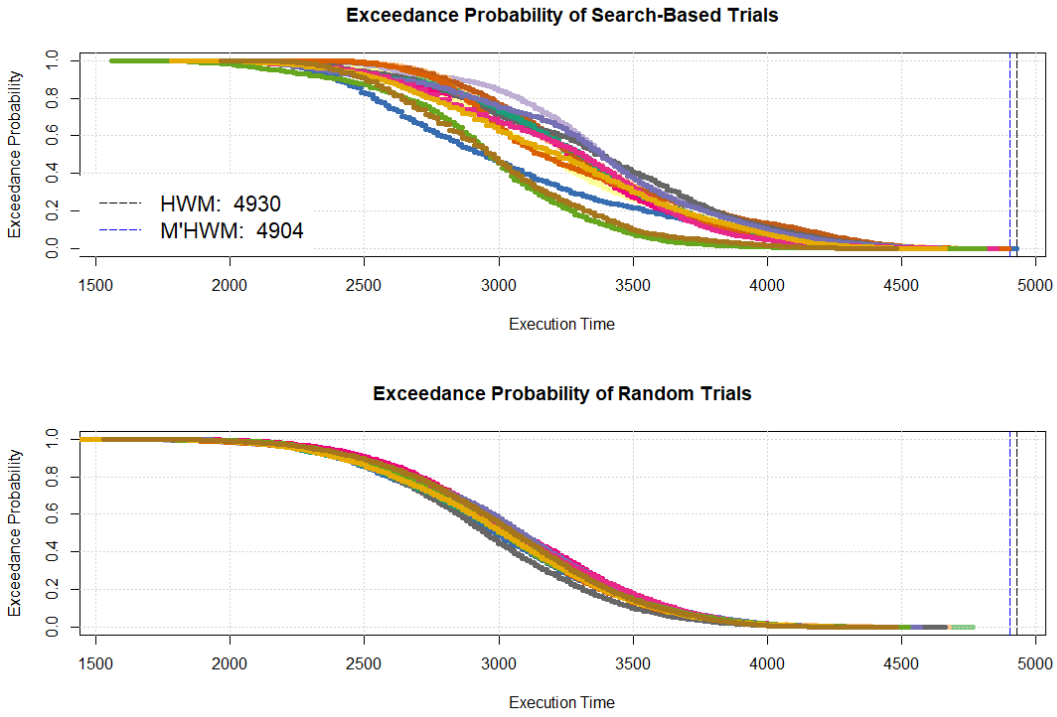


Figure 4.1: Execution time profile of Select K Largest benchmark tested by SBTG and RTG with its original test vector size. M'HWM stands for the execution time generated by the reportedly test vector triggering the WCET path.

was reduced to 9 elements plus k one. The execution time profile expressed as exceedance probability is portrayed in Figure 4.2. Unfortunately, even RTG outperforms CBTG in terms of HWM. This extreme data is only attained by SBTG.

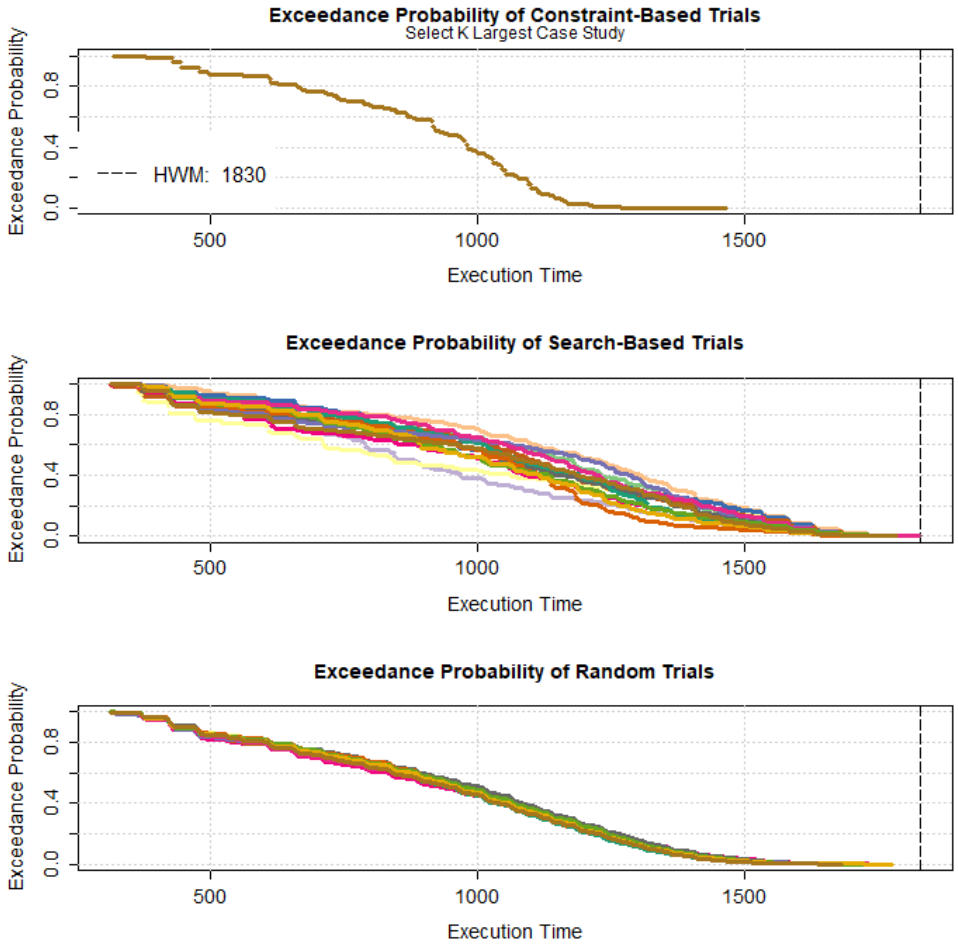


Figure 4.2: Execution time profile of last trial in Select_K_Largest after applying different test generators

The differences in terms of HWM are better expressed in Figures 4.3 with its attached statistical significance in Figure 4.4. As depicted in Figure 4.3 only SBTG hits the HWM followed by RTG performing the best. By looking at the relative error comparison and statistical significance in Figure 4.4, we see that CBTG underestimates on average around -17.5% the local HWM of the RTG and an average around -21%. The local HWMs of the SBTG are around 3% greater than the ones unveiled by the RTG. Statistical tests conclude that only the combination SBTG vs RTG is equivalent in terms HWM results.

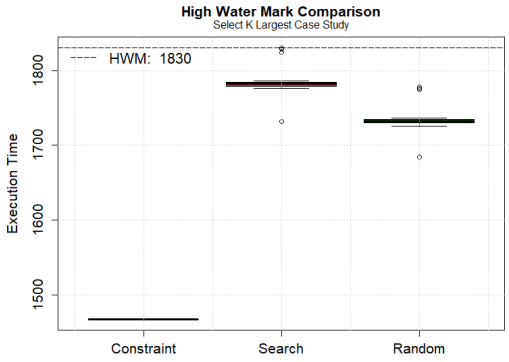


Figure 4.3: HWM of each TG and trial. Stripped line stands for the HWM.

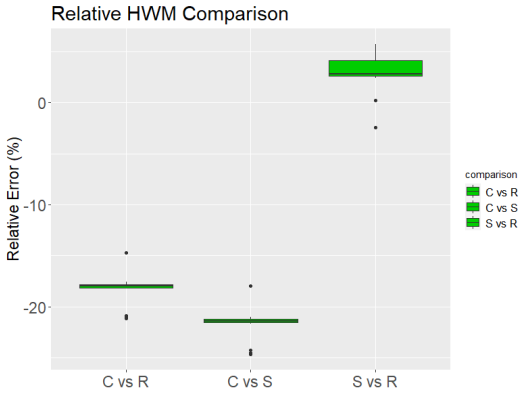


Figure 4.4: Comparison of the *difference* in HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the opposite

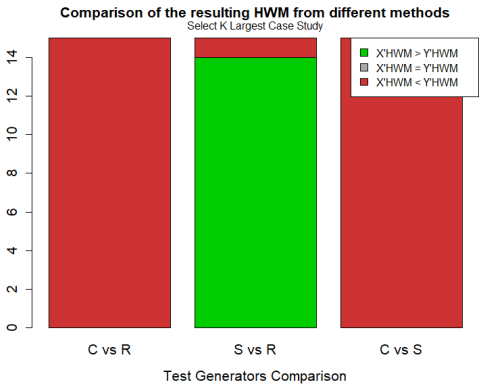


Figure 4.5: Statistical and HWM results summary.

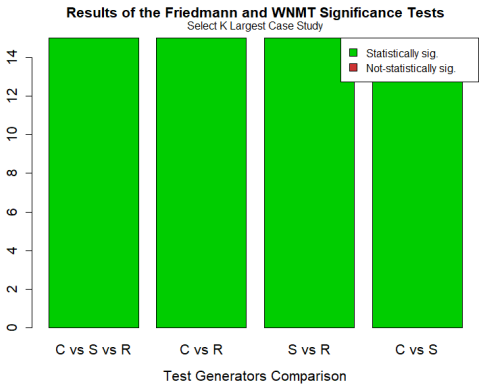


Figure 4.6: Results of the statistical significance delivered by WNMT test.

By looking more specifically at the comparison of the HWMs in Figure 4.5 we get that CBTG results are always upper-bounded by its counterparts whereas SBTG brings about the best results that are only surpassed by RTG in one trial.

When it comes to the statistical significance of the overall execution times, results are depicted in Figure 4.6. All three methods are significantly different in all cases according to the Friedmann and WNTM test.

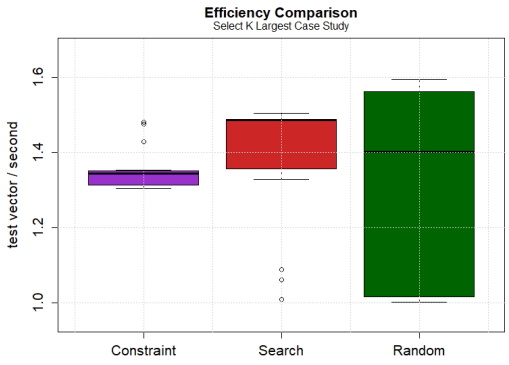


Figure 4.7: Test vectors generated per time unit

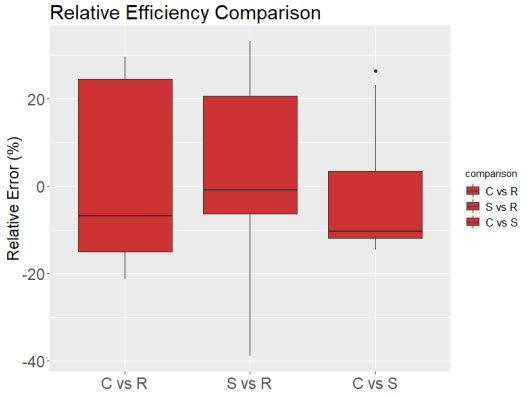


Figure 4.8: Comparison of the *difference* in efficiency distribution of previous plot.

Results of the efficiency are illustrated in Figure 4.7. This time, the CBTG is not concluded to have a significantly different efficiency but is comparable to the ones from RTG and SBTG. The reason for that is that is that the path tree contained a wide number of feasible paths.

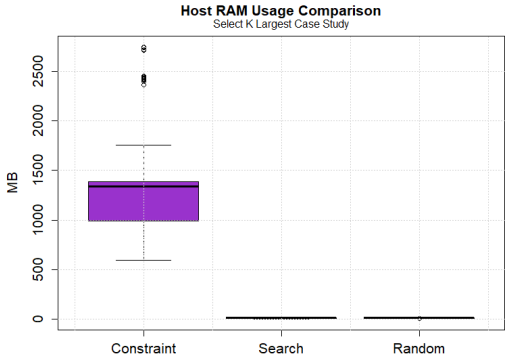


Figure 4.9: Memory usage.

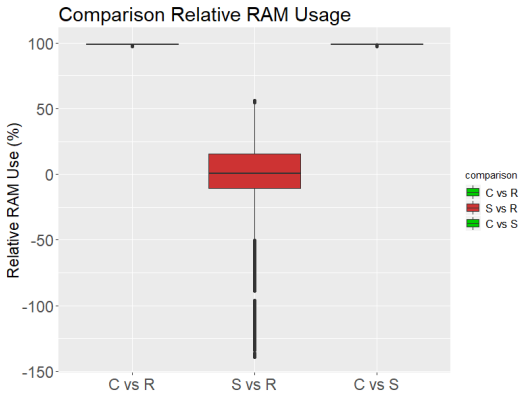


Figure 4.10: Comparison of the *difference* in RAM use distribution of previous plot.

RAM memory use results are displayed in Figure 4.9. Again, as expected CBTG uses a lot more memory than the rest of the TGs. The relative error presents a

huge difference of around 2 times of the RAM consumption amongst CBTG and the other two testing methods. Statistical significance results are presented in Figure 4.10 where the SBTG and RTG are concluded to use a relatively similar amount of RAM.

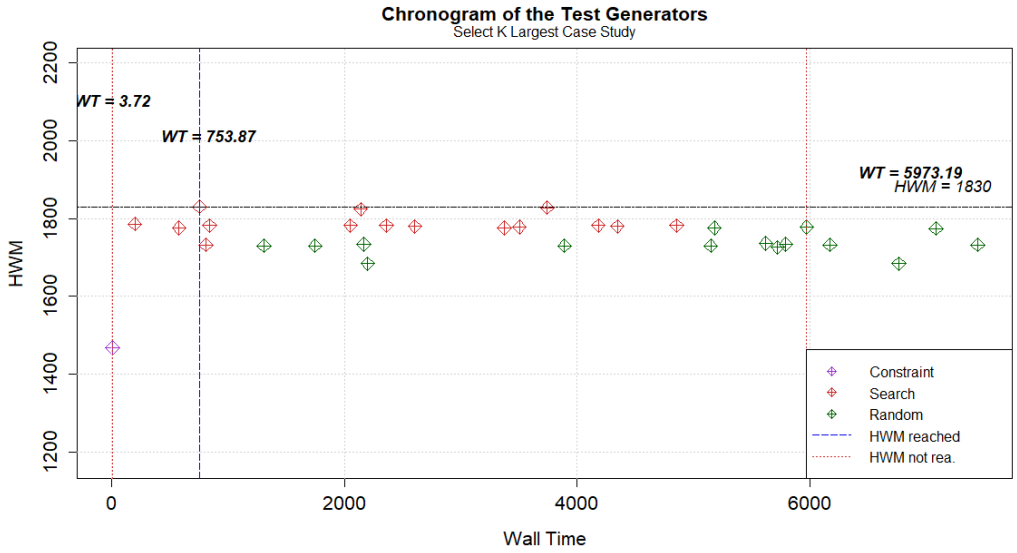


Figure 4.11: Chronogram

To have a better overview about the optimality of this case study a chronogram is depicted in Figure 4.11. SBTG hits the HWM in less that 13 minutes. RTG spots its local HWM around 1 hour and 40 minutes, a very large time comparison with the other TGs.

The wall time results are illustrated in Figure 4.12. In only one session of the SBTG the HWM is observed whereas the rest of the sessions and test generators are unable to observe such a data.

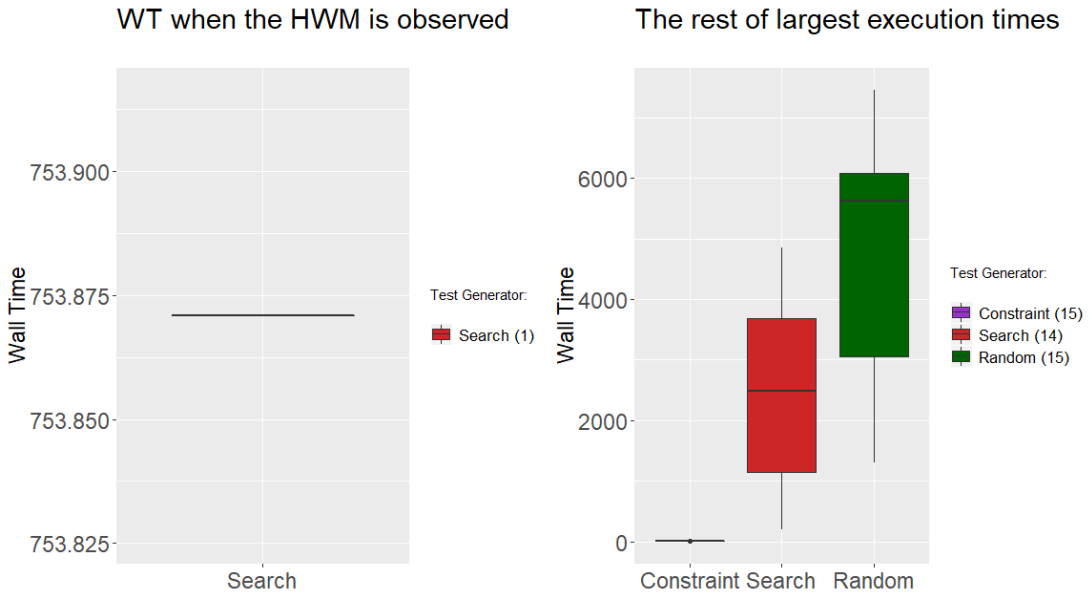


Figure 4.12: Wall Time

Figure 4.13: Cost and execution times. Each colour and shape denote a different trial.

Lastly, the cost and actual execution time relation is depicted in Figure 4.13. Clearly, the cloud of points suggests that the cost estimation is not accurate.

This was later corroborated by the Pearson correlation coefficient which returned a value of 0.06, the lowest correlation of all case studies. Presumably, the two split cluster of data with a different trend contributed to such a low score.

4.2.2 Quick Sort

The second of the case studies implements the popular sort algorithm *quick sort* in its recursive version. This algorithm exhibits a $O(n\ log_2(n))$ algorithmic complexity when sorting an unsorted test vector and $O(n^2)$ otherwise [96]. The test vector of this benchmark was reduced to 9 elements to allow the CBTG execution.

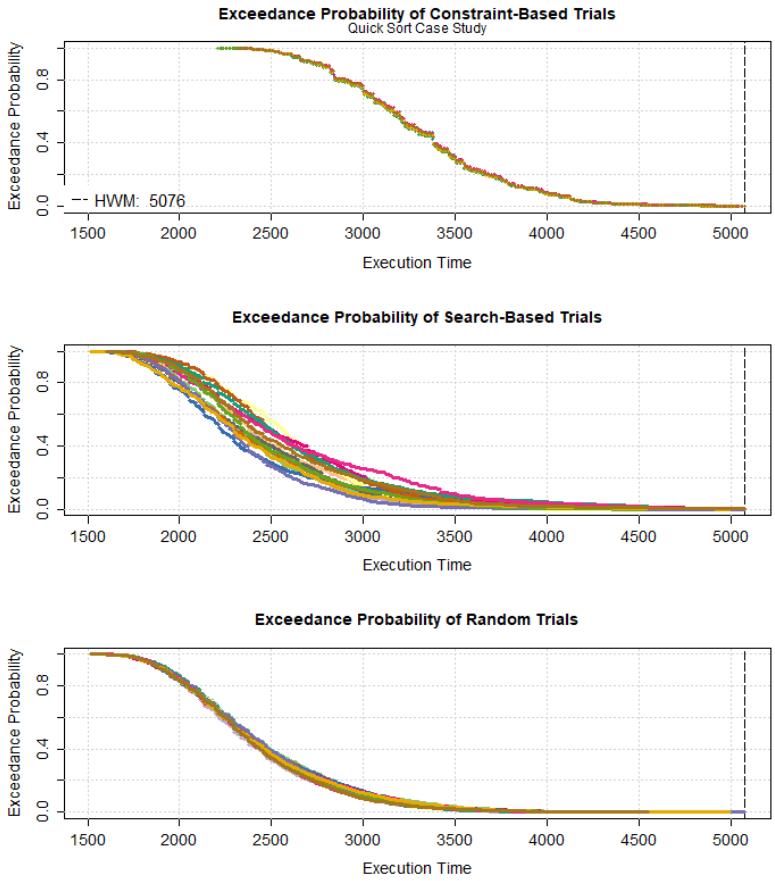


Figure 4.14: Execution time profile each trial corresponding to a different color.

Firstly, in terms of execution time profile the empirical exceedance probability is portrayed in Figure 4.14. All test generators succeed at finding the HWM and the test vector provided in its original Mälardalen benchmark hits the HWM as well. Thus, responding to our research question 8 we can conclude that the test vector from Mälardalen benchmark gives sound results.

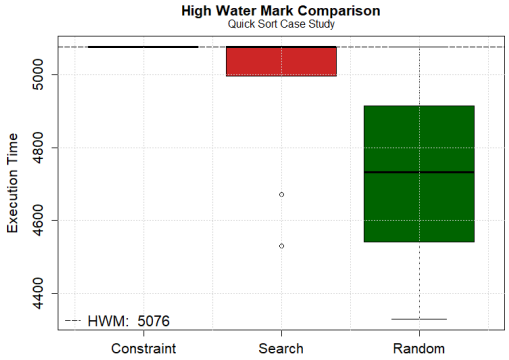


Figure 4.15: HWM boxplot

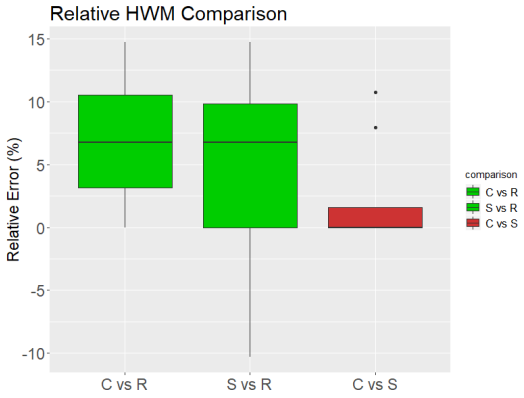


Figure 4.16: Comparison of the *difference* in the HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the opposite.

More accurately, HWM results are displayed in Figures 4.15 and 4.16. By all accounts, the CBTG always hit the HWM whereas SBTG does so in most cases and RTG in not-many. In terms of a statistical significance, WNMT test results in Figure 4.16 demonstrate that only the results of HWM of CBTG vs SBTG are not significant. On average, the HWMs of the CBTG and SBTG are around a 7% greater than RTG and RTG whereas SBTG is close to 0% greater than RTG.

By looking at the comparison of the HWM in Figure 4.17 we get that RTG is normally outperformed by CBTG and SBTG. There are only 2 cases where the HWM of the RTG is greater than the SBTG. CBTG method and SBTG draw in 9 out 15 cases as they collect the same HWM whereas in 6 session CBTG outperforms SBTG. Figure 4.18 displays results the statistical significance of the

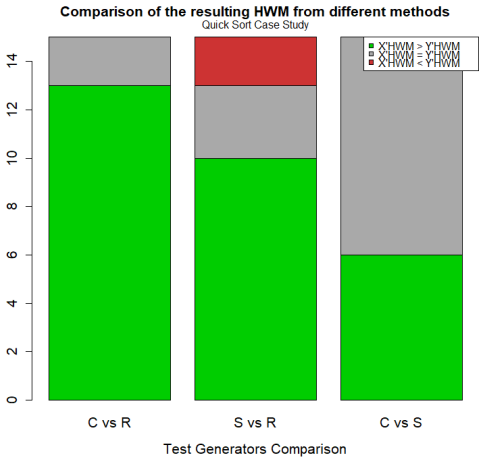


Figure 4.17: HWM Comparison

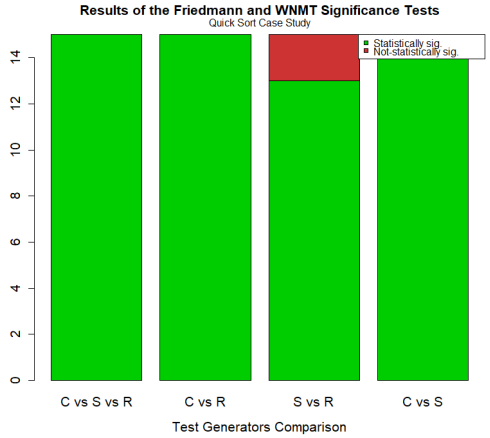


Figure 4.18: Comparison of the *difference* in efficiency distribution of previous plot.

difference of the execution times. In the first column, the three methods are concluded to be different according to the Friedmann test. As for the rest of them, only in 2 out of 15 trials, the comparison of SBTG vs RTG are claimed to be similar.

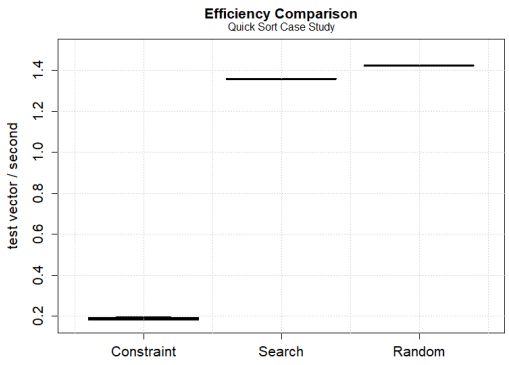


Figure 4.19: Efficiency results

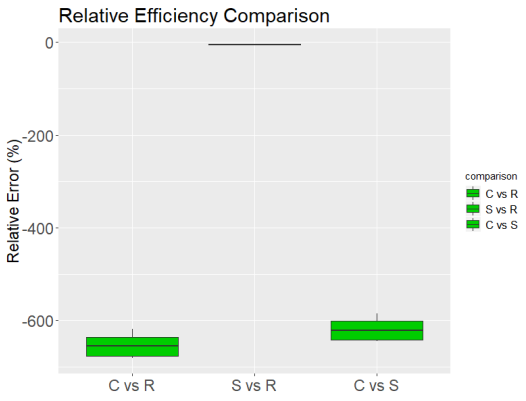


Figure 4.20: Comparison of the *difference* in efficiency distribution of previous plot.

When it comes to the results of the efficiency, these are portrayed in Figure 4.19

and its statistical significance in Figure 4.20. Results shows the common pattern so far where RTG is the most efficient followed closely by SBTG and lastly CBTG. Perhaps the only subtlety that becomes apparent is the extreme low efficiency of the CBTG. The underlying reason is the great number of infeasible paths in the SUT. Regarding the significant difference and relative error of the efficiency all plausible pairs of TGs are concluded to be different according to the results in Figure 4.20. Unlike, previous case study the efficiency of the CBTG is around -7.5 times smaller than the RTG and around -7.25 times smaller than the SBTG. The difference in efficiency of SBTG and RTG are close to 0% yet it is claimed to be significant by the WNMT test.

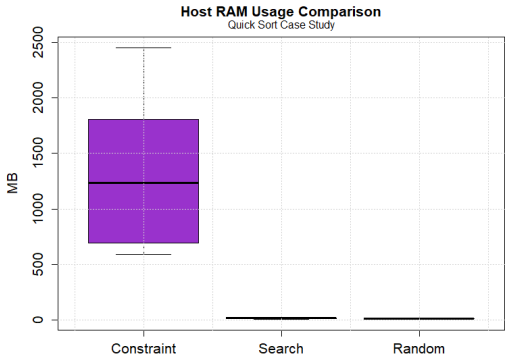


Figure 4.21: Memory usage.

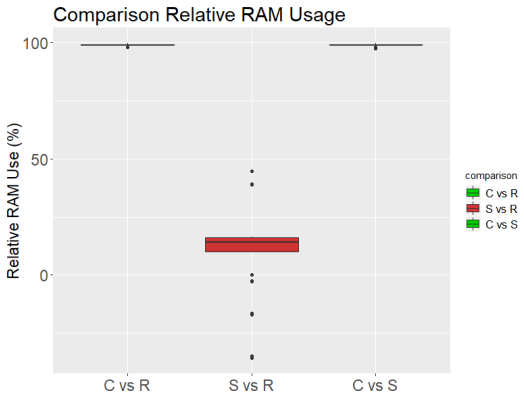


Figure 4.22: Comparison of the *difference* in RAM usage distribution of previous plot.

Resulting data on RAM usage are depicted in Figures 4.21 and 4.22. They clearly show that CBTG uses far more RAM than the other two TGs. In significance terms - as showed in Figure 4.22 - only SBTG and RTG are claimed to have a similar consumption of RAM with an average around 15%. Similar to previous case study, the RAM usage is around 2 times greater for CBTG with respect SBTG and RTG.

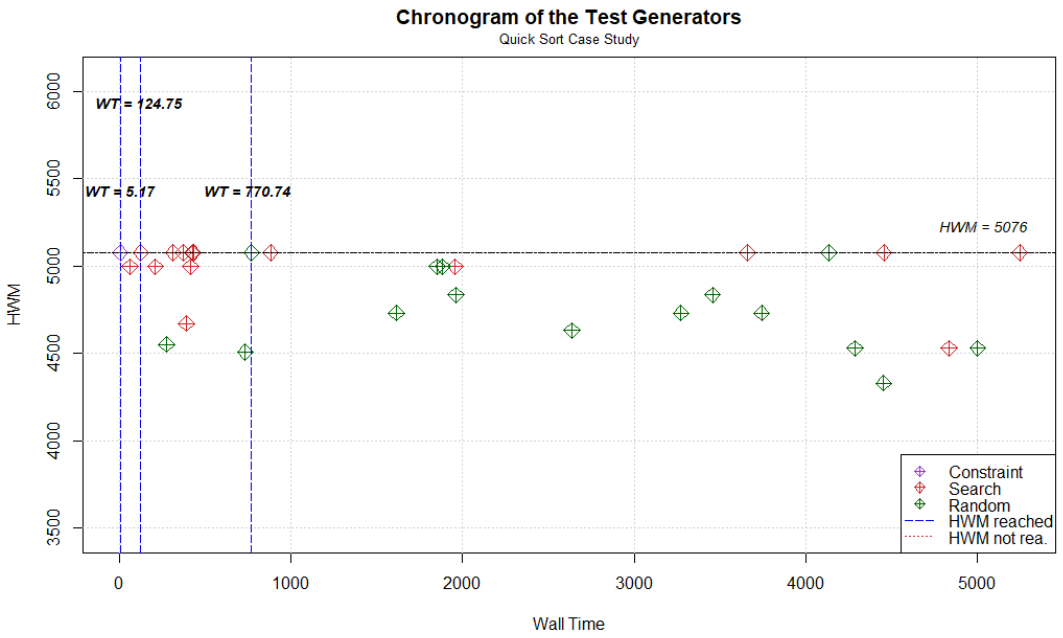


Figure 4.23: Chronogram

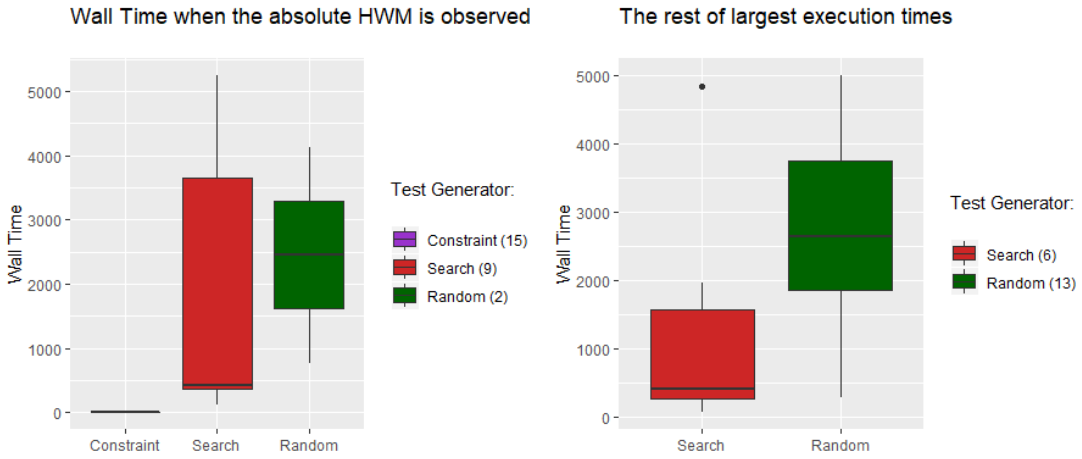


Figure 4.24: Wall Time Boxplot.

Chronogram data is depicted in Figure 4.23. The HWM is first hit by CBTG in 5.17 seconds the earliest followed by SBTG which collects in 2 minutes and 4.75 seconds. RTG perform generally the worst but it reaches the HWM in two occasion the first one takes around 12 minutes and 51 seconds. Last but not least,

the wall time when the HWM and local HWM is hit is depicted in Figure 4.24. All the trials of the CBTG hit the HWM and the earliest whereas 9 out of 15 trials of SBTG with a relatively low average close to 8 minutes and 20 seconds (500 seconds) of wall time. RTG only reaches the HWM in 2 out 15 trials in around 41 minutes (close to 2500 seconds).

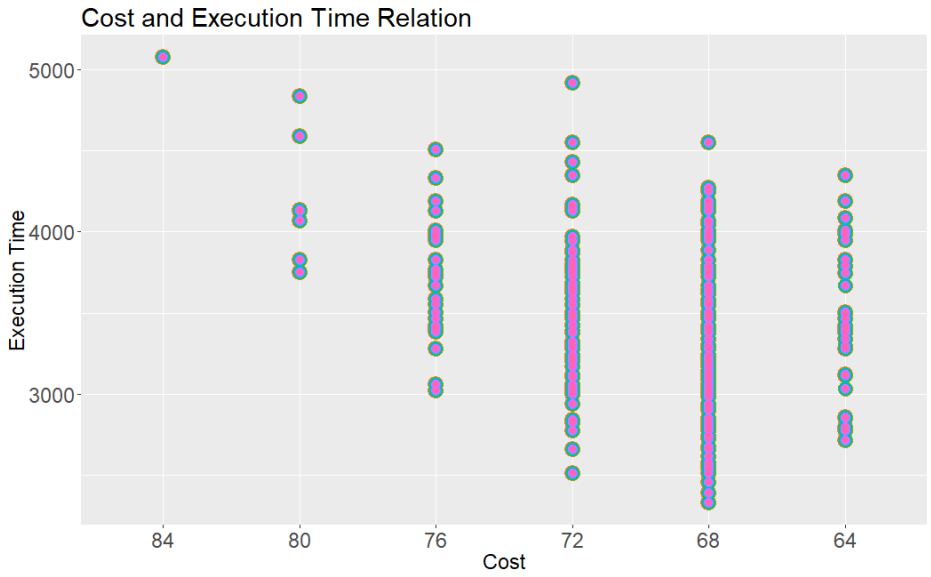


Figure 4.25: Cost and execution time.

Finally, Figure 4.25 displays the relation between the cost and execution time. Starting from the top left, we can see how the cost function estimates well the greatest cost for the HWM. However, toward the middle we can contemplate a great degree of overlap in the cost estimation - particularly 72 and 68 - which generated a wide array of execution times. The cost function seems to be slightly correlated between 84 and 68 because of the relative decreasing pattern. Yet, the last 64 estimation seems to break this trend.

To evaluate the data more objectively, the Pearson correlation coefficient computed a value of 0.31 for this data which suggests a low degree of correlation.

4.2.3 Insertion Sort

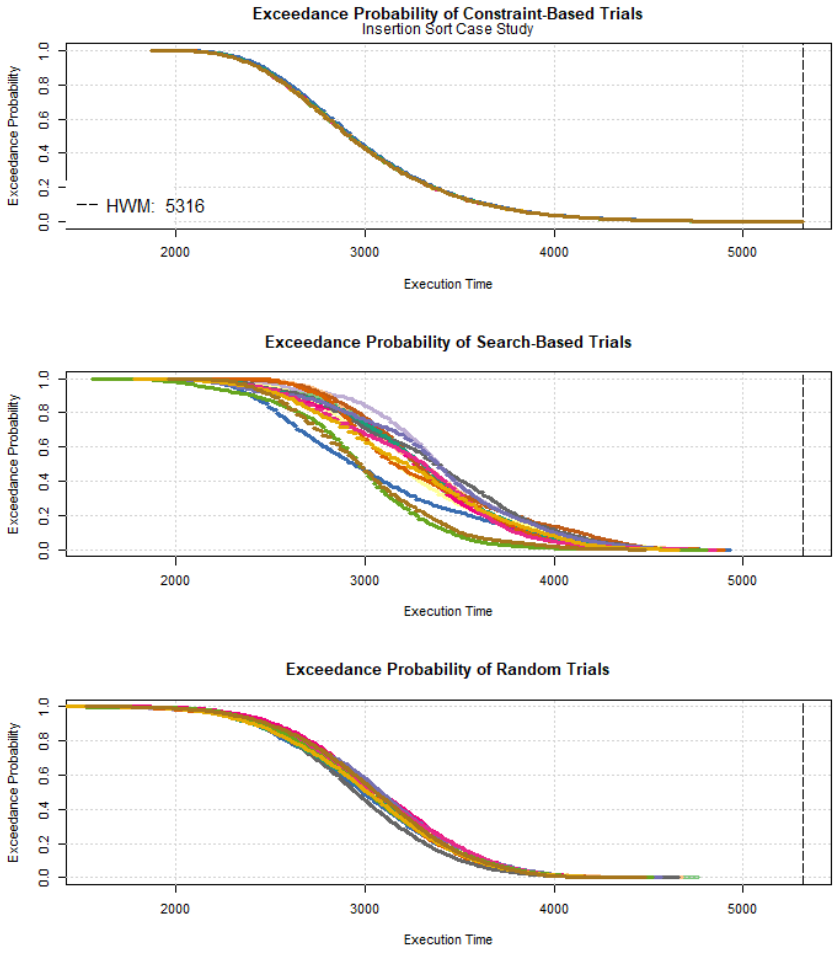


Figure 4.26: Execution time profile each trial corresponding to a different color.

In this case study, insertion sort benchmark is analyzed. As its name suggests, it is another sort algorithm whose complexity is $O(n^2)$ when the test vector is reversely sorted. The size of the test vector was set to 15 elements because of the memory constraints. Figure 4.26 illustrates the execution time profile of all the sessions and TGs. As it can be observed CBTG beats the other two TGs and presumably always hit the HWM.

It is worth saying that with the test vector provided in the original original Mälardalen benchmark only an execution of 4904 was attained (−8.4% underestimation). Hence, the same conclusion to Select K Largest is arrived in this case study: TGs finds test vectors that trigger the global HWM rather than the test vector to trigger the WCET path.

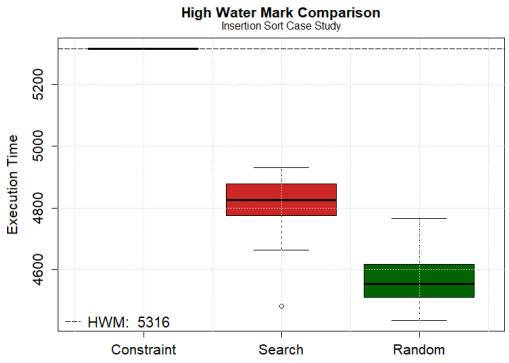


Figure 4.27: HWM results boxplot.

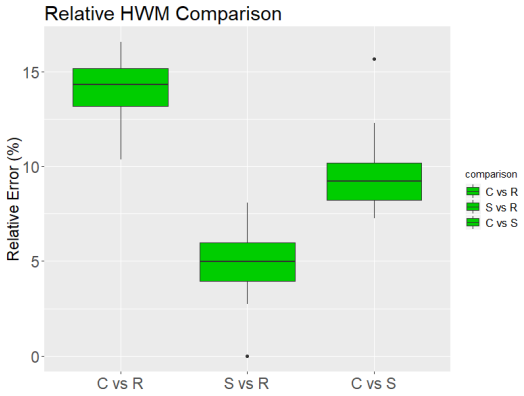


Figure 4.28: Comparison of the *difference* in HWM distribution of the HWM plot. Green plot indicates statistical significance from the WNMT test and red color the opposite.

The HWM results in Figure 4.27 unveil the difference between CBTG and the other two more clearly. According to this boxplot, CBTG is unsurpassed in terms of HWM by the other two. Additionally, the graph suggests that SBTG and RTG achieve some local HWMs that are in the same range. As for the statistical significance and relative error in Figure 4.28 all TGs' HWM are claimed to be different. The HWMs triggered by the CBTG are on average around 14% greater than the ones from RTG, and around 8% greater than the ones from SBTG. SBTG method identifies HWMs around 5% greater than its RTG counterpart.

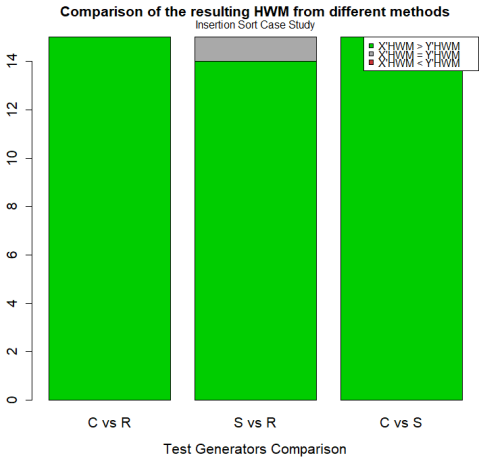


Figure 4.29: HWM comparison.

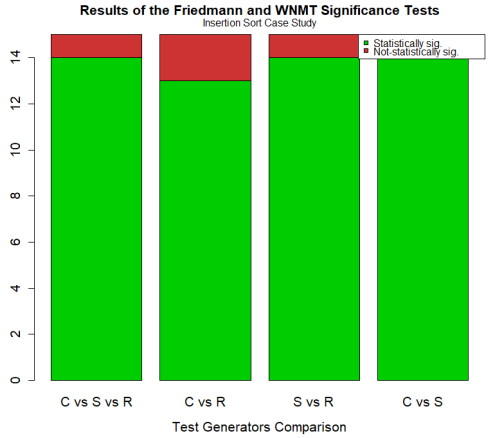


Figure 4.30: Results of the statistical significance delivered by Friedmann and WNMT test.

By comparing the HWM more accurately we get that in Figure 4.29 CBTG always beat SBTG and RTG which has already become apparent in Figure 4.27. SBTG and RTG only draw in one trial and SBTG generally beats RTG. With respect the statistical significance of the methods, Friedmann test contends that the three methods are different in 14 out 15 cases in Figure 4.30. Relatively, the same conclusions are drawn by WNMT test comparing pairwise the TGs. CBTG and RTG methods are claimed to be the same in 2 out 15 cases and only 1 in the case SBTG vs RTG. The rest of them WNMT test reports the TGs methods are significantly different.

Results of the efficiency are displayed in Figure 4.31 with its attached significance 4.32. Even though the efficiency results of the boxplots show the general pattern of escalation of the efficiency it is observed that the difference between CBTG and the other counterparts is reduced since CBTG shows a relative good performance. This motivated by the fact this SUT exhibits a greater number of feasible paths. Despite the fact that efficiency results are close to each other there are still statistical difference in each pairwise combination as depicted in Figure 4.32. In this example, the efficiency of the CBTG is on average around -17% smaller than RTG and approximately -7.5% on average smaller than SBTG. The

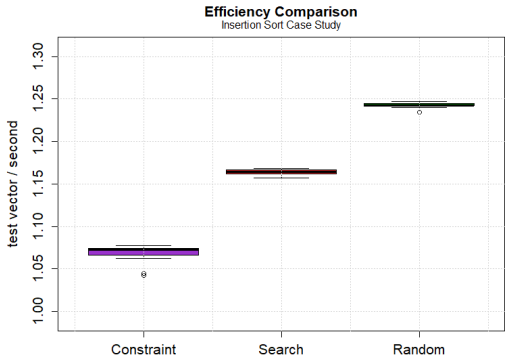


Figure 4.31: Efficiency results.

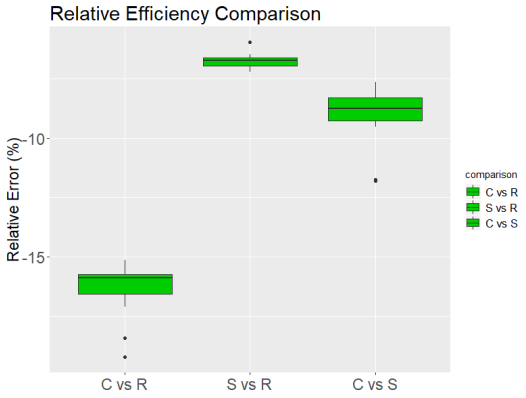


Figure 4.32: Comparison of the *difference* in efficiency distribution of previous plot.

difference of the efficiency between SBTG and RTG is just around 4% on average.

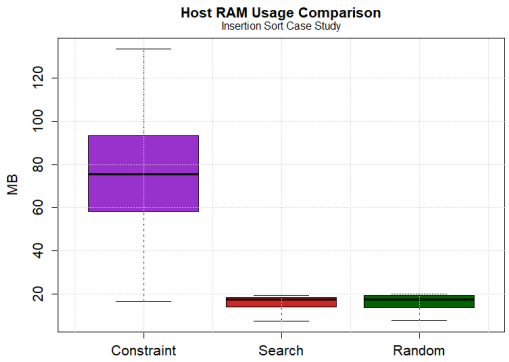


Figure 4.33: Memory usage.

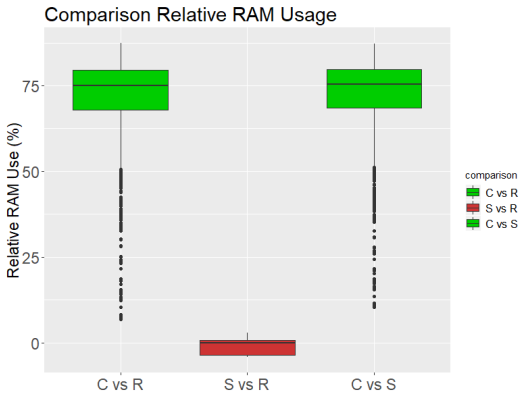


Figure 4.34: Comparison of the *difference* in RAM use distribution of previous plot.

In terms of RAM use as portrayed in Figures 4.33 and 4.34, CBTG again consumes the greatest amount of RAM memory. The interesting feature of this case study is that CBTG does not use as much dynamic memory as the other two cases. We can not actually figure out why this happens but according to the testing delivered the test vector size of this benchmark is as big as it can cope with.

On the other hand, SBTG and RTG show a similar memory use and close to 0% on average of the relative error. So much so, that this combination is the only not-significant combination as shown in Figure 4.34. The relative error indicates that the RAM usage of CBTG is on average around 75% larger for the than RTG and SBTG.

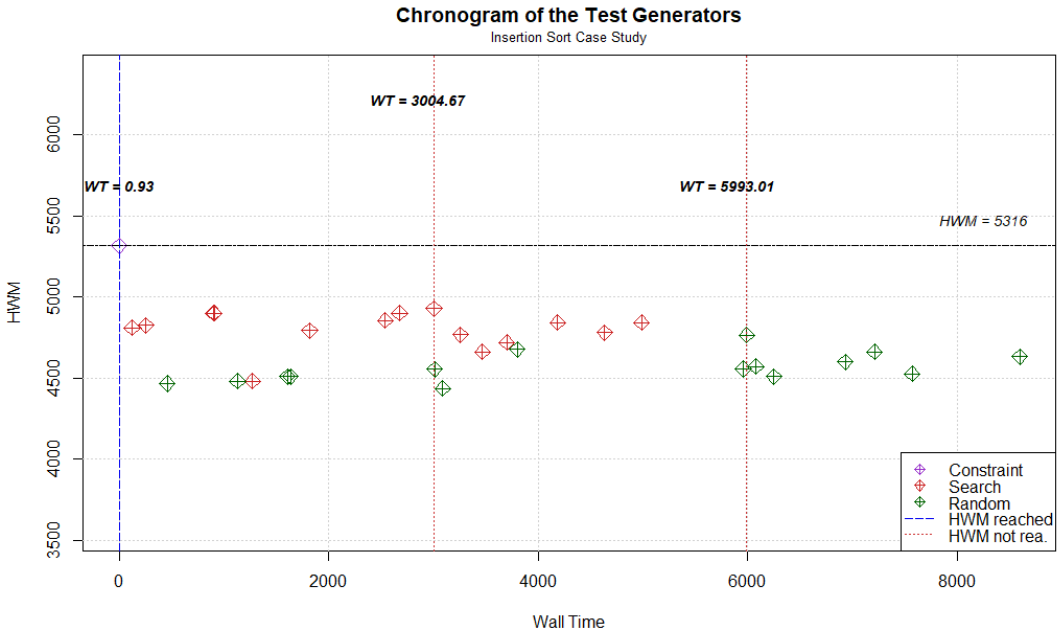


Figure 4.35: Chronogram.

The chronogram displayed in Figure 4.35 outlines when the HWM was collected. CBTG hits the HWM in less than a second. After that, SBTG emerges as the best one in terms of effectiveness and promptness though it never actually meet its objectives. Approximately, half of the RTG seems to unveil its local HWM later than SBTG. By looking at the wall time on the boxplots in Figure 4.36 we can again observe that the CBTG is the only which hits the HWM and does so the fastest. SBTG only hits their local HWM on average in 43 minutes in the SBTG and around 1 hour and 40 minutes in the case of RTG.

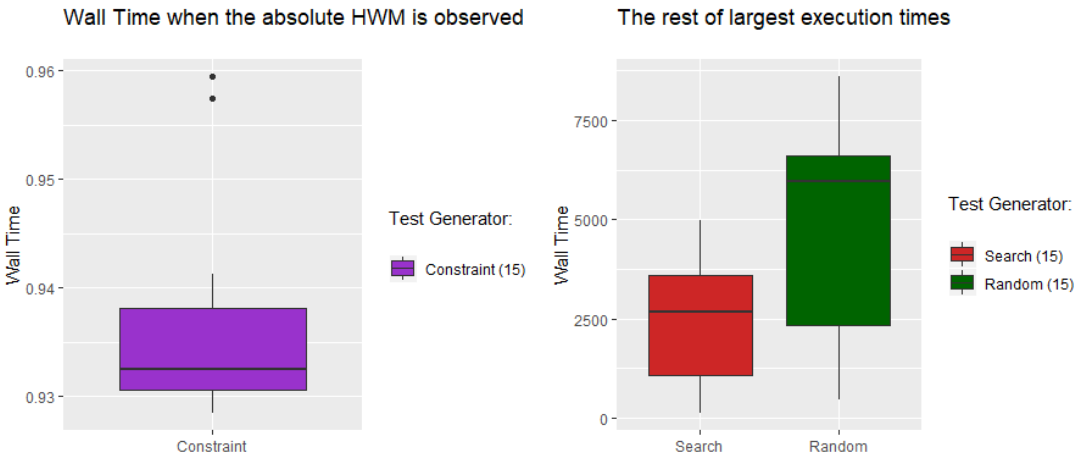


Figure 4.36: Wall time boxplot.

Figure 4.37: Cost and execution times. Each colour and shape denote a different trial.

Perhaps the main reason why this benchmark was so successful for our CBTG is its intrinsic testability in the sense of containing a great deal of feasible paths.

Figure 4.37 displays the correlation between cost and execution time. Similarly to Quick Sort, the heuristic manages to calculate the greatest cost for the HWM. In addition, the cost function is also able to calculate a wider array of different costs. The pattern displayed shows a strong correlation with a relative constant

overlap around cost 80. There are two main contributing factors to explain the results of this graph a) the above mentioned large number of feasible paths which led to create a greater number of different costs and b) the accuracy of the cost function as a result of calculating the innermost loop iterations, which was easily calculated by reading the indexes.

The Pearson correlation coefficient estimated a value of 0.81, the highest value of all case studies, and indicates a strong correlation.

4.2.4 Linear Search

This case study is concerned with search algorithms, in particular, linear search. This algorithm explores an array which may be unsorted or can contain more than one repetition of the element searched. Its algorithmic complexity is $O(n)$. It analyzes 20 integers and one of them is compelled to be the one found, which is controlled by an additional index variable.

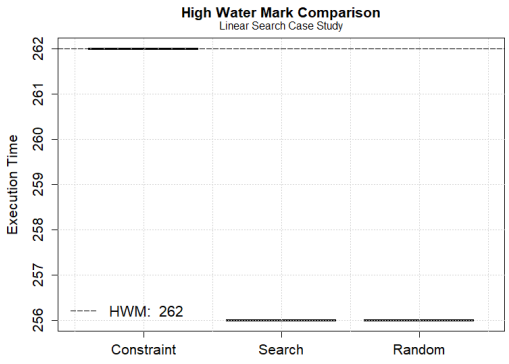


Figure 4.38: HWM Boxplot Comparison

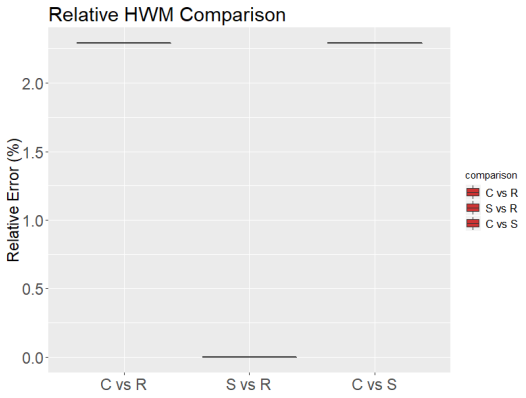


Figure 4.39: Comparison of the *difference* in HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Results of the HWM are depicted in Figures 4.38 and 4.39. CBTG spotted the greatest HWM. SBTG and RTG provided equivalent results. Nonetheless, from the statistical significance point of view - as illustrated in Figure 4.39 - none of the techniques were concluded to unveil a significantly different HWMs.

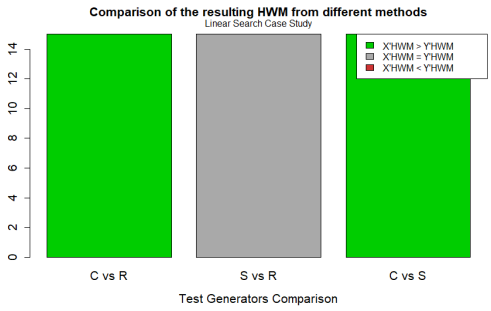


Figure 4.40: HWM Comparison

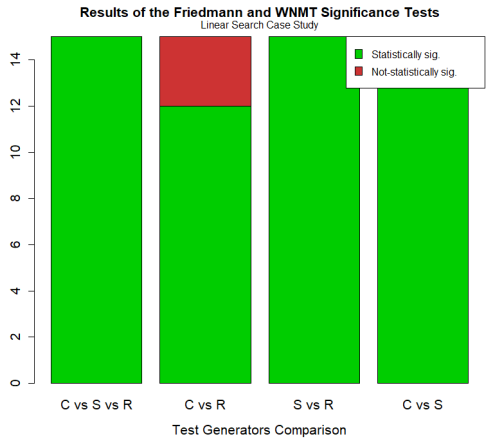


Figure 4.41: Friedmann and WNMT Test results.

The comparison of the HWMs is displayed in 4.40. The new data showed in here is the fact that SBTG unveiled the same HWM as RTG in all cases. Friedmann and WNMT tests on Figure 4.41 concludes that all the methods are mainly different with the only exception in 3 out 15 cases in the CBTG and RTG combination where it was claimed to be similar methods.

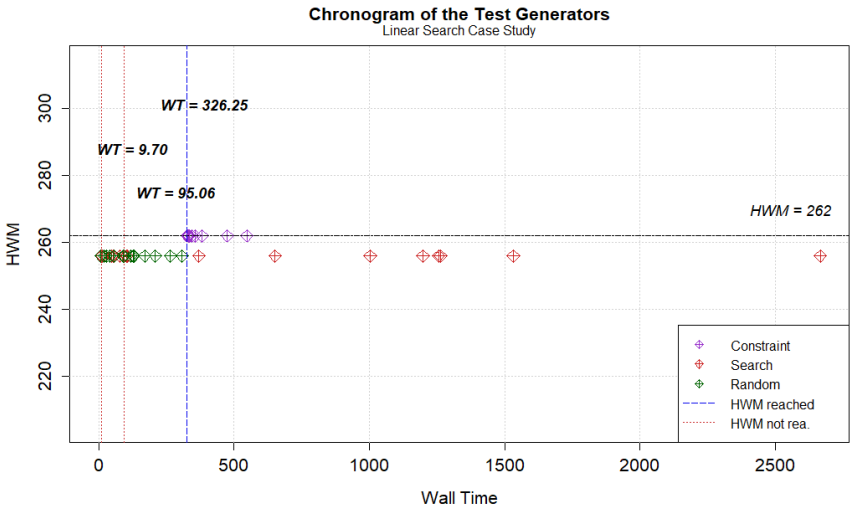


Figure 4.42: Chronogram. Wall time expressed in seconds. Stripped vertical lines denote the first observations of the largest execution times of each test generator.

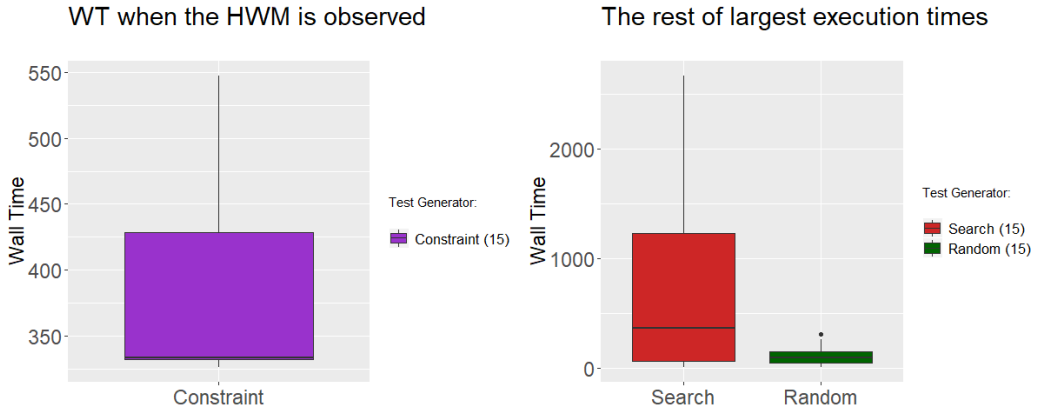


Figure 4.43: WNMT test applied to the wall time distributions

Figures 4.42 and 4.43 are concerned with displaying the wall time data. The CBTG takes the longer the unveil the global HWM in around 5 minutes and 25 seconds the soonest. By contrast the other two TGs did not hit the global HWM. RTG achieves its HWM on average around 2 minutes whereas SBTG takes around 7 minutes.

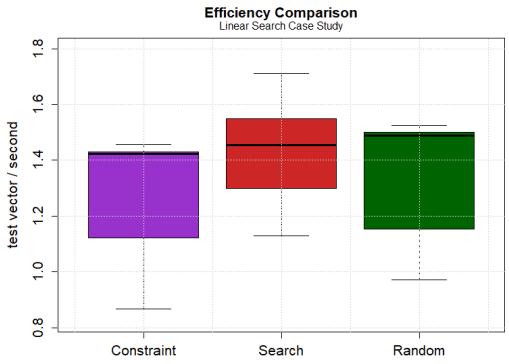


Figure 4.44: Efficiency results.

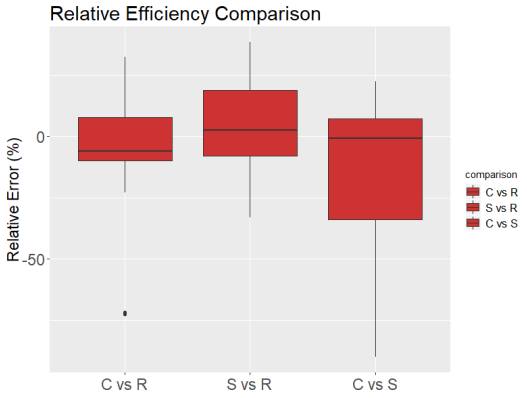


Figure 4.45: Comparison of the *difference* in efficiency distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

When it comes to the efficiency results these are portrayed in Figures 4.44 and 4.45. Interestingly, all of them exhibited a similar efficiency as the statistical tests on Figure 4.45 conclude not statistically significant in all cases.

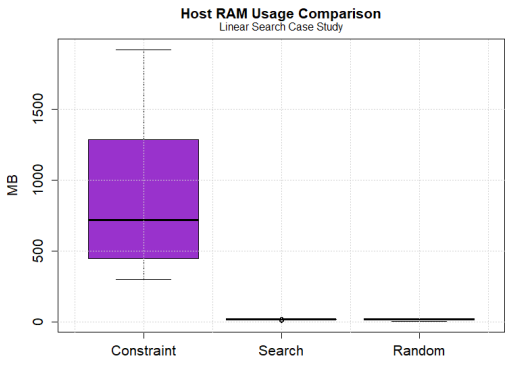


Figure 4.46: RAM usage.

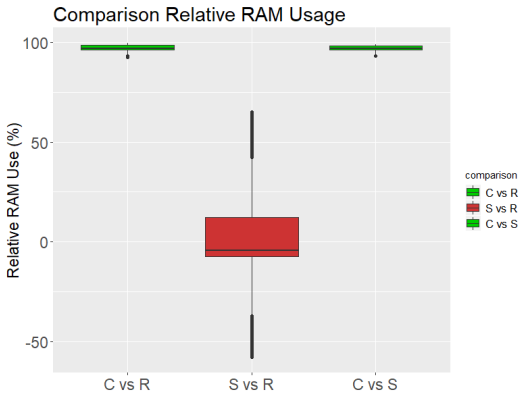


Figure 4.47: Comparison of the *difference* in RAM usage distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

RAM memory results portrayed in Figures 4.46 and 4.47 shows the expected an usual results. CBTG consumes more memory than the other other two TGs. Statistical tests on Figure 4.47 conclude that this data is statistically significant in CBTG vs SBTG and CBTG vs RTG. Their difference was around 2 times greater the consumption for CBTG. Nevertheless, SBTG and RTG display a similar not statistically significant RAM memory usage.

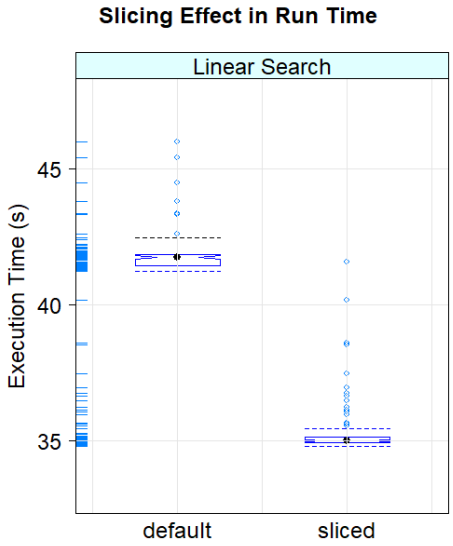


Figure 4.48: Slicing effect in the run-time of the CBTG

The effects of the slicing in the run-time are shown in 4.48 where an slicing of around 11% generated around 16.7% run-time reduction in the path tree traversal. Not such a big impact like other case studies. Friedmann test concluded that this difference in the slicing was statistically significant.

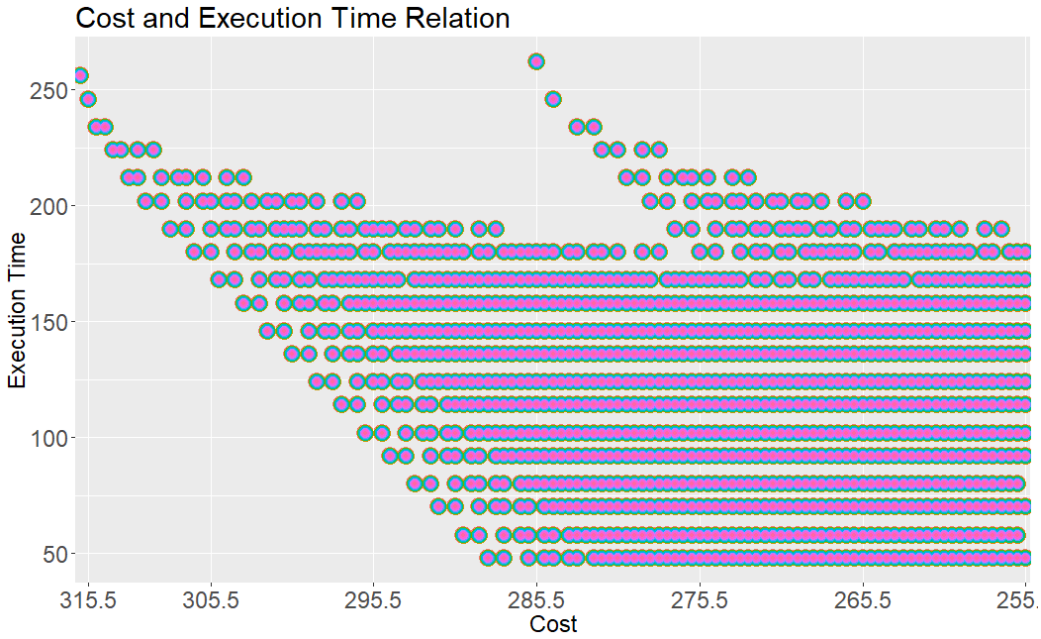


Figure 4.49: Cost and execution time. Each colour and shape denote a different trial.

Lastly, Figure 4.49 displays the estimated cost and the resulting execution time. Unlike other graphs we see two distinguishable decreasing patterns. This is because the random input data supplied which generated different path trees and costs. In addition, the HWM is only achieved in the second one where the cost is not the greatest. The second striking feature is that at higher costs correlation becomes stronger. This is because the characteristics of the benchmark which targets hitting the same decision in the loop and thus the greatest cost is added several times. However, because of the way the addition of cost is defined, it produces a lot of similar cost computations when the different combinations of the rest of the branches are considered. That is why a greater dispersion is found for smaller costs.

The Pearson correlation coefficient reported a value of 0.41 which shows a weak correlation. Presumably, the two differentiated cluster of data in the Figure 4.49 negatively impacted the correlation assessment.

4.2.5 Binary Search

The other search presented as case study is known as binary search. It is another popular search algorithm which assumes that the array is sorted to skip some parts. As a consequence the efficiency is $O(log(n))$. It analyzes a test vector with 20 elements plus the index indicating which the element to be found is.

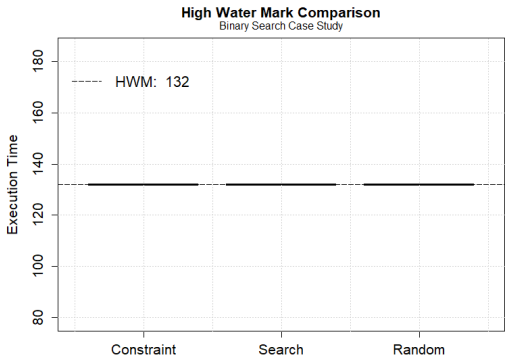


Figure 4.50: HWM Boxplot Comparison

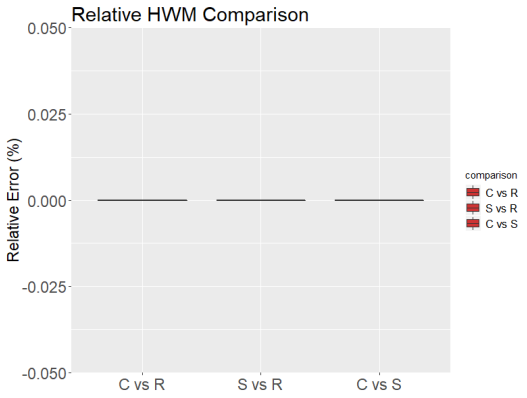


Figure 4.51: Comparison of the *difference* in HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

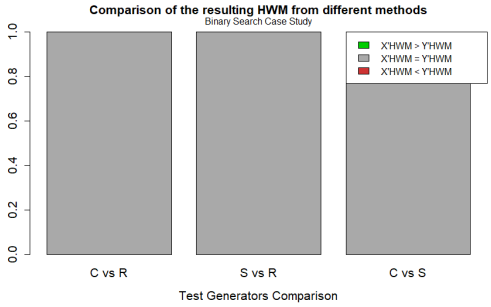


Figure 4.52: HWM Comparison

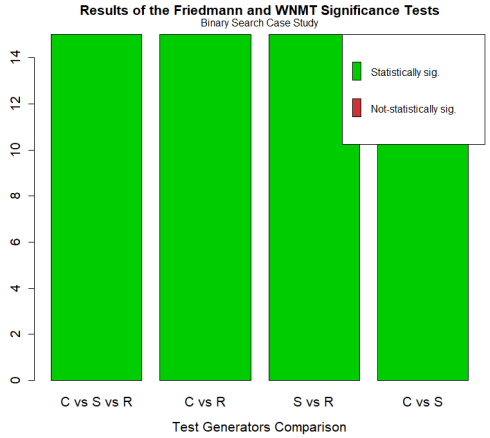


Figure 4.53: Friedmann and WNMT Test results.

Results of the HWM displayed in Figures 4.50, 4.51 and 4.52 are self-explanatory, in all cases all the HWMs are the same and there is no statistical significance as a consequence. However, in terms of the execution time, as depicted in Figure 4.53, all methods are claimed to be significantly w.r.t the resulting execution time.

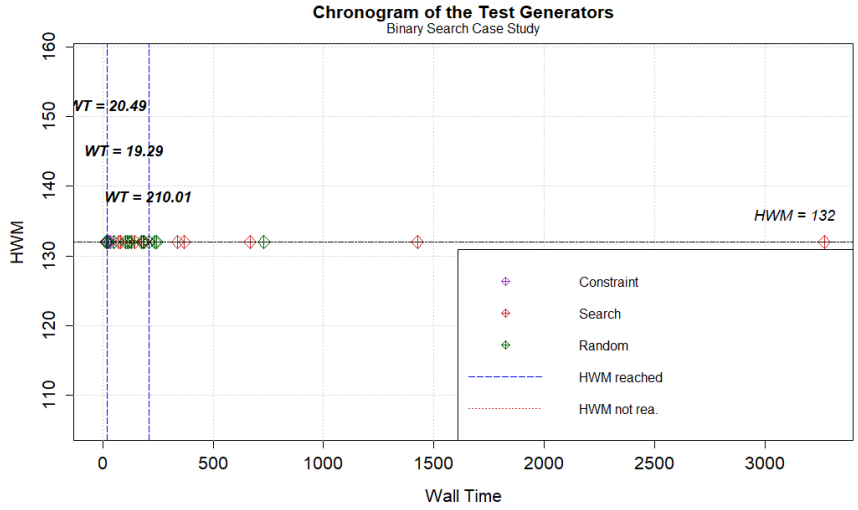


Figure 4.54: Chronogram. Wall time expressed in seconds. Stripped vertical lines denote the first observations of the largest execution times of each test generator.

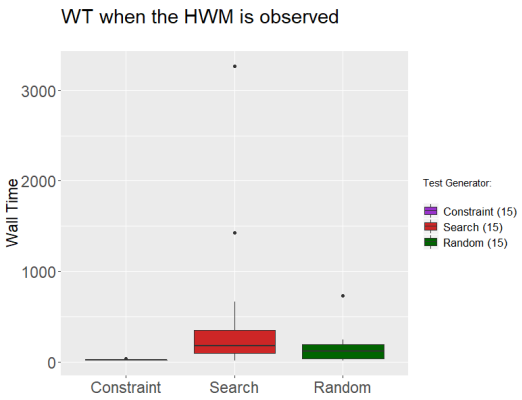


Figure 4.55: Wall time distribution.

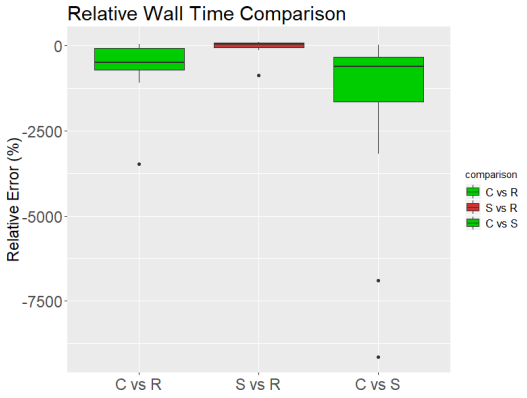


Figure 4.56: WNMT Test results to wall time.

The wall time is depicted in Figures 4.54, 4.55 and 4.56. Unfortunately, the chronogram in Figure 4.54 exhibits some overlapping of the results which are hard to read. On the other hand, Figure 4.55 shows the same data more clearly. By comparing this graph with the chronogram we could infer that CBTG takes around 20 seconds to spot the HWM. However, the average for SBTG is around 3 minutes whereas RTG would be slightly less than SBTG. Statistical significance of the wall time is shown in Figure 4.56 where the wall time of CBTG is significantly different than the other two counterparts. Yet, the wall time was not concluded to be significantly different between RTG and SBTG.

As in previous case the efficiency results - which are depicted in Figures 4.57 and 4.58 - are concluded to be similar according to the statistical tests results depicted on the right-hand side figure. The interesting feature here is that it is the first case where the average of the CBTG is higher than the SBTG. By analyzing the features of the experiment we concluded this was due to the low depth of the path tree traversed along with the great number of feasible paths which did not generate many constraints.

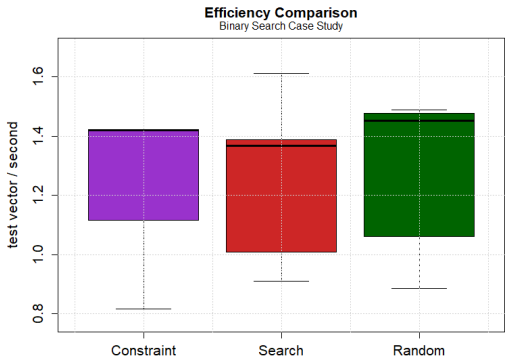


Figure 4.57: Efficiency results.

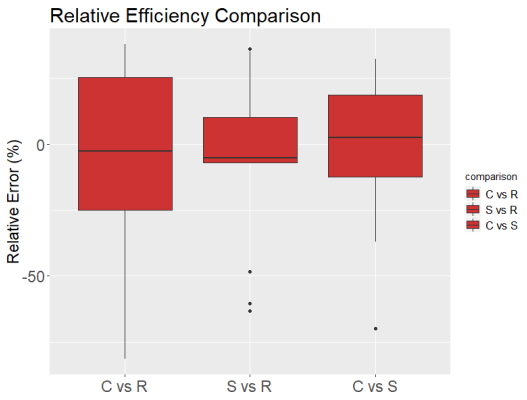


Figure 4.58: Comparison of the *difference* in efficiency distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

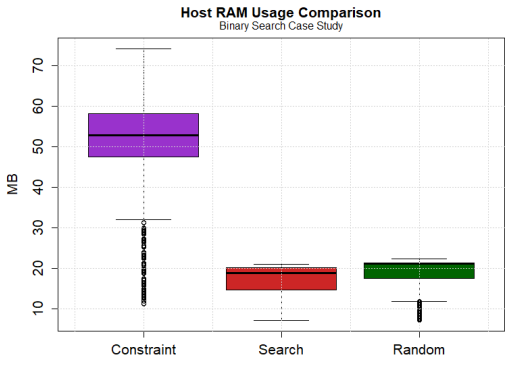


Figure 4.59: RAM usage.

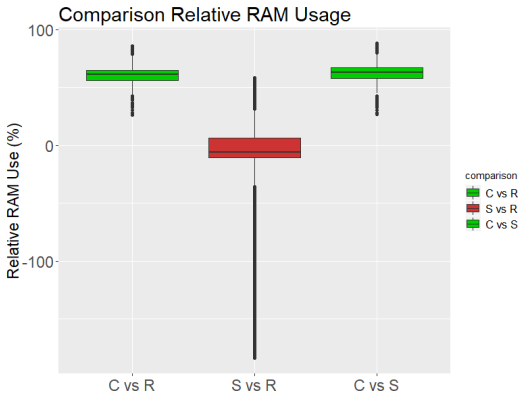


Figure 4.60: Comparison of the *difference* in RAM usage distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

RAM results depicted in Figures 4.59 and 4.60 shows the expected results. CBTG has more memory usage than the other two TGs. This is corroborated by the statistical tests in Figure 4.60. By contrast, RAM usage was not claimed to be statistically significant in the SBTG vs RTG case.

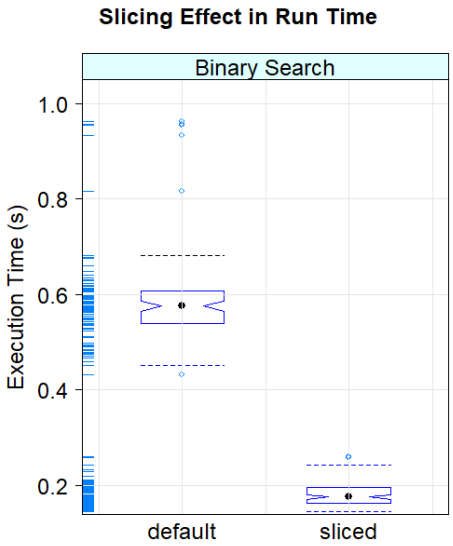


Figure 4.61: Slicing effect in the run-time of the CBTG

Figure 4.61 shows the effects of the slicing in the run-time of the CBTG. The slicing removed around 50% of the unnecessary statements which generated a reduction of approximately 68% improvement of the run-time when applying the BFS to generate the path constraints. Friedmann test concluded that this difference was statistically significant.

Lastly, Figure 4.62 shows the estimated cost and the actual execution times. The cloud of points indicates once more that the cost is not accurate as the same execution time was attached to different costs.

Pearson correlation coefficient provided a 0.68 value which demonstrates a good correlation, despite the apparent dispersion in the cluster of data. Once more, this case study shows similarities with the RC Car in the pattern of the Figure 4.62 as well as in the former coefficient (which was 0.6 in the RC Car case study).

Figure 4.62: Cost and Execution Time. Each colour and shape denote a different trial.

4.2.6 Hash Function

The last case study is inspired by the pathological example for Static Symbolic Execution announced by Godefroid [30, 95]. Hash functions e.g., `x == hash(y)`, are often employed in command parser and text processors [30]. However, these modular equations are beyond the functionality of constraint solvers such as SCIP [90]. Yet, this does not necessarily mean that these functions cannot be tested. The same author proposes a technique [30] by which this kind of functions are treated as a black box. Starting with a random input for the above predicate, the returned value of `hash(y)` is collected by means of instrumentation. In the next iteration, the same value for `y` is preserved and `x` is matched to that collected returned value using a constraint solver so as to trigger the branch. The same technique is applied in our framework.

Our benchmark implemented two hash functions with 4 input integer variables which were distributed between the two hash functions with the `if,else-if,else` structure. The hash function comparison took place in the first two decisions.

Since we do not have a coverage tool, each of these branches included a `sleep` function with different parameter to generate different execution times.

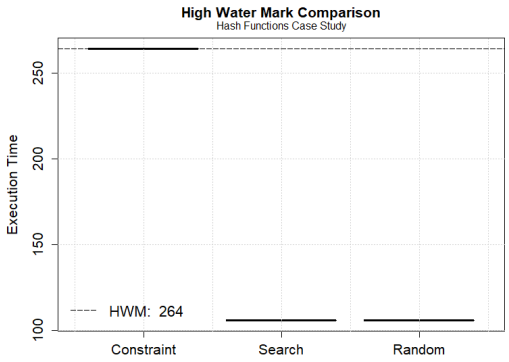


Figure 4.63: HWM Boxplot Comparison

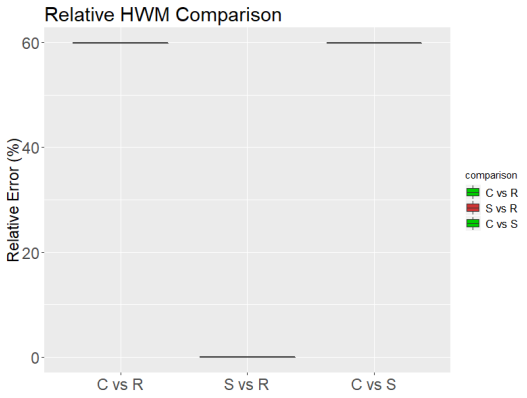


Figure 4.64: Comparison of the *difference* in HWM distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Figures 4.63 and 4.64 display the resulting HWM of the experiments with only CBTG achieving the global HWM and thus the maximum degree of coverage. Statistical significance tests on Figure 4.64 concludes that the HWM unveiled by the CBTG is significantly higher than the other two counterparts. The HWMs from SBTG and RTG did not show a significant difference.

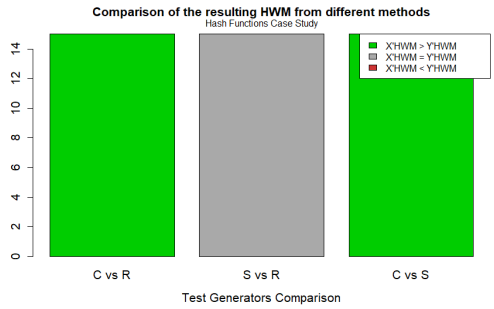


Figure 4.65: HWM Comparison

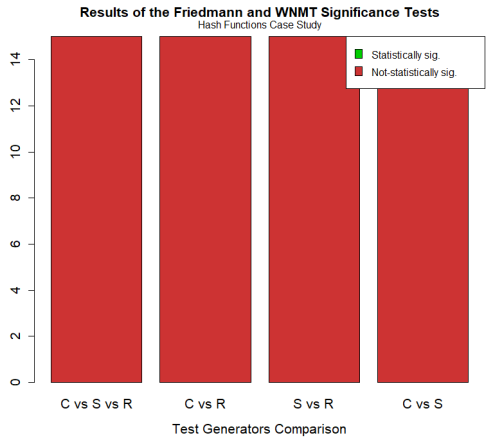


Figure 4.66: Friedmann and WNMT Test results.

Results of the HWM comparison are displayed in Figures 4.65 which shows how SBTG and RTG always found the same HWM yet its HWM always upper bound them. Friedmann and WNMT concluded that all the methods did not generate sufficient difference in the execution time data to argue that they are different methods.

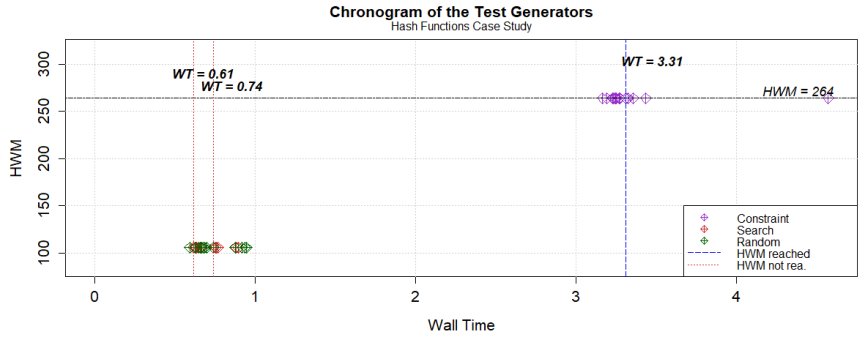


Figure 4.67: Chronogram. Wall time expressed in seconds. Stripped vertical lines denote the first observations of the largest execution times of each test generator.

Figures 4.67 and 4.68 depict the data concerning the wall time. CBTG achieves the HWM in around 3.25 seconds on average whereas SBTG and RTG achieves their local HWM in less than a second.

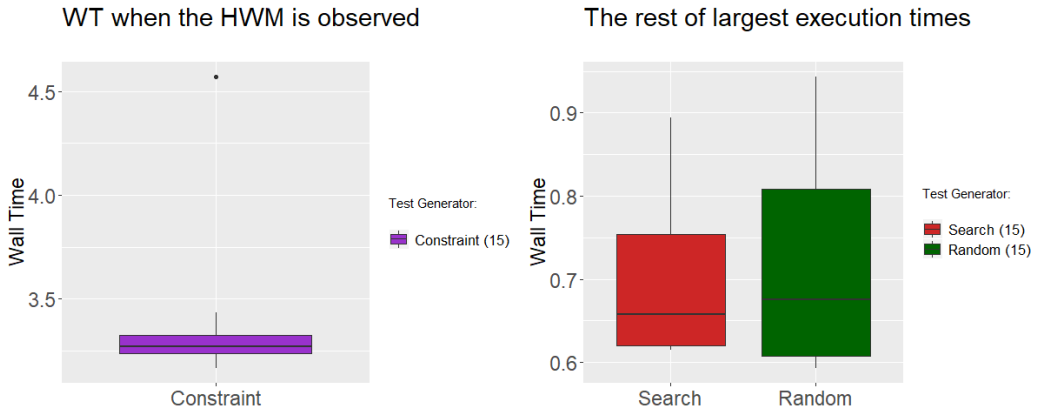


Figure 4.68: WNMT test applied to the wall time distributions

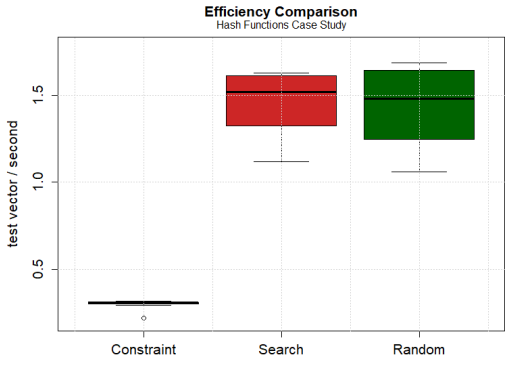


Figure 4.69: Efficiency results.

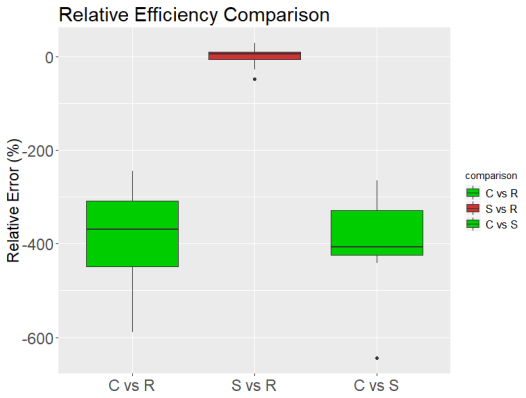


Figure 4.70: Comparison of the *difference* in efficiency distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

Regarding the efficiency - which is shown in Figures 4.69 and 4.70 - data shows the expected results. CBTG has a significant lower efficiency than SBTG and RTG as pointed out by the statistical tests in Figure 4.70. By contrast, SBTG and RTG did not show a significant difference in the efficiency.

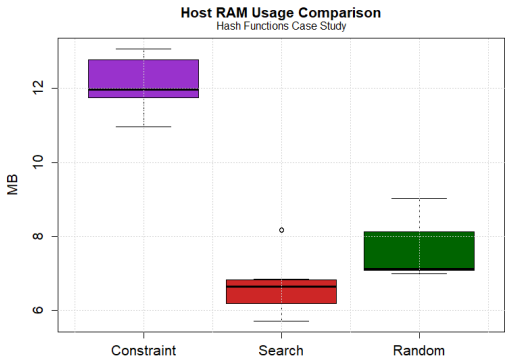


Figure 4.71: RAM usage.

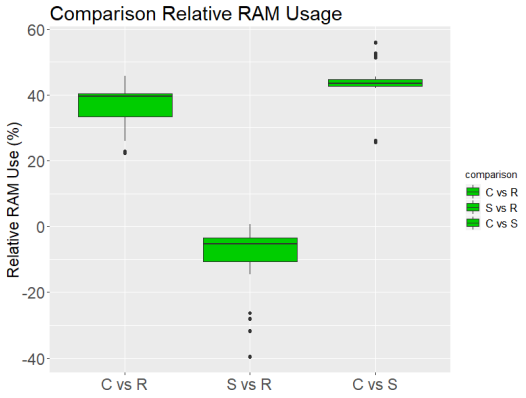


Figure 4.72: Comparison of the *difference* in RAM usage distribution of previous plot. Green plot indicates statistical significance from the WNMT test and red color the contrary.

When it comes to RAM usage, results are illustrated in Figures 4.71 and 4.72. Unlike other case studies, the RAM consumption was claimed to be significantly different in *all* cases.

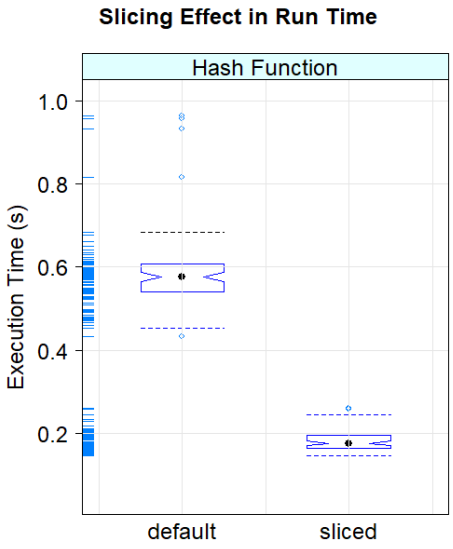


Figure 4.73: Slicing effect in the run-time of the CBTG

Figure 4.73 shows the effect of the slicing on the run-time of the CBTG. The slicing removed 25% of the statements i.e., sleep functions, which had an impact of around 67% reduction in the run-time. Friedmann test concluded that this difference was significant.

Figure 4.74: Cost and Execution Time. Each colour and shape denote a different trial.

Finally, the calculated cost and execution time relation is depicted in Figure 4.74. Results show how the cost is not generally accurate and how the cost function estimated the sleep statement with the same estimation regardless of the sleeping time argument.

The result from Pearson correlation coefficient was 0.35 which entails a poor correlation. It is worth noting that in this case only 3 significant points were achieved for the cost and execution time relation. This fact negatively impacts the confidence of the coefficient result.

4.2.7 Threats to Validity

This section discusses the threats to validity. Firstly, the experiments presented share most of the threats discussed in Subsection 3.4.5. In addition, we would argue the following threats.

- **Threats to internal validity** includes the kind of benchmarks employed to evaluate the dynamic constraint collection as they have analyzed popular algorithms in computer science. There exists other dynamic constraints that may become apparent when using Operating Systems and using function calls to which the source code is not available [30]. These cases have not been contemplated because the restrictive access to proprietary software and the technical limitations of our tools.

  Another threat comes from the data in the Hash function case study in which the resulting HWM is influenced by the manual addition of the sleep function rather than an pre-existing code. This may cause a bias in the HWM data to which statistical tests may only lead to questionable conclusions. In addition, research question 2 concerned with assessing the similitude of the methods based on the execution time is also negatively impacted because of the low number of execution times as a result of the benchmark. Last but not least, it also has the similarity of the equality predicate as equivalent to Needle-in-haystack case study so maybe some existing techniques in SBT could achieve better results [58].

  Another threat comes from the fact that the number of iterations could be calculated easily for insertion sort which could explain why the cost and execution time was relatively easy. This could be a potential bias in the experiment as this feature is not necessarily available for all loop constructs.

  Lastly, the way the path tree can only be partially constructed when adding dynamic constraints in these experiments, lead to the fact of not observing all the resulting costs that would have been observed if the full path tree

were constructed.

Even though, Select k Largest has shown bad results for our approach, its internal flow structure violates the guidelines to write real-time analyzable software [10]. Thus we have some doubts that such a software is a representative piece of software for industrial real-time systems.

- **Threats to external validity** to other embedded targets or benchmarks may come from the fact that the cost function is mostly inaccurate and for large-scale programs only a subset of paths can be built based upon this potentially misleading guidance. As a result, the CBTG can target paths which may not actually hit a high-confidence HWM and leave out other paths which could achieve this objective.

  Because of the potentially misleading effect of the BFS algorithm as a result of the cost inaccuracy, the counting of statements may not be adequate. This effect is well illustrated in Subsection 3.2.4 where even for relatively simple benchmarks the cost estimation failed at giving similar results as results ranged from around 15% in the best case to around 13 times difference in the worst case. Thus, a more accurate path cost definition i.e., that the cost is representative to the actual execution time, is required in order to improve the confidence of the resulting HWM and the wall time of the proposed CBTG methods

  However, most of the experiments in this chapter epitomize a path explosion effect and the resulting HWM from the CBTG has only underestimated the global HWM in 1 out 6 cases.

  Moreover, by instrumenting and collecting the constraints on the host may create a side effect known as *divergences* [30] between the features of the host computer and the embedded platform. For example, divergences in the

precision in fixed point and floating point numbers. In practice it would be more convenient to collect values from the an actual execution of SUT rather than similar one on the host computer.

4.3 Coverage Observability and Infeasible Paths Detection

An instrumental argument to argue about the confidence of any testing process for MBTA is the degree of code coverage achieved [18]. To give confidence that these objectives are met, we need way to know how the coverage is traversed. However, software is not observable *per se*. The observability of testing is achieved in the embedded industry thanks either by instrumenting the SUT [49, 35] or by using complex hardware debugging units [32].

In reference to TGs for MBTA, a downside of the SBTG and RTG is that so far, it is not possible to quantify the degree of underestimation [21] unless the above mentioned observability mechanisms are available. Nevertheless, when embracing CBT a record of the traversed branches or paths can be stored as a result of the tight coupling between the SUT and the test generator. This statement holds water under two assumptions: I) the parsed source code is representative of the resulting object code and II) the test vector is actually run on-target. Regarding first assumption it can be deemed reasonable as CRTES often use qualified compilers [19] so as to guarantee the compiler's output does not jeopardize safety.

As a result, by holding three sets namely, all constraints ACC which is initialized when the constraints are added with two additional sets initially empty: non-covered constraints, NCC and the covered constraints CC. Next, CC adds the constraint traversed every time a path to test is feasible and thus a test vector is generated. At the end of the test session CC will be initialized and the degree of coverage can be computed with the cardinal of ACC and CC ($\frac{\#CC}{\#ACC}$) whereas the non-covered branches can be computed by the disjoint union $NCC = CC \sqcup ACC$. It is worth mentioning that given that our CBTG is concerned with the notion

of paths, in each feasible analysis of the path the constraints covered indicates which path in the code has been traversed. This function is not possible with state-of-the art CBT approaches for MBTA [40] as they are not equipped with the notion of path.

4.3.1 Reduction of Pessimism in CWCET derived in Hybrid Approaches

Aside the observability record when traversing the constraints, the feasibility analysis of the paths can be really useful to reduce the pessimism of hybrid approaches. This objective is tackled in the realm of STA where state-of-the-art approaches aims for analyzing at instruction level and thus it analyzes a lot of redundant information rendering an inefficient analysis [34]. Further, not all can be analyzed with this approach because some states could not be concatenated and time calculation explosion. However, to carry out this optimization a complex analysis of the data flow must be done which is not the case when using path composition techniques of Hybrid approaches [50]. Yet, this kind of analysis is delivered in the CBTG we have undertaken as we need to decide the input variables.

In order to compute a CWCET, hybrid approaches focus on gathering execution time stamps in strategic points along the SUT by means of instrumentation or some other tracing mechanisms. When these time stamps are processed, the structure of the SUT they are inserted in is considered so as to derive sound estimations of the CWCET. The so-called timing schema derive a CWCET by representing the structure of the SUT in a tree where transition of instrumentations points correspond to the leaves of the tree [50].

The parent nodes in the tree contain some labels which indicate which pattern to apply to compute a sound CWCET. These patterns are: *sequence* (block of code), *selection* (decisions) and *loop*. Such patterns in turn use *abstract operators* to compute a CWCET in a tree fashion where the root holds the CWCET at the end of the process. These abstract operators have different definitions depending upon the data gathered at the instrumentation points. Particularly, $\otimes$ stands

for standard addition when the data is a single execution time and convolution when an execution time profile is gathered. Later on, same authors included the notion of *copulas* [77] to replace convolution since it assumes the distributions are independent. Moreover, $\odot$ denotes multiplication and power operator in the execution-time profile case. The abstract operators are used along with code patterns to conform a CWCET. They can be defined as:

- Sequence $\{i, j, k\}$: $C_{i,k} = C_{i,j} \otimes C_{j,k}$
- Alternative $C_{i,j}$ outside the alternative and $\pi \dots m$ the execution time of alternative paths: $C_{i,j} = max(C^{\pi}_{i,j}, \dots, C^{m}_{j,k})$
- Loop assuming i and k are outside the loop and n the maximum number of iterations: $C_{i,k} = C_{i,j} \otimes (C_{j,j} \odot n) \otimes C_{j,k}$

A striking feature of this approach is that the composition of the CWCET is done with regard the structure of the SUT and not the data flow. As a consequence, a CWCET may be the result of an infeasible path that can potentially increase the pessimism of this bound. Still, because of the data analysis and infeasible path detection delivered by the proposed path-based CBT this information can be supplied and thus reduce the pessimism. Next section offers a case study about this advantage.

4.3.2 Landing Gear State Machine Case Study

To give a more realistic example we present a benchmark advocated to enhance the importance of detecting infeasible paths to reduce the pessimism. This benchmark was translated to Ada from one of the demos of the industrial hybrid timing analysis tool RapiTime [35] includes.

In this benchmark the user must supply annotations manually of infeasible paths to reduce the CWCET by RapiTime. The benchmark in question consists of a sequence of 4 main branches that in turn contain others decisions inside. Each of these branches discern different output depending upon 3 operating mode the

undercarriage is in. Only two of these branches have the same state. Because of the syntax of the structure the CWCET composition assumes that all these branches are feasible. When analyzed this benchmark by GenI, the tool reported that the benchmark contains 81 paths and only 8 are feasible (9.87%). A test vector was created for each feasible path and reported 2 times branch and path coverage. When it comes to timing analysis - which is the bulk of this experiment - the code was manually instrumented and tested on-target using GenI framework.

By using the above mentioned timing schema on a selection structure along with the test vectors from GenI, a theoretical path-based CWCET would be 368 clock cycles whereas the high confidence HWM is only 250 after running all the tests and achieving full path coverage. This entails a reduction of 47.2% pessimism if infeasible paths are not detected.

The obvious threats of validity here is the number of case studies. In addition, this benchmark is fully controllable.

4.4 Summary

This chapter has relaxed two hypotheses of the former chapter and has established that: I) Partial path coverage can normally be achieved and II) dynamic constraints need to be collected. As for II) we have used concrete value execution and running the code statically. The underlying motivation was the good results in the realm of Dynamic Symbolic Execution [95] and just to embrace a simple solution. When examining the selected benchmarks we also noticed that a) some of the test vectors were equipped with fixed point input data and b) for the sort algorithms the test vector was read after being written during the execution. The latter fact compelled us to describe a tracing algorithm based upon sets and hash maps so as to provide sound constraints for the constraint solver.

The evaluation has included 6 different benchmarks. The most remarkable conclusions of each case study is described as follows:

1. **Select K Largest** has shown how CBTG has neither met its objectives on maximizing the HWM nor it has achieved the best wall time. The fact that our approach struggles with infeasible paths leaves some gaps to be addressed as part of future work at constructing paths for CBTG.

2. **Quick Sort** has demonstrated how CBTG has succeeded at identifying the global HWM and doing so the soonest. Thus, its objectives have been met. Perhaps the main obstacle in this evaluation stems from the fact that a relatively small search space was generated due to the path explosion effect.

3. **Insertion Sort** has been positive at meeting the objectives of our CBTG as it could trigger the HWM and achieve the best wall time.

4. **Linear Search** has exhibited how our CBTG was the only one unveiling the global HWM. However, this data was not statistically significant w.r.t any of the other two test generators.

5. **Binary Search** has shown similar results to RC Car case study. There is no difference in the HWM but our CBTG exhibited the smallest wall time. The similarity of the HWMs was due to the fact that the worst-case exit condition of the main loop was relatively easy to achieve by a large number of test vectors.

6. **Hash Function** has demonstrated how our approach is able to handle hash functions by applying some black box techniques which consists of collecting the value of the hash function and match the other input data to this value. Results have shown how only CBTG was able to achieve the global HWM. Again, this was due to the accuracy of the constraint solver.

The experiments of this chapter have demonstrated how those loops which have a single exit decision are more compatible with our approach since maximizing the loop iterations can be more easily controlled by suitable constraints. Whereas those loops exhibiting a more unstructured flow with several exit decisions distributed in the body loop e.g., Select k largest, have been more problematic and

our approach has achieved poorer results. These observations match the prerequisites to apply static timing analysis techniques [7].

The effects of the slicing have been statistically significant in the cases where attaining this data was possible be leaving some memory margin to build a larger non-sliced path tree. A removal of 11%, 50% and 25% produced a 16.7%, 68%, 67% execution time reduction. On the other hand, the cost function has been normally inaccurate.

The evaluation has also assessed the accuracy of the test vector provided in the Mälardalen benchmarks. The results of the case studies has shown that the default test vector only gave sound results in Quick Sort and it was upper bounded by the TGs in Select K Largest and Insertion Sort.

The threats of validity have argued some potential bias in the Hash function case study because the manual addition of sleep functions. In addition, insertion sort, whose loop cost estimation has been simple, could give the impression the cost estimation is accurate. Additionally, it has contended that the inaccuracy of the cost estimation and the fact that partial path coverage is often required may produce a significant underestimation of the HWM. Further, some divergences may occur as a result of constraint collection and test generation in the host computer and the actual embedded hardware platform. For example, due to the precision of floating points numbers.

The other part of the chapter has described how the CBTG process is able to provide more knowledge than SBT and RT on whether constraints were covered by the test vector. Last but not least, thanks to the use of a constraint solver and the data collection from the SUT, this process may help at reducing the pessimism of the CWCET in those path-based approaches that collect execution time data from branches and then a CWCET is composed by looking at the structure of the code. The landing gear state machine case study - whose benchmark is provided by a commercial instrumentation tool [35] - has shown how our approach reduces the pessimism of the CWCET considerably. Its threats to validity have argued

that it is only a single case study. However, this contribution is more secondary.

So far, the cross comparison of the HWMs by the multiple TGs have shown, in some cases, different levels of *uncertainty*. In the following chapter a new form of probabilistic analysis is investigated. This upper-bounding method is advocated fill this gap by using the tail of statistical distributions.

Chapter 5

MBPTA with Distributions in Maximum Domains of Attraction of GEV Distributions

The bulk of this work so far has been to devise a test generation process that maximizes the HWM. However, it is generally hard to tell how close to the actual WCET the HWM is and thus a safety margin is sought. Since the WCET is generally unknown we need to appeal to the notion of confidence. In this respect, probabilistic approaches take the tack that the upper-bounding can be achieved with a probabilistic argument.

From the literature survey, two main methods of probabilistic analysis have been developed:

- **Copulas** are to some extent, similar to *convolution*. They are suitable to compose ETPs from instrumentation points to derive a global CWCET. Still, to the best of our knowledge, they are only useful in path-based approaches to compose CWCETs [77].
- **EVT** is a modern field of statistics advocated to predict extreme unobserved magnitudes. It has become increasingly popular, yet in the recent years a number of issues have become apparent [84, 72]. In our view, the most relevant issues are I) the lack of automation derived from the selection of

maxima which may hinder its exploitation by the industry [23]. And II) the way exceedance probabilities are calculated which damages the confidence of the pWCET.

In the 2000s some research works devise tail tests [88, 24] which consists of GoF tests for the tail of some parametric distributions. With this analysis the tails of parametric distribution are equipped with predictive power of extreme events. To do so, EVT is used underneath to compare the tail results against them. A key advantage of this approach is the full automation. In addition, this approach may be used as a way to compensate for the uncertainty between the HWM and the WCET when applying MBTA by considering the asymptotic properties of the parametric distributions.

Some other works have focused on fitting methods and have embraced meta-heuristics [75, 89] to maximize the confidence of the fit of the distribution. Such methods can be used to meet the requisites of the tail tests.

The objective of this chapter is to make a contribution toward using probabilistic analysis to give a confident upper-bound in those testing scenarios with *unknown uncertainty* of the WCET. To do so we are using the *known uncertainty* i.e, the gap between the local HWM of a TG and the global HWM to determine reasonable exceedance probabilities. To achieve this goal, the current chapter is concerned with describing, integrating and evaluating this novel tail tests for MBPTA as we believe it has some advantages w.r.t EVT. Therefore, the contributions of this chapter are:

(a) Contribution 7 from Section 2.5 on describing the theory of tail tests that enable automation of the analysis and sounder pWCET estimates.

(b) Contribution 8 from Section 2.5 on providing with an evaluation of the resulting data of the case studies and an industrial case. The goal is to investigate whether the new analysis can be applied in order to calculate confident exceedance probabilities as a measure of uncertainty which may be later used to calculate a confident pWCET using such tail tests.

(c) Contribution 9 from Section 2.5 on including SBF in a attempt to increase the number of cases where tail tests can be applied and thus obtain more data of exceedance probabilities.

The rest of the chapter is organized as follows: Section 5.1 outlines a critique the application of EVT to MBPTA with the purpose of understanding the weaknesses and how the proposed approach may alleviate some of them. Section 5.2 introduces the theory behind tail tests so as to comprehend its fundamentals and provide with Contribution (a). Section 5.3 tackles the evaluation of the proposed analysis and therefore describes Contributions (b) and (c). Eventually, Section 5.4 summarizes the content of this chapter.

5.1 Critique of EVT-Based MBPTA

The application of EVT-Based MBPTA underpins three assumptions:

1. The WCET is not observed in the testing process and thus there exists some **uncertainty** from the testing process to be upper-bounded by the tail.

2. **EVT can be applied** which entails certain shape of the empirical distribution plus more verification steps according to the prescription of EVT analysis [84].

3. The derived **pWCET** using exceedance probabilities **upper bounds the WCET**.

As for the first assumption a perfect TG would not yield any uncertainty and thus the main motivation upholding EVT would vanish. Paradoxically, the contribution of TG was not studied when MBPTA was propelled [1, 105]. In most cases, because the complexity of path and state coverage, some form of uncertainty will normally rise in MBTA.

The following subsection tackles the second the assumption and third assumption in the last ones.

5.1.1 Execution Time Profiles and Probability Distributions

The second assumption for the application of EVT hinges on the execution time data and results more elaborate. Common practice in statistics dictate to supply *representative* data for the analysis [23]. The notion of representative in this context would entail to test the systems as if deployed which is often called statistical testing [36]. Actually, there is seldom a single distribution that is representative given that different users or contexts normally generate different ETPs. Let alone, the effect on the physical environment in the circuits of the hardware. If performance testing was delivered using statistical testing, the resulting distribution may not still be EVT friendly i.e., to satisfy GoF tests, due to the intrinsic structure of the software and hardware it runs on. To give an example consider Figure 5.1.

Figure 5.1 epitomizes the difference between the ETP from a industrial controller and the insertion sort tested by the RTG. The first benchmark exhibits a multimodal distribution with two main lumps plus some imperceptible few execution times on the right of the second one. A similar profile was depicted for the MBPTA projects [1]. However, it is hard to predict what selection of maxima will be delivered and thus where the EVT distribution would be located. Most statisticians would agree that the second distribution is easier to analyze with a parametric distribution given its bell shape which resembles to Normal distribution. The root of the problem is the resulting data of the performance testing process.

State-of-the-art works have proposed two main solutions to derive EVT friendly distributions regardless of the testing process.

- **To randomize the execution time** in order to reduce the *discreteness* of the ETP. The main issue with the randomized execution time is the controllability of the process plus the objective of achieving state coverage.
- **To pad the empirical distribution** [68] and thus modifying the frequen-

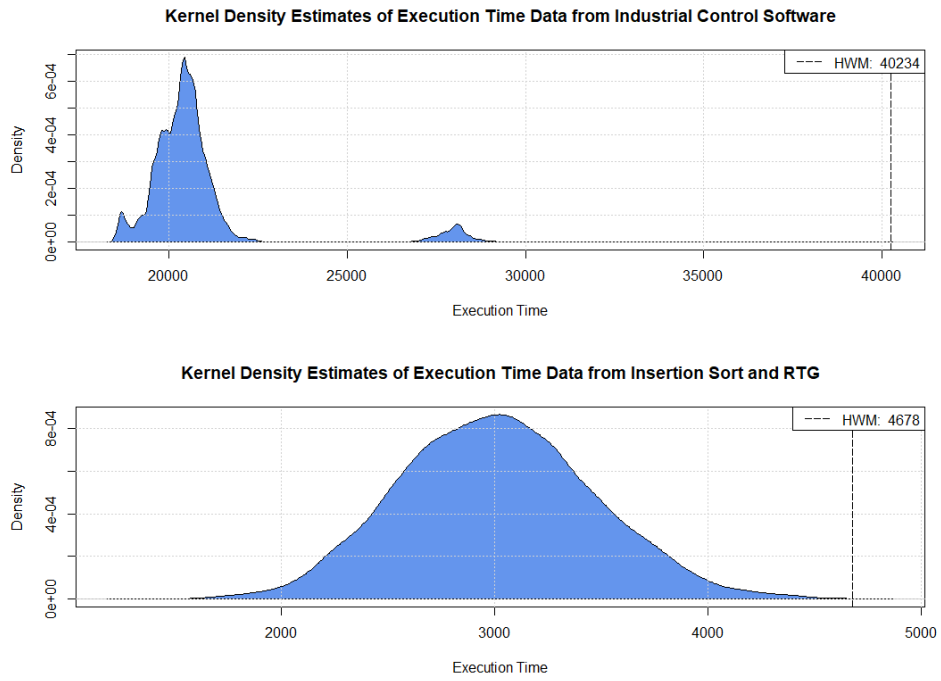


Figure 5.1: Execution time profile of an industrial benchmark and a academic benchmark of a sort routine. Some details of the upper benchmark are anonymized because of non-disclosure agreements.

cies. The downside of this method is that it may alter the asymptotic properties of the resulting EVT distribution.

The first approach aims for changing the technology to which critical systems are reluctant to [19] because of the attached risk and the cost-benefit analysis. Apart from that, Lima et al. [84] work concludes that randomization is not sufficient to guarantee EVT application and time-deterministic architectures may allow its application in certain cases.

Lastly, these argument of the integration of ETPs and probabilistic distribution can be extrapolated to our novel tail tests for MBPTA.

5.1.2 Exceedance Probability

The last assumption of EVT-based MBPTA consists of believing that the pWCET upper bounds the WCET. Admittedly, this hypothesis is hard to verify if we attempt to know the WCET. Especially in complex modern architectures equipped with multicore and multiple level of caches as state coverage is unmanageably complex. Thus, EVT can only be useful as long as it is the only way to justify a safety margin.

Nevertheless, by looking more carefully into EVT analysis we note that regardless of how the distribution is expressed, namely, density, cumulative or exceedance probability it will always tell us a value of probability, y, given a value of the random value variable being modeled, x, or the other way round. Because we intend to calculate a pWCET from the Complementary Cumulative Distribution Function (CCDF) (x) we need inexorably a value of exceedance probability (y). Most works on probability analysis have applied a general way to compute an exceedance probability. For instance, Cucu-Grosjean et al. [26] calculated the *'tail extension'* by taking the Safety-Integrity Level of the task being analyzed as well as its period (T).

$$p \geq \frac{1}{3600} \times \frac{SIL}{T} \times \frac{failure}{activation} \tag{5.1}$$

With Equation 5.1 they were able to compute the some values what were upper bounded by 10^{-15} by taking the most critical Safety-Integrity Level (10^{-9} $\frac{\text{faults}}{\text{hour of operation}}$ [19]) and the greatest period found in embedded avionics systems.

It is worth remembering that in this work only Gumbel distribution was believed to be appropriate for the fitting of the analysis. The parameters employed in the tail extension equation are independent of the performance testing and thus disregard the *uncertainty*. Admittedly, the data used in these case studies resulted from manual testing [26]. This point is important because as we saw in the former chapters, different *uncertainties* arose in different TGs and case studies.

Furthermore, the calculation of exceedance probability in Equation 5.1 encom-

passes the system as a whole and dismisses the effects of the selection of extremal data when the EVT distribution is fit. Next subsection reviews this issue.

5.1.3 Selection of Maxima

Any EVT distribution is only concerned with the distribution of the maxima observed. Therefore, it ignores a great deal of data of the sample. This decision also impacts the exceedance probability of the resulting EVT distribution as identified by Lima [87]. In a seminar, Lima deduces an equation based on the work of Coles [70] to transform the exceedance probabilities of the complete sample (p) to the exceedance of the extremal data (p') using the number of BM or PoT (b).

$$p' = 1 - (1 - p)^{\frac{1}{b}} \tag{5.2}$$

To contemplate the consequences of this equation in a visual fashion, consider Figure 5.2.

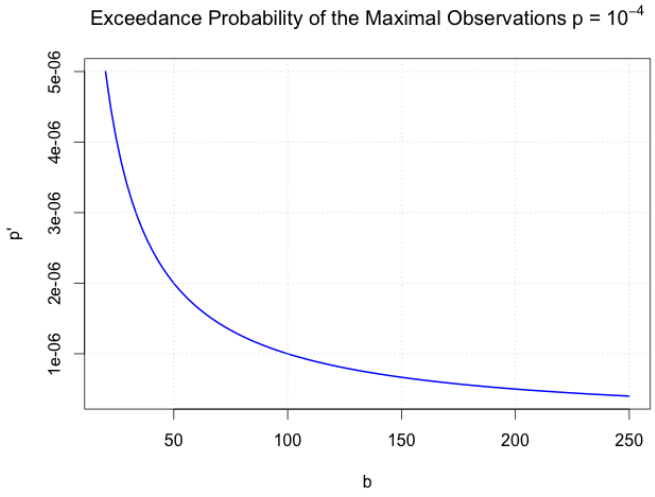


Figure 5.2: Exceedance probability for the extremal distribution for an exceedance of the overall system $p = 10^{-4}$.

After interpreting the Equation 5.2 along with its plot in Figure 5.2 we can conclude that the exceedance probability of the maximal observations diminishes

when the number of extremal data increases.

Deciding what observations are deemed extremes to later fit an EVT distribution is one of the main issues with EVT analysis [71, 69]. Moreover, this decision is relative and hard to automate [72, 23]. In a survey from 2012 on threshold estimation for EVT [73] the authors poll and synthesize several methods to figure out this selection of maxima. These methods comprise:

1. **Model Diagnostics:** They consists of inspecting the data visually with plotting instruments such as *mean residual life*, *threshold stability* plots or quantile plots [106]. Threshold stability for instance fit an EVT distribution iteratively changing the number of exceedances and the parameters variability is investigated. After analyzing the plots the maximum threshold up until which stability is held is chosen. The downside of these approaches is that they are hard to automate, they are *subjective* and require experts to analyze the data.

   Some of these instruments are also useful to investigate the asymptotic properties of the empirical observations and thus gain some insight *a priori* about what EVT distribution could be best fit to it.

2. **Fixed Thresholds and Rules of Thumb** apply either constants or arithmetic rules that can be really simple and automatic to apply. In the case of applying fixed threshold by means of constants, it can only be carried out on those problem domains where thresholds have certain meaning. Aside that, some rules of thumb are often applied such as $k = log(n)$ or $k = \sqrt{n}$ where n is the size of raw data and k the number of excesses. The downside of this approach is that for certain data the fitted EVT distribution may vary substantially i.e., shape parameter, if changing slightly the threshold or the block maxima size.

3. **Computational Approaches** resort to simulations to generate a greater number of observations so as to give confidence in the analysis. Occasionally, for some of these models to be effective prior manual analysis must

be done. Apart from simulation, there exists some heuristic methods orientated to minimize errors of the estimators. Allegedly, these methods impouse some restrictive assumptions that hinder a wider application.

5.1.4 Asymptotic Analysis of EVT distributions

Along with the choice of extreme observed data, the intrinsic asymptotic behaviour of the tail of the distribution produces a great *variability* depending upon how far we extrapolate on the tail. In fact, Fedotova et al. [86] gives evidence on how the pWCET varies around 766 times greater when using either Gumbel or Frechet for an extrapolation of 10^{-9}.

The analysis of the tails is studied using model diagnostics. To name a few of these tools orientated to study the asymptotic behaviour: *mean excess function* [71], *coefficient of variation* [85], or typical *Quantiles Plots* (QQ Plots) [71]. Amongst them, perhaps the most popular and simple to understand is the QQ plots which plots the quantiles of specific distribution diagonally and on top of data observations are plot to check whether they share the same asymptotic properties. Occasionally, this also serves to check the GoF by checking how aligned both distributions are.

To become aware of the importance and features of the asymptotic behaviour of the EVT distributions consider Figure 5.3. In this figure, the exponential distribution is taken as a reference and is plotted along with all GEV and GPD distributions. From the asymptotic analysis point of view, they are worth categorizing so as to distinguish how much its CCDF approach to 0 as the tail tends to *infinity*.

Painted with blue and purple colours we can notice *light-tail* distributions which are characterized with a *negative* shape parameter ($\xi < 0$). Their tails of CCDF are the ones which decays more rapidly to 0. Their tail are proved to have a finite endpoint [85, 71]. Amongst GEV distribution, the Weibull one has a light-tail. Next, *exponential-tail* distributions are displayed diagonally. They are identified

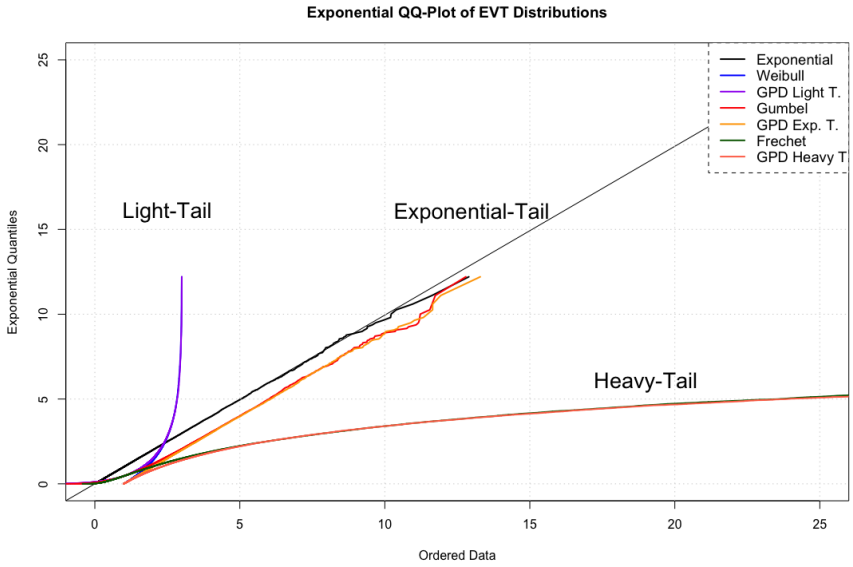


Figure 5.3: Asymptotic behaviour of EVT distributions plus the exponential one.

by shape parameter equals to *zero* ($\xi = 0$). Our aforementioned Gumbel distribution falls into this type of tail. Distributions exhibiting exponential behaviour are said to reach 0 at infinity [85, 71]. Last but not least there exists a category with the so-called *heavy-tails* whose shape parameter is *positive* ($\xi > 0$). These distributions converge to 0 even slower than the exponential ones. Frechet distribution is the distribution exhibiting a heavy tail amongst those integrating GEV distributions.

5.2 Goodness of Fit Tests for the Upper Tails of the Distributions

Apart from a strict application of EVT, some other authors have attempted to combine EVT with others distributions [73]. Chief amongst them is the work on tail tests [88, 24]. The motivation for this work stems from a structural reliability problem where confident predictions about the robustness of bridges were required. Due to the economic costs of destroying such a structure they had to

predict to what extent the structure would stand the large stress by observing small stress data.

This approach is certainly unusual since the vast majority of the GoF tests are focused on the central part of the distribution. Notwithstanding, in our problem domain we aim for getting confidence in the part of the tail of the distribution where no observations are available. Yet, the tail of the distributions is of interest as it contains the extreme predictions. This part of the tail has been named the *extreme upper quantile* which is usually greater than the maximum observations [24].

Definition 5.2.1. *Be n the number of observations of the sample, be $p_n \leq \frac{1}{n}$ and be $\bar{F}$ the CCDF. An extreme upper quantile x_{p_n} is the quantile $(1-p_n)$ such that $\bar{F}(x_{p_n}) = p_n$.*

Unlike EVT, which fits the aforementioned semiparametric distributions, tail tests use parametric distributions and thus it uses the entire sample. This difference is particularly relevant on those problem domains where a few number of observations are available or the parameters of the distributions are meaningful for scientists [107]. As a consequence a probabilistic analysis based upon tail tests does not require as much data as EVT.

Traditionally, GoF tests have compared empirical data against the analytical results of the distribution. By contrast, the goal of tail tests is deliver a GoF but with having no observations to compare against. To achieve this tail tests resort to EVT and compares the predictions of $\bar{F}$ against the ones from EVT. To give confidence on this conclusion tail tests require to validate two hypotheses for $\bar{F}$ [88].

1. That usual GoF tests like Anderson-Darling or Cramér-von Mises check whether central part of the distribution provides with an acceptably good GoF (usually at the standard confidence level $\alpha = 0.05$).

2. That $\bar{F}$ belongs to a so-called *Maximum Domain of Attraction* (MDA).

The second hypotheses may sound more unfamiliar and is detailed as follows [71, 24].

Definition 5.2.2. *Be X_n the largest observation of the ordered sample $\{X_1, \ldots, X_n\}$. The Cumulative Distribution Function, F, is said to belong to a MDA if there exists two deterministic sequences t_n and $s_n > 0$ such that $(X_n - t_n)/s_n$ converges in distribution as $n \to \infty$ to a nondegenerate random variable.*

More precisely, the conditions that tell whether a distribution belongs to a MDA can be found in [71]. Luckily, some distributions have already been proven to belong to some MDA [71, 24]. From the tail tests perspective two domain of attractions are of relevance. Firstly, the Gumbel MDA and secondly, Frechet and Weibull.

- **Distributions belonging to Gumbel MDA:** Exponential, Weibull, Normal, Log-Normal and Gamma.
- **Distribution belonging to Frechet or Weibull MDA:** Pareto, Cauchy, Burr, Log-Gamma, Beta, Student and Khi2.

With respect to tail test, there exists two versions. Namely, ETT [88, 107] which is applied when F is in Gumbel MDA and was devised first and the *Generalized Pareto Distribution Test* (GPDT) [24] when F belongs to any of the other GEV domains.

The core of the tail test lies on the comparison of two parameters, $\hat{x}_{param}$ obtained from the parametric distribution, F, and $\hat{x}_{ET}$ or $\hat{x}_{GPD}$ which is calculated from the empirical data. The smaller the difference between $\hat{x}_{param}$ and the corresponding estimator the greater the confidence of the predictions of the tail is. By tail we refer to the tail of $\bar{F}$, denoted from now on as $F_{\hat{\theta}_n}$.

Unfortunately, given that these estimators employ EVT underneath, they do not get rid of the issue of selecting the maximal observations by using PoT method. Fortunately, because of the fact that parametric distributions are used, a computer data generation can be performed and thus this analysis can be delivered

automatically. The outcome of such a computational approach is a lookup table like the one depicted in Table 1 in [24]. These tables indicate for a certain distribution and certain sample size, n, how many observations to take, k_n, beyond the selected threshold, u_n. The parameters of the tail test are defined as follows:

$$\hat{x}_{param} = F^{-1}_{\hat{\theta}_n}(1 - p_n) \tag{5.3}$$

$$\hat{x}_{ET} = X_{n-k_n} + \hat{\sigma_n}\ ln(\frac{k_n}{np_n}) \tag{5.4}$$

Equation 5.4 defines the ETT-based estimator. Inside this equation X_{n-k_n} stands for $(n - k_n)$th observation from the ordered sample $\{X_1, \ldots, X_n\}$. It is worth noting that $u_n = X_{n-k_n}$. Lastly, $\hat{\sigma_n}$ is the average of excesses defined as $\hat{\sigma_n} = \frac{1}{k_n}\sum_{i=1}^{k_n}(X_i - u_n)\ \forall X_i > u_n$.

$$\hat{x}_{GPD} = X_{n-k_n} + \frac{\hat{\sigma_n}}{\hat{\gamma_n}}\left[\left(\frac{k_n}{np_n}\right)^{\hat{\gamma_n}} - 1\right] \tag{5.5}$$

On the other hand, in Equation 5.5 the same definitions hold. The only exception are the parameters $\hat{\sigma_n}$ and $\hat{\gamma_n}$ which corresponds to scale and shape parameters of a fitted GPD.

5.2.1 Tail Tests

Having described the estimators and the principles integrating the tail tests it is appropriate to enumerate and define the different types of tail tests. These tests examine the null hypothesis, $\mathcal{H}_0$, which is defined as: *the tail of the parametric distribution, $F_{\hat{\theta}_n}$ provide with reliable extreme predictions compared to EVT* [24]. By contrast the alternative $\mathcal{H}_1$ concludes that these predictions are not confident.

To arrive to this conclusion tail tests compares the difference of the parametric and semiparametric extreme upper quantiles: $\hat{x}_{param} - \hat{x}_T$ (Note $\hat{x}_T$ is a generic term for either $\hat{x}_{GPD}$ or $\hat{x}_{ET}$). This difference is then checked whether it belongs to a Confidence Interval (CI) with a certain significance level α, CI_α. If this is the case $\mathcal{H}_0$ is accepted. Otherwise, the hypothesis is rejected. There exists 3 different versions of the tail test and they trade-off accuracy and computational

complexity [88, 24]. In a scale from more accurate and greatest computational complexity to the least accurate with the least complexity we get: *Full Parametric Bootstrap* (FPB), *Simplified Parametric Bootstrap* (SPB) and the *Asymptotic* version. Reportedly, the asymptotic version is said to be disappointing [24] in finite sample situations. By contrast, Full Parametric Bootstrap was deemed the best one but concerns were raised w.r.t to the tractability of the approach.

Bearing in mind that these works were done in the early 2000s and thus computer hardware was not as fast as it is nowadays we believe that Full Parametric Bootstrap could be useful for our experimental setting. After testing the version we found that it was feasible to include this version of the test.

The essence of this test is to create a confidence interbal based on an empirical evaluation with data generated by Monte Carlo simulation. By assuming $\mathcal{H}_0$ we generate N samples of n size from the parametric distribution. Then from each of these N samples a estimator of the $\hat{x}^*_T$ and $\hat{x}^*_{param}$ are calculated followed by the computation of its difference $\delta^*_n = \hat{x}^*_{T;n} - \hat{x}^*_{param;n}$. The resulting vector of δ^*_n is sorted and then $[N\alpha/2]$ greatest and smallest are removed. As usual α denotes the confidence level. Thus the confidence is defined as:

$$FPB.CI_{\alpha;n} = [\delta^*_{N\alpha/2,N}, \delta^*_{N(1-\alpha/2),N}] \tag{5.6}$$

This leads to the next check:

$\mathcal{H}_0$ is accepted with a confidence level α if $\hat{x}_{T;n} - \hat{x}_{param;n} \in FPB.CI_{\alpha;n}$

These definitions conform the tail test that are based in Parametric Bootstrap evaluation.

5.2.2 Asymptotic Analysis of Parametric Distributions in GEV MDA

According to the premises in Section 5.1.4 the asymptotic behaviour of the parametric distribution belonging to a MDA deserve being analyzed asymptotically. That is why we aim for computing a *sound* exceedance probabilities that are later used to represent the level of uncertainty.

In our experimental setting we attempted to include all the reasonable distributions we were aware of which met the requirements of the tail tests. Unfortunately, due to technical problems we couldn't include Burr, Student and Khi2 and for this reason we skipped them in the asymptotic analysis.

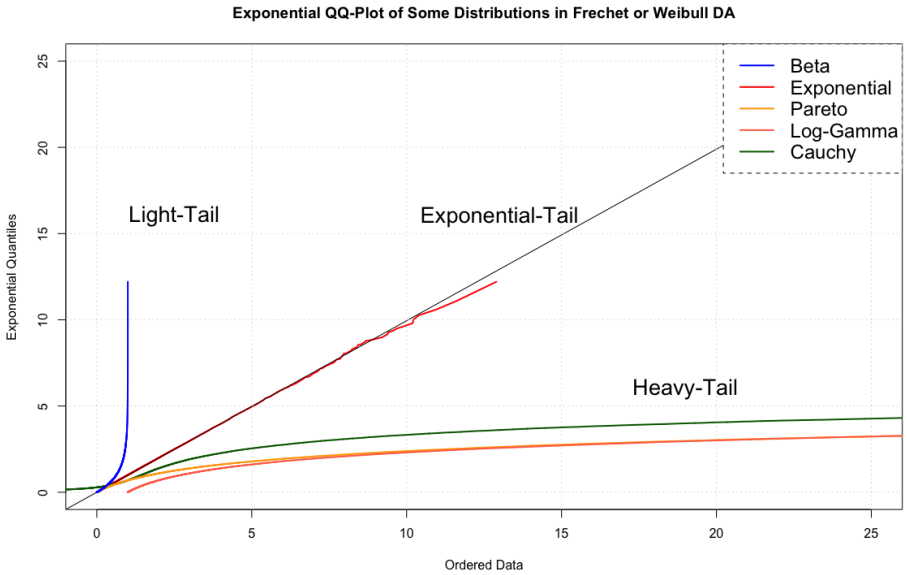


Figure 5.4: Asymptotic behaviour of some distributions in either Frechet or Weibull DA. Exponential distributions included just by reference.

Figure 5.4 displays the asymptotic behaviour of the distribution whose appropriate test to apply is GPDT. The only distribution exhibiting a light-tail is Beta. The other three all exhibit a heavy-tail distribution.

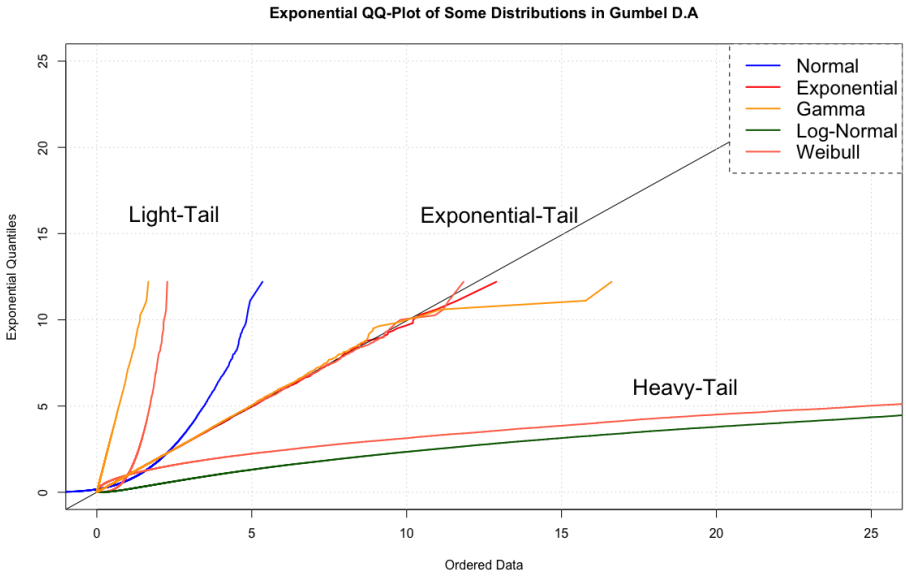


Figure 5.5: Asymptotic behaviour of some distributions in Gumbel DA.

Additionally, the distributions corresponding to the ETT are displayed in Figure 5.5. Normal distribution is the only one endowed with a light-tail. It's not surprising that the exponential distribution has an exponential tail. By contrast, Log-Normal distribution has a heavy tail. The unusual feature of this plot is the asymptotic behaviour of Gamma which can be light or exponential and the Weibull distribution which can be any. This certainly may hamper our TT-based MBPTA if these distribution are chosen as good fit of the data. In such a case we should study the asymptotic behaviour of these distributions with the fitted parameters and then check whether we could add up together exceedance probability data.

5.2.3 Search-Based Fitting

An open discussion in the realm of EVT is what the best fitting method is [72]. After fitting of the parameters, the *checker* of the fit is the well-known *GoF* tests. As described in the literature survey, in the particular case of EVT, we need extremal data to verify the results of the tails. A prediction of an arbitrary CCDF of say 10^{-4} needs 10^4 observations from the fitted data to check the results. Non-

visual GoF tests normally return a p-value from which to check the hypothesis whether the distribution is a good fit for the data or not.

Along with GoF, there exists some score rules [75] advocated to measure the error of extreme predictions i.e, difference between an analytical and empirical data . This process of trial and error can be contemplated from a search perspective where the objective is to minimize the difference between the analytical and empirical model by finding optimal fitting parameters. So much so, that some authors have named *Maximum Goodness of Fit* [89]. Aside that, others works centered in EVT fitting [75] have used a Simulated Annealing whose objective was to optimize score metrics. They concluded that this SBF gave the best result although it did not change much the shape parameter of the GEV distributions.

The gap to be filled here is to combine tail tests theory with SBF so as to maximize the number of times tail tests are passed and thus have a reliable predictive power thanks to the tail of the distribution. Obviously, for SBF be effective proper guidance must be supplied. This guidance must synthesize the objectives and requirements of the tail test.

A plausible objective function to be *minimized* is displayed in Algorithm 19. Considering the vicissitudes of the tail tests a first requirement would be to identify fitting parameters that achieve an acceptably good p-value in the GoF of the central distribution. This is computed in Line 3 of Algorithm 19. After that, the result is checked, if it happens it is not significant a feedback is returned bearing in mind that: 1) our objective is typically the $p.value \geq 0.05$, 2) the fact that it is a function to minimized and 3) this feedback must be substantially greater than the feedback of the tail test.

Next, in lines 8 and 9 the parameters of the tail test are computed. For the sake of simplicity we didn't differ from ETT and GPDT but in practice it is compulsory to do so. The decision in line 10 is the best case scenario as the tail test and its assumptions would be satisfied. In this case, just the feedback of the difference of the extreme upper quantiles is returned. In case TTs fails the absolute value of

Algorithm 19 Objective function to pass TT

```
1: function TAIL_TESTS_OBJECTIVE_FUNCTION(X, θ, α)
2:                 ▷X is the data, θ the distribution and parameters, α sig. level
3:     central_GoF_p.value ← Anderson_Darling_GoF(X, θ, α)
4:
5:     if central_GoF_p.value < α then                                  ▷Bad GoF
6:         feedback ← (1 - central_GoF_p.value) × 10^4
7:     else
8:         Δx ← x̂_T − x̂_param
9:         FPB.CI ← Compute_FPB(X, θ, α)
10:        if Δx ∈ FPB.CI then
11:            feedback ← |Δx|
12:        else
13:            ▷SCI and ICI denotes the upper and lower bound of the FPB.CI
14:            feedback ← |Δx| + |FPB.SCI − FPB.ICI|
15:        end if
16:    end if
17:    return feedback;
18: end function
```

the difference of the CI interval is added to the difference of the main parameters. The reason why applying absolute values is because this objective function is to be minimized and Δx and CI may be negative whereas the *global minimum* is $\Delta x = 0$.

5.3 Evaluation and Case Studies

On the one hand, we have obtained execution time data from the evaluation of different TGs in a time-deterministic architecture. After comparing them some *known uncertainty* has become apparent. Neither test generation nor the architecture were orientated to meet probabilistic analysis demands which gives us *unbiased* data. Coincidentally, maximizing the HWM is useful to verify the prediction of extreme events using scoring rules [75].

On the other hand, this chapter has described the theory behind tail tests as well as proposing a new fitting for this new probabilistic analysis. The objective of

this section is to integrate both parts so as to tackle contributions (b) and (c) from the introduction of this chapter. Additionally, given our critique on how the exceedance probabilities are estimated in EVT-based MBPTA, we aim for calculating reasonable exceedance probabilities depending upon the distribution, based on known uncertainty of the data. Finally, the experiments should shed light on how this exceedance varies by changing the number of observations in each sample.

The two cases where tail test-based MBPTA can not be applied are those where local HWM matches the HWM or when statistical tests fail either the central GoF or the tail test. In order to check what probability distribution gives the best fit, all the distribution discussed in Section 5.2.2 are tested.

An R script [108] is implemented to automate the statistical analysis. For the central GoF we chose Anderson-Darling test which is one of the advised tests for TTs [88, 24]. When it comes to meeting the requirements of the tail tests, there exists 4 plausible outcomes:

1. That the central GoF test is passed and the tail test. This is the most favourable result.
2. That the central GoF is not passed and TT would not be valid regardless of the result.
3. That the central GoF is passed but not the TT.
4. Neither of them are passed.

In those cases where the tail tests requirements fail a SBF method aims at bailing out the MBPTA by computing valid fit parameters. To achieve that, the Simulated Annealing from package [109] was used with its default parameters and a time limit of 40 seconds per optimization. In reference to the guidance function the function employed was outlined in Algorithm 19.

Regarding the tail tests the Full Parametric Bootstrap version was performed with $N = 200$ samples following the guidelines from [24]. Unfortunately, the

number of excesses of the look-up table of the tail tests work [88, 24] do not include the size sample we are using. Despite so, this decision was taken with rule of thumb $k = log(n)$ as embraced by Diebolt et al. [110]. Experiments suggest that this rule works well given the great number of samples (> 1000) [110].

The number of synthetic observations of the parametric distribution (n) is set to the length of the empirical sample. The actual sample size is varied in a iterative fashion and is increased around 10% in each iteration. The sampling of the real data is done *randomly* enabling repetition of the data. The reason for doing so is because during several experiments we witnessed how the p-values of the GoF tests often change substantially by varying the number of observations.

5.3.1 Insertion Sort

The first of the case studies analyzes the traces of the insertion sort case study whose ETP was depicted in Figure 4.26. From that figure it seems there is no uncertainty for CBTG so its observations were skipped. The first samples to be analyzed are the ones given by the RTG.

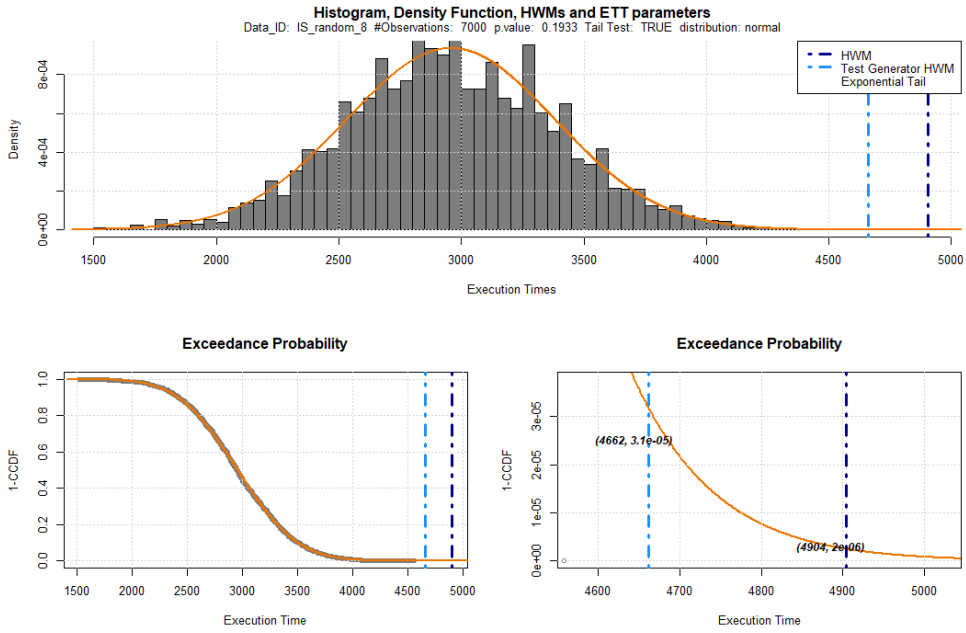


Figure 5.6: Normal distribution fit to 7000 observations from RTG. The test trial is the number 8.

Figure 5.6 displays the fit of a Normal distribution that passes both central GoF test and ETT. The dark blue stands for the location of the HWM whereas the light blue locates the local HWM. The known uncertainty region at the bottom right displays the gap to be filled by the tail of the Normal distribution. The HWM is located for $2 \cdot 10^{-6}$ exceedance probability for that tail in particular.

A more general view of the statistical analysis results is depicted in Figure 5.7. To start with all the cases have some form of known uncertainty. For the sample of 1000, only in 4 cases all the tests were passed. The Simulated Annealing allowed to include 1 out 4 successful cases for the analysis that would have been rejected

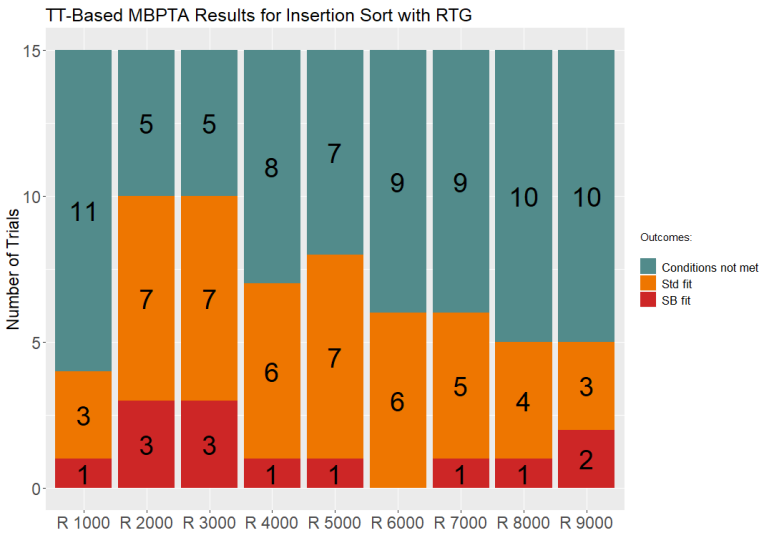


Figure 5.7: Summary of Results of TT-Based MBPTA for RTG.

otherwise. 2000 and 3000 samples distributions achieve 2/3 of successful fitting. After that, an average of 7.4 (49.3%) trials are successful for the appplication of the new tail test-based MBPTA. In these samples the 19.35% of the cases the fit was performed thanks to SBF.

Figures 5.8, 5.9 and 5.10 contain the calculation of the exceedance probabilities in a boxplot format. In Figure 5.8 we can observe the overlapping of the Normal and Weibull distribution. Moreover, we can display the difference in the exceedance probability since they are different probability distributions and probably the tail of each exhibits different asymptotic behaviour. Figures 5.9 and 5.10 display only the Normal distribution as the best distribution to fit the sample. The interesting feature is the fact that the average of *all* exceedance probability distribution but the one with 1000 observations range between 10^{-7} and $2 \cdot 10^{-7}$. Speculative speaking, this observation suggests that this form of MBPTA could more independent of the number of observations than the EVT counterpart.

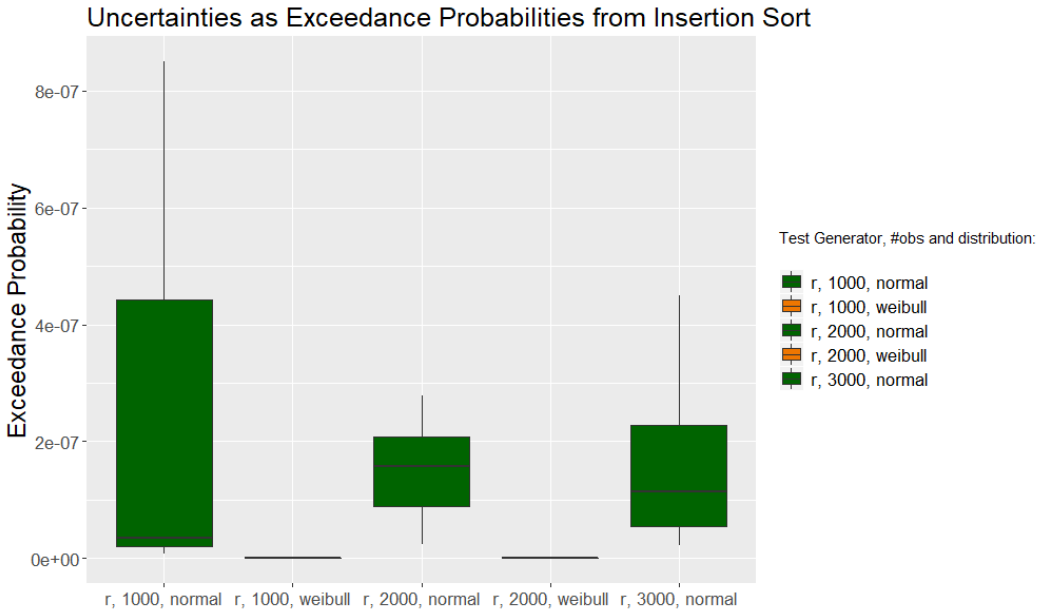


Figure 5.8: Exceedance probabilities calculation for RTG with samples ranging from 1000 to 3000

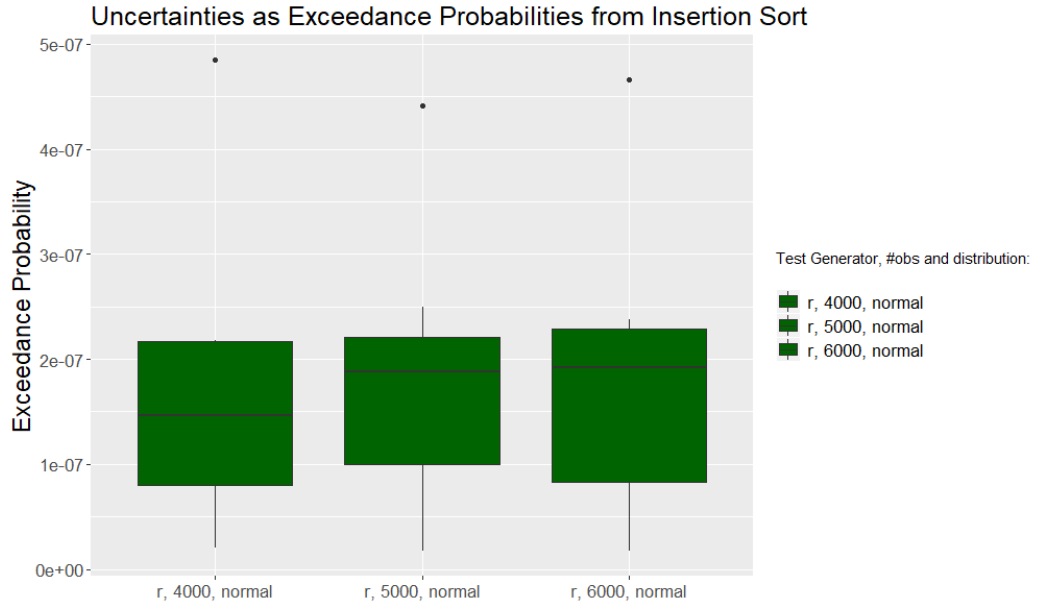


Figure 5.9: Exceedance probabilities calculation for RTG with samples ranging from 4000 to 6000

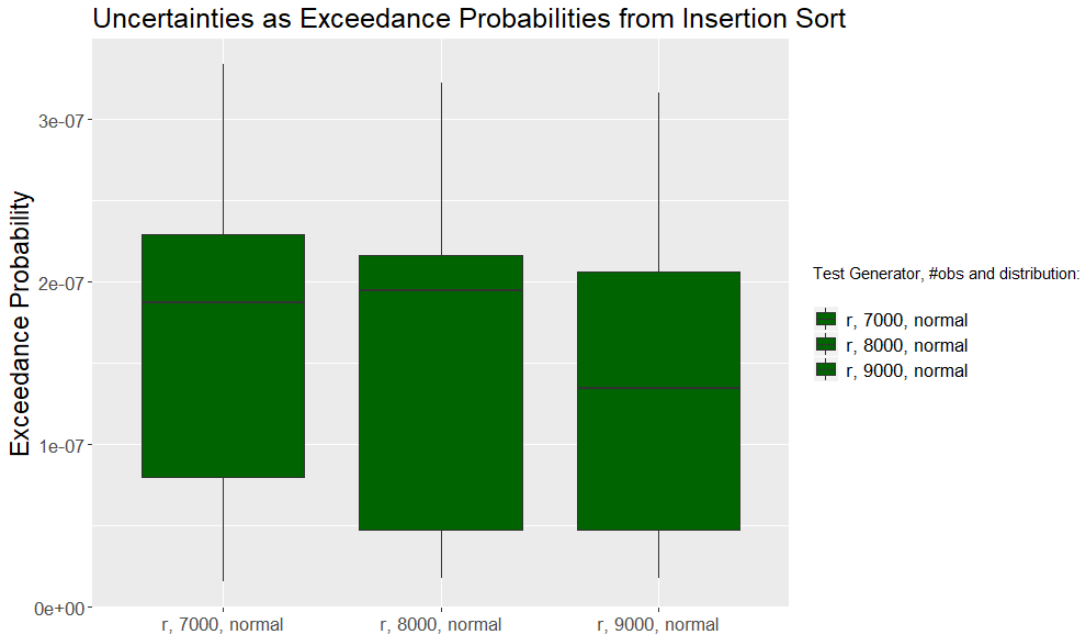


Figure 5.10: Exceedance probabilities calculation for RTG with samples ranging from 7000 to 9000

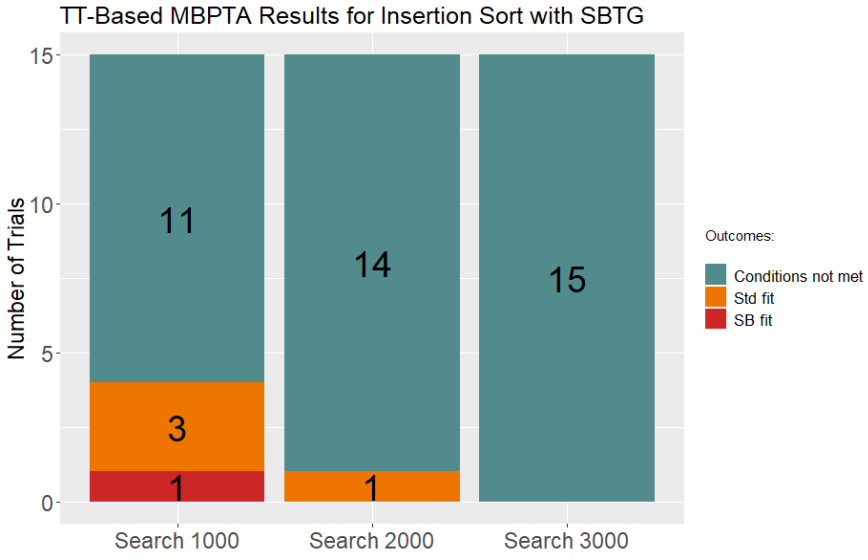


Figure 5.11: Summary of Results of TT-Based MBPTA for SBTG.

On the other hand, the execution times of the SBTG were also analyzed and its results are displayed in Figure 5.11. The vast majority of theses cases did not perform the required statistical tests. More accurately, 11 out of 15 cases for the 1000 observations sample, 14 for the 2000 and all of them for the 3000. In the 1000 case, the data allowed the fitting of 4 distributions and one of them was boosted by SBF. In the 2000 sample only 1 distribution could be fitted using the standard fitting method. From 3000 till 9000 the statistical conditions were not met.

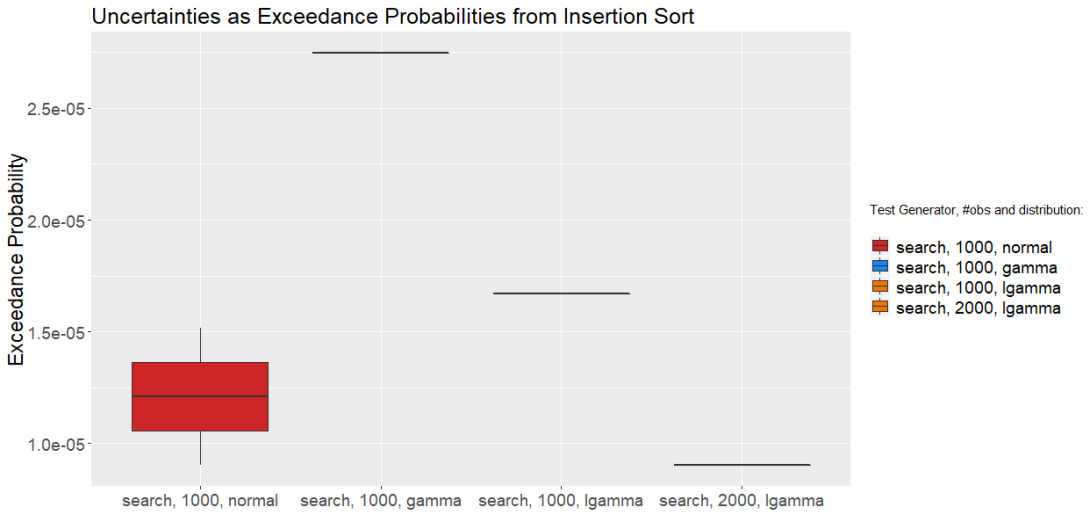


Figure 5.12: Exceedance probabilities calculation for SBTG

In terms of exceedance probabilities of the SBTG the data are depicted in Figure 5.12. This time Normal, Gamma and Log-Gamma distributions were found appropriate for fitting. Apparently, Normal distribution collected the most observations according to the boxplot. The 1000 observations case fitted all of these three distributions whereas in the 2000 one only the Log-Gamma could be fitted. The variability of the exceedances is greater than the previous case since in this case it ranges around $3 \cdot 10^{-5}$ and $2 \cdot 10^{-5}$. Unfortunately, not many distributions generated by CBTG in this case study could be fit and hence we could not witness the change in calculation of exceedances as the sample varies.

This evaluation has shown how the proposed tail test-based could help at filling the gaps of the uncertainty and how SBF could remarkably contribute to meet the conditions of this statistical analysis. RTG has clearly been more suitable to generate friendly for distributions tail tests -based probabilistic analysis.

5.3.2 Quick Sort

In this second case study the results from the quick sort benchmark are analyzed. The ETP in the form of empirical CCDF is illustrated in Figure 4.14. Again, the CBTG left not known uncertainty to be covered by a distribution.

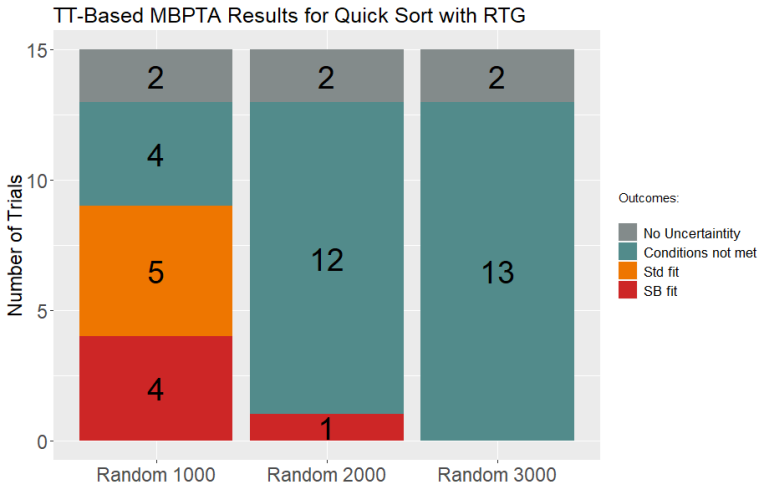


Figure 5.13: Summary of Results of TT-Based MBPTA for RTG.

Figure 5.13 displays the results for the RTG. In only 2 cases the HWM is hit and thus tail test-based MBPTA is irrelevant considering the available data of uncertainty. The first sample with 1000 observations achieves the greatest number of successes for the new MBPTA since in 9 cases a distribution is successfully fit. Next, the number of trials whose statistical conditions are satisfied decreases rapidly. In only 1 trial a distribution is fitted thanks to SBF. From 3000 observations onward tail test-based MBPTA was not possible to apply.

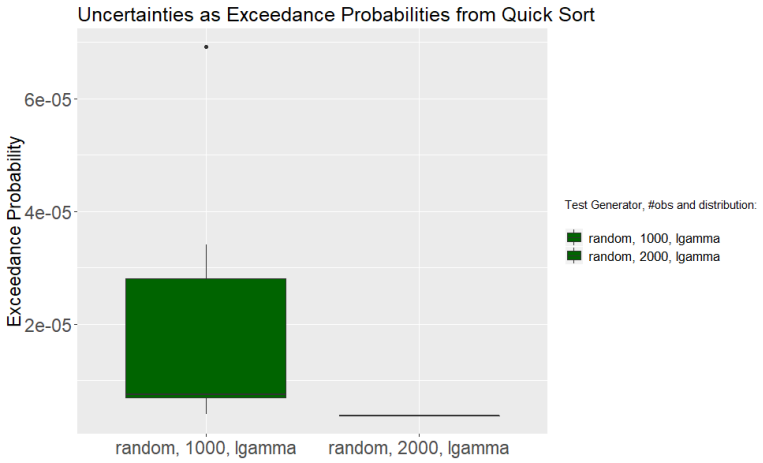


Figure 5.14: Summary of Results of tail test-Based MBPTA for RTG.

Figure 5.14 depicts the uncertainty estimates in the form of exceedance probabilities. Unlike previous case study which used mainly Normal distribution for the RTG, Log-Gamma distribution was found appropriate as a model to predict extremes. Reportedly, the estimations of the exceedances are in the order of 10^{-5} with this distribution in comparison to the order of 10^{-7} of previous case study and Normal distribution.

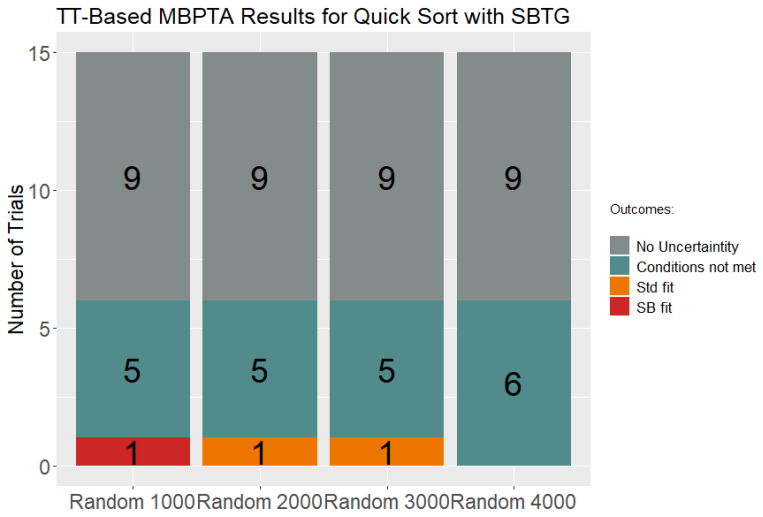


Figure 5.15: Summary of Results of tail test-Based MBPTA for SBTG.

Figure 5.15 displays the results for SBTG. The results show that only 6 samples from the trials did not attain the HWM and thus they were candidates to apply probabilistic analysis. In the 1000 data sample, only 1 could be fitted by the SBF. The cases of 2000 and 3000 only a distribution could be fitted using the standard method. The number of trials was different in this cases from the one fitted using SBF. This implies that the random sampling contributed in passing GoF tests by adding more data. These results are the opposite of rest of the results shown so far since the trend was generally that the more observations we add the harder the conditions are to apply tail tests result.

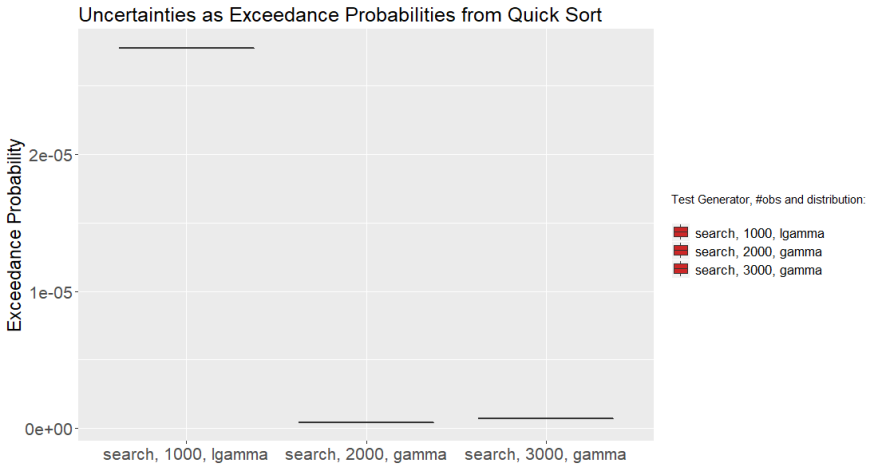


Figure 5.16: Summary of Results of TT-Based MBPTA for SBTG.

Figure 5.16 displays the data of the exceedance probability derived by using a Log-Gamma and Gamma distribution. They clearly show a great deal of variability between the Log-Gamma for the 1000 observations case and the Gamma for the rest. This is due to the fact the data came from different trials, the distribution is different and the fact that the asymptotic behaviour of these distributions is different.

Unlike previous case study, the data from quick sort benchmark often had no known uncertainty to motivate this new form of MBPTA. In this assessment, RTG still showed to provide with friendly data for probabilistic analysis in 10 out of 14 cases (71.42% of up to 2000 observation trial). For this data, SBF made

a difference in half of these 10 cases. On the other side, SBTG generated even less uncertainty to be covered by means of EVT and in only 3 cases out 18 (16.67% for up to 3000 data) tail tests could be applied. When it comes to exceedance probabilities calculation, Log-Gamma was deemed the best distribution from the fitting point of view.

5.3.3 Averse ETP for tail tests-Based MBPTA Application

Along with those cases which tick all the boxes for tail tests there exists some others where this new MBPTA could not be applied. Experimental results from Lima et al. [84] confirm what most statistician have guessed by looking at the empirical distribution: The level of discreteness of data and the dispersion of data clearly anticipates whether common bell-shaped distributions can be fit on top.

Such an assumption was not the case for the few execution times of our Needle-in Haystack case study in Section 3.4.1 and Hash Function in Section 4.2.6. A similar conclusion is arrived Landing Gear case study in Section which does not even contain loops. Autopilot program could be a candidate for tail test-Based MBPTA. However, after inspecting its ETP in Figure 1 (See Appendix B) there is no bell shape as it looks more like a stairs. Moreover, a considerable level of discreteness becomes apparent. The HWM was not very hard to attain and only two cycles separated the HWM of the CBTG and the closes HWM of the competitors. Similar conclusion can be deduced for the rest of the case studies presented in this thesis. Some of their ETP are also available in Appendix B.

Lastly, and industrial data was plotted and analyzed. This data came from an industrial benchmark running on an equivalent hardware to the one used for the experiments. This benchmark was tested using SBT with different fitness functions [18]. Its exceedance probability is displayed in Figure 5.17. Given its irregular shape and the challenging multimodal distribution these distributions could not be analyzed since central GoF tests would fail. Though this method is not entirely correct, we tried to analyze the second lump as a single distribution but still GoF tests did not accept any distribution.

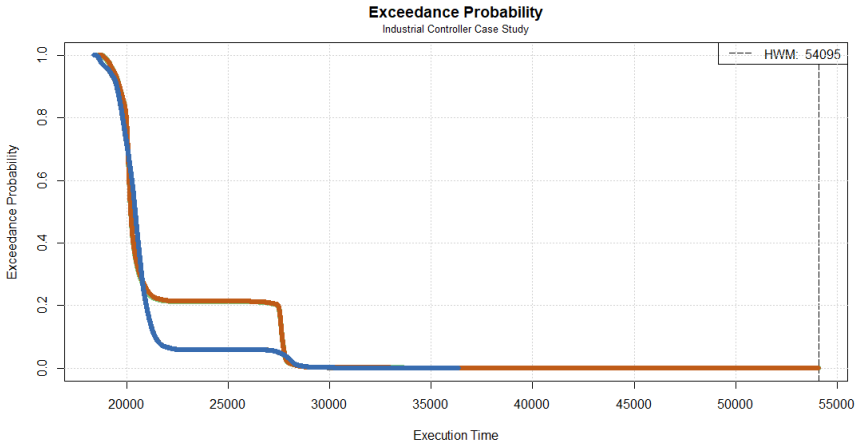


Figure 5.17: Execution time profile. Each trial corresponds to a different color.

Perhaps the middle point case between an stair type distribution and a bell shape was showed by Select k Largest and Linear Search case study. Select K largest execution time profile was depicted in Figure 4.2 and SBTG and RTG shows an CCDF close to a straight line. Unfortunately, none of these distribution met the requirements for tail test either. Linear Search shares some similarities with this ETP in Figure 3 but in this case CBTG shows a staircase pattern and RTG a relative straight line.

5.3.4 Threats to validity

As for threats to **internal validity** we would argue that even though the test generation from the cases studies did not target to be compatible with our approach, some papers have argued that the required data for MBPTA should come from

the system when deployed [23]. This may impact the confidence of the evaluation. However, we believe that RTG could give to some extent, a representative sample.

The other source of threat of internal validity comes from applying the rule of thumb to select the maximal data. This decision to the select observed extreme data for fitting is still contested by statisticians working with EVT [70, 68].

When it comes to threats to **external validity** we would point out, based on the example of the industrial software along with the few cases where the proposed probabilistic analysis could be applied, that this approach is not likely applicable to similar time-deterministic computer architectures. This is due to the divergence between the compatible histograms of probability distribution and the discrete and often shapeless ETPs of real-time software.

5.4 Summary

The aim of this chapter was to calculate more confident exceedance probabilities with the known uncertainty from case studies. This data may in turn be used as part of future work to derive a more confident pWCET.

To meet this objective a new form of probabilistic analysis has been proposed as a result of a critique to state-of-the-art EVT analysis. The main arguments of such a critique are:

- Real-world data may exhibit an ETP that may be hard to analyze using either parametric distribution e.g., Normal distributions, or (semiparametric) EVT distributions [24].
- Irrespective of the TGs approach, the implementation of the randomization may be unacceptable for safety and business reasons whereas the padding-based approaches to make an EVT friendly distribution may supply biased data to the EVT analysis.
- If pWCETs are derived using the CCDF then sound values of exceedance

probabilities must be calculated. Current approaches are oblivious to the impact of the selection of maxima and the asymptotic properties of the EVT distributions on the calculation of exceedance probabilities when in reality these two factors do matter [86]. Some works [87, 85] along with a brief evaluation have shown that both issues have a bearing on the estimation of this crucial data.

The novel tail tests for MBPTA display the following differences with respect EVT:

- Using parametric distributions such as Normal or Exponential distributions which are in any of the MDA of a GEV distribution.
- These distributions must pass a standard GoF in the central part of the distribution i.e., where there are observations.
- All the data is used for fitting and thus less data are required for the analysis.
- Though it uses EVT for the calculation of the tail tests, its selection of the maxima can be automated by performing computer simulation beforehand.
- SBF is used as a fitting method which helps at identifying fit parameters but an objective function must be defined.

The evaluation has checked the applicability of this new MBPTA to calculate exceedance probabilities. SBF is integrated to facilitate such application.

Results show that only in 2 out 10 cases where all TGs were applied, tail tests can be employed which reinforces the assumption that probabilistic approaches may not be suitable for our problem domain [80]. SBF fitted 26.5% of the cases of successful application of tail tests. In none of the cases CBTG provided with an ETP that met the requirements to be analyzed by tail tests using the distributions tail tests dictate. By contrast, SBTG and RTG showed some friendly ETP for the sort algorithms. Unfortunately, the samples between 2000 and 4000 observations generally set the limit of the application of this form of analysis as the addition of more observations generally damaged the central GoF test. The

satisfaction of this test is a necessary condition to apply tail tests.

The estimated exceedance probabilities ranged between $[10^{-7}, 7 \cdot 10^{-5}]$ and resulted from Normal, Weibull, Gamma and Log-Gamma distributions. The *known uncertainty* gap and the asymptotic behaviour of the parametric distribution had the greatest impact on the exceedance probabilities estimation.

Likewise, the ETP from vast majority of the case studies where all TGs took part was not suitable for our new MBPTA. The industrial benchmark displayed a multimodal distribution which made infeasible the fitting of the parametric distributions. A similar issue happened when Select K Largest and Linear Search were analyzed. Some of their empirical CCDF resemble to a diagonal line more than a bell-shaped distribution.

Lastly, threats of validity have argued that execution time data does not come the system when deployed, the debatical selection of extremes for the analysis and its low likelihood of the proposed analysis to be extrapolated to similar time-deterministic architectures.

Chapter 6

Conclusions and Future Work

The timing requirements of Real-Time Systems can only be verified as long as the necessary time data is available. The WCET of software tasks is arguably the most challenging time data to give confidence that these timing requirements are met. Because of the complexity of path and state coverage the WCET is normally unknown.

When it comes to the approaches to tackle this problem, STA has the basic disadvantage of portability to other hardware architectures as it requires hardware modelling. For this reason, MBTA, which collects execution time empirically, has been of our interest. This strand is only reliable as long as test vectors are provided for the performance testing. Unfortunately, these test vectors are often generated manually and have a considerable economic cost [16]. That is why automatic test generation techniques have become increasingly important.

SBT and CBT have been the main methods to decide test data. Both approaches were reviewed Chapter 2 as part of the literature survey and present the following weaknesses:

- ☞ Depending on the SBT implementation, this approach may struggle at triggering branches like equalities which depends on very specific data from the search space [29]. Even the most accurate approaches require a few thousands iterations to hit this kind of branches [58].
- ☞ Current approaches on CBT consists of an extrapolation of coverage criteria

of functional testing but they are restricted to branch coverage [40]. There is no CBT which takes into account WCET analysis requirements such as creating test vectors for promising WCET paths.

- ☞ The heuristics employed to derive constraints for CBT do not target to identify the most promising goals in the first place. In reality, attaining the required results as soon as possible is relevant for the industry [18].
- ☞ Current program slicing for CBT collects unnecessary data from the SUT to apply CBT yielding a more inefficient performance when search strategies are applied.

Our contributions address these weaknesses and we have put forward, firstly, an optimal program slicing to apply CBT that eliminates the inefficiencies of the current approaches. An evaluation of effects of the slicing have been provided. Secondly, BFS has been embraced as a method to build constraints for paths to try to identify the paths leading to the largest execution times in the first place. This method uses the notion of cost to discriminate paths. Another contribution has evaluated the accuracy of the cost estimation. Thirdly, an evaluation has been provided to observe the difference in terms of performance amongst the proposed CBTG, and a representative state-of-the-art of SBTG applied in MBPTA. In addition a RTG has been included as a way to evaluate the statistical significance.

Probabilistic approaches have become increasingly popular in the recent years in order to define a safety margin for MBTA. Chief amongst them is EVT. EVT-based MBPTA was surveyed in Chapter 2 as well. The following issues were identified [72, 23].

- ☞ Automation of any statistical analysis is an important objective for the industry. Unfortunately, a step in the EVT analysis hinders this automation.
- ☞ The calculation of a pWCET must be delivered using the notion of exceedance probabilities. Yet, current analysis dismiss important issues on this calculation such as the asymptotic behaviour of the EVT distribution.

Other works on probability theory have defined tail tests [24] which can lead to similar results to EVT. However, automation can be achieved. Additionally, SBF contemplates fitting any distribution as an optimization problem. This sort of fitting may be helpful to meet tail tests requisites.

Our last contribution has integrated this new form of MBPTA with the results from TGs with the objective of examining whether this approach could be applied and whether this approach is useful to calculate exceedance probabilities that can be used in the future to cover the resulting uncertainties of the TGs. If the latter is plausible, then a more confident pWCET can be estimated.

By including the automation of the protocol and observing the differences in the calculation of exceedance probabilities between our approach and EVT an improvement with respect the state-of-the-art arises. Next section revisits each contribution in detail.

6.1 Review of the Research Contributions

To recap the contributions, they are listed along the following subsections.

6.1.1 Constraint-Based Test Generation

1. To deliver a path composition algorithm based upon BFS that builds paths to be analyzed by using a CBTG. These paths not only lead to the largest execution times but these are computed first.

The description of the path composition algorithm was provided in Chapter 3. This approach implements a form of BFS which is guided by the notion of cost. This path construction targets paths with the greatest cost in the first place and is different from state-of-the-art approaches which some of them use constraint solver [38] to remove irrelevant constraints whereas others negate constraints [30] to maximize code coverage. Our approach is, to the best of our knowledge, the first one targeting paths which can be relevant for a WCET analysis based upon

MBTA.

A downside of BFS is its implementation. In our problem domain it was not enough to unveil the goal path but to also build a wide array of them to generate multiple test vectors. Moreover, not only we had to compute the required bound but also update it consistently when a path is built and checked. Lastly, handling the construction and deletion of new nodes to conform new paths has also a daunting implementation.

2. To evaluate the accuracy of the guidance given to the BFS in the form of cost.

The former heuristic is only applicable as long as some guidance is provided in the form of cost. These estimates are calculated considering the number of statements. An initial and relatively simple evaluation was delivered in Subsection 3.2.4 and it has strongly concluded that this estimation is not precise even for the relatively time-predictable architecture employed. In each case study where BFS and CBTG was used, the relation between cost and execution time has been portrayed. In the light of the data, we can conclude that this heuristic is not accurate.

Despite this burden, the real objective is to maximize the HWM as soon as possible. Even though the cost is not accurate, the systematic exploration of the path tree has mitigated this effect and the resulting HWMs do not generally underestimate the global HWM.

3. To devise an *optimal* program slicing for MBTA which eliminates the inefficiencies of current approaches reducing the number of paths to analyze.

This contribution was also given in Chapter 3. The optimal program slicing becomes apparent with respect state-of-the-art [40] when the SUT contains some statements or predicates which do not impact the flow of the program or these

can not be controlled with the input data. The result is our notion of CG i.e., a graph which stores the minimum data to apply CBT.

4. To evaluate the effects of the slicing in the run-time of a CBTG and discuss the impact on the test generation.

A discussion and small evaluation about the effect at generating test vectors for the CBTG is contemplated is Subsection 3.1.4. In a nutshell by not applying the slicing a CBTG would generate less test vector as a result of unsound constraints or the test vector could include unsound input data with an unsound range.

The effects on the run-time is a more extensive problem. Results along case studies have shown that even a small percentage of slice removal has *always* had a statistically significant reduction on the run-time of the BFS. This is because the iterative and tree exploration process of the paths construction. However, it is worth noting the measurement noise as a result of the complex hardware architecture of the host computer. Particularly, results with a low run-time i.e., less than one second. However, our priority in terms of contributions is to maximize the HWM on the embedded target and not necessarily achieve the best performance of the test oracle process running on the host.

5. To show how this test process can be used to reduce the pessimism if an hypothetical CWCET is composed by infeasible paths by detecting such paths.

The fourth contribution was addressed in Chapter 4. The surveyed instrumentation-based Hybrid approaches [35] builds a CWCET by looking at the structure of the code and not the flow of data. This implies the presence of pessimism in the CWCET as a consequence of the infeasible paths. Its detection is left to the developer who is in charge of writing annotations. Because of our data flow analysis and the use of a constraint solver our approach can detect these infeasible paths and reduce the pessimism of the CWCET. A case study using of the examples

of an industrial tool [35] was used. According to our estimation the pessimism of the CWCET was reduced a 47.2% with this optimization. Nevertheless, more case studies are needed in this respect and effects such as uncontrollable branch removal may turn this approach infeasible.

6.1.2 Comparison of Test Generators

Perhaps the most significant contribution is:

6. To evaluate the results and performance of the proposed CBTG process against state-of-the-art SBTG and a Random Test Generator (RTG) where the main objectives are to maximize the HWM and attain results promptly.

Later on, this contribution has been refined into the following questions, which were applied to 10 case studies from which 9 were equipped with statistically significant tests.

- **Research Question 1 - Effect of the test generators on the HWM**. Experiments on Chapter 3 has shown that our CBTG provides, in most of the cases, a HWM that matches the global HWM. However, Certyflie case study concluded that this is not the cases because of a statistically significant underestimation of the HWM. The reason for this behaviour is that our CBTG is unable to generate operands for some arithmetic operations so as to inflate the execution time of some blocks of code. This is due to the fact that the constraints representing SUT does not encompass all the aspects impacting the execution time.

  Aside that, experiments on Chapter 4 have demonstrated how the HWM unveiled by the CBTG matches the global HWM with the exception of Select K Largest. Still, threats to validity have argued how this benchmark may not be representative of Real-Time software.

A different question is how CBTG performed in relation to the other two TGs and most importantly SBTG. Firstly, there is a threat to validity because of the partial reproduction of state-of-the-art works in SBT. Our aim was to reproduce a representative state-of-the-art SBT in the context of MBTA and we argued how we would not need instrumentation for test generation. On the contrary, the programming language employed on the benchmark may have contributed to give good results to SBTG and RTG on account of the reduced range of variables.

Aside that, results have concluded that the HWMs unveiled by CBTG are only statistically significant greater w.r.t SBTG in Insertion Sort and Hash Function case studies. Hash function results were also matter of discussion in threats to validity. Reversely, the same number of case studies i.e., Certyflie and Select K largest, the SBTG beat CBTG. In the middle ground, the results of the HWM were not claimed to significantly different in 3 case studies. The low percentage of feasible paths which reduced the search space, along with the fact that in Binary Search case multiple test vectors could achieve the global HWM, have contributed to the collection of similar HWMs. In conclusion, there is no strong evidence to argue that one approach is generally more effective than the other at maximizing the HWM.

- **Research Question 2 - Effect of the test generators on the execution time distribution**. The case studies have shown how in general the effect on the execution of each test generation is generally significant. In RC Car and Hash function however this was not the case most probably because the low cardinality of the set of execution times of the benchmarks.

  Threats to validity have argued how the requirements of this test removed a good chunk of evidence from SBTG and RTG.

- **Research Question 3 - Effect of the test generators on the wall time** or when the HWM is observed during test generators execution.

Results shown in the case studies have demonstrated how in *all* the 5 cases where the global HWM was achieved by the CBTG and one of the other two TGs, CBTG has *always* spotted it in the first place and arguably in *all* the trials thanks to its determinism. As a consequence, we believe it is safe to conclude that our CBTG provides the best wall time.

- **Research Question 4 - Effect of the test generators on the efficiency (η)** which is expressed as number of test vectors produced per time unit.

  The evidence of the case studies has demonstrated that in 6 cases the efficiency of the CBTG was significantly lower than the other two counterparts. SBTG and RTG normally exhibited a similar efficiency. Therefore, we concluded that the efficiency of the CBTG is generally lower than the ones from SBTG and RTG.

- **Research Question 5 - Effect of the test generators on RAM memory usage.** The data exhibited in 8 case studies have concluded that the RAM usage of the CBTG is statistically significant with respect to the other two TGs. Furthermore, SBTG and RTG are concluded to have an equivalent RAM usage according the statistical tests.

- **Research Question 8 - Effect of the test vector from the Mälardalen benchmarks in the HWM.** The 3 case studies provided have illustrated how in only one of them the test vector provided generated a sound HWM. These results epitomize why it is very difficult to anticipate the actual execution time by alluding to the code only.

6.1.3 Probabilistic Analysis

The contributions on MBPTA are listed below.

7. To formulate a novel tail tests-based MBPTA advocated to provide full automation and calculate tighter pWCETs.

8. To evaluate the applicability of such an analysis for the resulting *unbiased* execution time data from different TGs as well as figuring out what are reasonable exceedance probabilities using the HWM data representing *known uncertainty*.

9. To assess how search-based fitting may help at the application of tail tests based MBPTA.

The former contributions were tackled in Chapter 5. After observing the ETP of the case studies we noticed that some form of known uncertainty emerged. This notion of known uncertainty is understood as the difference between the local HWM of an arbitrary TGs and the global HWM. In order to contribute on how this gap could be covered probabilistically and evaluate the application of alternative analysis, we tried to employ a new way of MBPTA based upon tail tests. Our experiments targeted the calculation of exceedance probabilities to be used as part of future work when there is unknown uncertainty and try to give more confident estimates of the pWCET.

As for the probabilistic evaluation a central debate is the whether the properties of the empirical real-world distributions (or part of it) are friendly with either parametric or EVT distribution. Moreover, even though this data is prone to pass GoF tests, there exists further statistical conditions to be met e.g., independence. Without studying some representative industrial data sets, it is hard to provide with a probabilistic-based solution. Assuming that such a solution there exists.

By using tail tests, only a subset of parametric distributions can be used as their extremes converge to some EVT distribution. Once these distributions are chosen they must pass a standard GoF test in the central part i.e, where observations are available. Next, tail tests are in charge of arguing about the confidence in the part of the right tail where no observations are available by comparing these results with an hypothetical EVT tail.

A core difference between tail tests-based MBPTA and EVT is the use of data. On the one hand, tail tests use the *entire* sample which leads to make the assump-

tion that the ETPs must fit candidate distributions. In truth, this restriction may be excessive for real-world execution time data. By using the complete sample, less data are required to perform extreme events predictions. However, in our problem domain is relatively easy to generate execution time data. On the other hand, EVT is only interested in the extreme observed data and only the ETP of this data must fit any of the EVT distributions. In our view, this imposes a less restrictive assumption on the entire sample than our tail tests analysis.

In terms of evaluation results, only a subset of the trials of the SBTG and RTG met all the conditions to apply tail tests and they happen to be the ones from the sort algorithms. Such conditions consist of the existence of known uncertainty and the conditions of the tail tests outlined in Chapter 5. SBF found distribution parameters that met the tail tests statistical conditions and added a 26.25% of successful cases that otherwise would not have been possible using standard fitting methods. Normal and Log-Gamma were the most common distributions successfully fitted. The estimated exceedance probabilities varied depending on the uncertainty and the asymptotic properties of the parametric distributions. To give a range they were between $[10^{-7}, 7 \cdot 10^{-5}]$. These probabilities are far from the others employed in EVT in a fixed way such as 10^{-15} [26] or 10^{-9} [86].

Our new MBPTA could only be applied to 2 out the 10 case studies we provided. In addition it could not analyze the industrial benchmark because of their dissimilarity between the resulting execution time data and the bell-shape distributions used by the tail tests.

Despite the fact that not many ETP could be analyzed by this approach, this does not necessarily mean that the introduced tail test-based MBPTA is not useful as there are some options as part of future work discussed in the last section.

6.2 Discussion about our CBTG

As stated in Chapter 3, our CBTG approach is mainly concerned with software in general and the notion of path in particular because of the portability argument. However, this is not the only contributing factor of the execution time as the hardware states may also have a big impact. This impact would entail having, to a greater or lesser extent, underestimations of the WCET as a result of achieving an optimistic HWM, which results from ignoring such hardware features.

Issues caused by the hardware, and thus impacting the execution time include:

- **Caches or scratchpads** could cause significant underestimations because even apparent simple and short paths could cause a large number of evictions and thus large execution times [44]. However, in our experimental setting, which included an embedded target with an instruction and data cache, we have not experienced any significant issues related to caches. Even though a number of test generators were used for a long time the relative exhaustive testing. Our approach could only handle the impact of these hardware units if there was a way to map the objective to the path cost. This could be defined somehow by the number of evictions, and thus our CBTG could target paths that maximize the evictions rather than having some code-related quantification.

- **Resource contention such as main memory access buses in multi-core systems**. It is probably the source of underestimation that could have the largest impact on the execution time because of the concurrent access to shared resources [103]. This is an ongoing problem in research and in the industry but this issue is beyond the scope of our work. To our knowledge, our CBTG could contribute to targeting different pieces of code running on different cores that theoretically are advocated to generate contention. This would allow the observations of large execution times. However, this would entail a careful analysis of the computer architecture and the availability of the appropriate code to generate this contention.

- **Arithmetic operations**. It may be the source causing the least significant

underestimation according to the evidence collected in the thesis. From the case studies - particularly Certyflie - this difference was around 0.35% between the HWM of the CBTG and the global HWM. However, this piece of code did not contain a large number of operations. By contrast, other pieces of code like the one presented in Appendix A probably would cause larger differences. These arithmetic operations encountered in blocks of code have unveiled how our CBTG is oblivious to these operations. As a consequence, it is unable to generate test vectors which generate operands which in turn trigger the largest execution times.

In this respect, the only way we can come up with for our approach is for example by adding constraints to avoid values of 0 and 1 in the test vector. This could cause larger execution times in some arithmetic operations [54].

When it comes to exercising the performance enhancement units, SBTG techniques may be more convenient as they can target some performance counters [31] which monitor for instance, the number of cache misses. Fitness functions employed in Khan and Bate work [31] using these counters have demonstrated the benefits at maximizing the resulting HWM taking into account certain properties of the program.

Nevertheless, taking into account the experiments performed in the thesis, the issue of caches has not represented a significant burden when switching between SBTG and CBTG which have generated the same HWM in most of the cases. Yet, those cases where the CBTG underestimated the results of the SBTG it has not been caused, to our understanding, to cache related effects but code structure related issues and the above described mathematical operations of the code.

The **limitations of our approach** become apparent in terms of scalability i.e., when facing a large number of paths. The first issue is motivated by the fact that a path tree with exponential complexity [27] is to be modelled and the intrinsic complexity of our BFS implementation. The program slicing presented here has

made a significant reduction in this respect by reducing this tree complexity as the data of the run-time mirrors. However, this reduction was still facing an unavoidable path explosion. Infeasible paths hinder the testability of a piece of software. In our framework, the number of test vectors is the same as the feasible paths analyzed. However, our heuristic does not check or prioritize feasible paths when building the path tree. As a consequence, it may struggle to generate test vectors in cases where the resulting path tree contains a greater number of infeasible paths.

Lastly, as BFS is guided by the cost of the path, which was concluded to be inaccurate even for the time predictable architecture we employed, this could mislead the search. As discussed in threats to validity in Chapter 4, this may cause the test generation to focus on a subset of paths that may not lead to the global HWM and to damage the confidence of the resulting HWM.

Regarding the sort of programs our approach can handle, two distinction are made:

- **On the abstract side** (some of these features are discussed in threats to validity in Chapter 3) programs exhibiting *multiple returns in the functions* cannot be analyzed successfully because of the way path constraints are concatenated. Having a single return point is a sound assumption for programs written in Spark [15] but not necessarily for the rest of the programming languages used in critical real-time embedded systems [19] i.e., C, C++, Ada and assembly.

  As reported in the summary of Chapter 4, our experiments have demonstrated how those loops having multiple exit decisions distributed in several points in the loop body damages the results of the CBTG. That is why the objective of maximizing the loop iterations by controlling the exit decision can be problematic. An equivalent assumption for loops is established in Static Timing Analysis works [7].

Programs whose *input data are not enumerates, integers, floating/fixed point or arrays* are not claimed to work with our approach.

Lastly, since the program slicing removes parts of the code that may not be relevant to the flow, the removal may contain some blocks of code containing operations whose largest execution time is sensitive to some operands. This was the case of Certyflie. Therefore, our CBTG may cause underestimations in those benchmarks containing a significant number of arithmetic operations not impacting the flow of the program.

- **On the technical side**, GenI is only compatible with a subset of programming constructs for Spark [15]. Extending the compatibility to other programming languages is warned to be an incredibly difficult and challenging task. As a starting a point a repository of representative SUT is required to test any static analyzer. Getting hold of these representative benchmarks is significantly difficult, if at all possible, because of intellectual property clauses for some of these proprietary and representative benchmarks. As stated in Chapter 4, according to the current version, it can only handle linear constraints even though the constraint solver SCIP [90] can handle a wide array of constraints e.g., polynomial.

6.3 Hypothesis Check

Returning to the central hypothesis of this work:

> The proposed Constraint-Based Testing process provides the best test generation process in terms of increasing the largest observed execution time and collecting this result earlier than state-of-the-art approaches. The novel probabilistic analysis is able to derive safety margin with an automatic process and its results are more confident than standard approaches.

The first line on test generators is mapped to research question 1 and 3 and were discussed in Subsection 6.1.2. In conclusion, the evidence provided indicates that our approach does not generally underestimate the global HWM. However, there is not enough evidential support to conclude that it achieves this better than state-of-the-art approaches. On the contrary, the evidence regarding research question 3 suggests that our approach collects, when it succeeds, the global HWM earlier than state-of-the-art techniques.

As for probabilistic analysis, experiments in Chapter 5 show that the proposed tail tests for MBPTA can only decide a safety margin *in a few cases*. Again, it is worthy of mention that neither the hardware, the benchmarks or the TGs had the objective to apply any form probabilistic data. Therefore the test generation did not target to be friendly with our probabilistic approach. Thus, unbiased data were analyzed to give relatively trustworthy results.

Nevertheless, our novel MBPTA enable the full automation thanks to the assumption it makes on the distributions which consider the entire sample. By using computer simulations on these distributions the number of excesses can be determined and stored in look-up tables. Then when executing our probabilistic

protocol this data is read enabling full automation.

The way exceedance probabilities are calculated gives confidence in the pWCET results. Data from the Insertion Sort and RTG, which allows tail tests for a wide variety of sample size, suggests that our approach is more independent to how the number of excesses is calculated than EVT. Thereby this condition allows a more accurate exceedance probabilities calculation. The second contributing factor impacting the exceedance probabilities is the asymptotic behaviour of the tail of the distribution. That is why we have categorized different exceedance probabilities depending upon the distribution. The last contributing factor consists of approaching where the WCET might be. We have done this by using the HWM that is used to derive the attached exceedance probability. As a consequence, this entire protocol is advocated to estimate confident exceedance probabilities that are expected to be used as training data in the future to derive pWCET having *unknown uncertainty*.

6.4 Future Work

From the work outlined in this document there are two main topics worth looking at.

- **Test Generators**: From the limitations of our method, an important objective is to improve the scalability. More accurately, how to elaborate more efficient graph structures and explorations bearing in mind the often intractability of path coverage. The heuristic may also assume that the SUT contains a wide array of infeasible paths and it may focus on spotting the feasible ones as well. In addition, a more accurate definition of the cost is desirable so as to target more successfully promising paths for the WCET.

  Additionally, the collection of constraints is, in our experience, the most challenging task to apply CBT. Future work could investigate whether the sort of constraints found in real-time software can be generated statically

avoiding the need of instrumentation. A promising alternative to instrumentation is use powerful debuggers like the one employed in [25], which are equipped with a significant tracing functionality. According to our experience, this sort of hardware is often employed in the embedded industry.

Perhaps the most stimulating goal would be to integrate SBT and CBT. In this respect, some existing works have already paved the path integrating symbolic execution and SBT [63, 62]. Baars et al. [63] for instance, have merged the generated knowledge by symbolic execution with fitness functions in order to enhance them. In the light of the results, this enhancement improved the efficiency of the approach and slightly optimized the coverage achieved.

Galeotti et al. [62] have also combined both approaches but their approach switches between SBT and symbolic execution dynamically. Results show how this hybridization had a positive statistical significance impact on the branch coverage. However, new problems emerge when applying these techniques together such as the specifics about how and for how long we should apply each of them.

To our knowledge, in the realm of MBTA this integration has not been that explored. The only known work is delivered by Bünte et al. [25] in which, after a test vector is generated by CBT initially, the SBTG takes over to maximize local HWM to compose later a CWCET.

Speculative speaking, we believe that an optimal test generation targeting both maximization of coverage and efficiency would integrate CBT and SBT by making the most of their strengths. A different question is how this hybridization could make a difference at timing analysis depending on the embedded architecture the SUT runs on.

- **Probabilistic Analysis**: In our estimation, the central challenge if probabilistic approaches aims for being used is firstly, to calculate the level of uncertainty of a particular TG in a predictable way. The controversial point is that, if we achieve this objective to argue whether probabilistic analysis is needed at all. Then, assuming there is a way, link this notion of uncertainty with a exceedance probability. Eventually, if the data passes the required checks of the probabilistic analysis protocol, a distribution can fill this gap and derive more confident results. SBF may help to achieve this objective.

  Additionally, TGs techniques may have the objective of building statistically friendly distributions whenever this can be possible considering the idiosyncrasies of the benchmark and the hardware platform.

Appendix A

```
/** Composition (multiplication) of two rotation matrices.
 * m_a2c = m_a2b comp m_b2c , aka  m_a2c = m_b2c * m_a2b
 */
void float_rmat_comp(struct FloatRMat *m_a2c,
    struct FloatRMat *m_a2b, struct FloatRMat *m_b2c)
{

  m_a2c->m[0] = m_b2c->m[0] * m_a2b->m[0] +
    m_b2c->m[1] * m_a2b->m[3] + m_b2c->m[2] * m_a2b->m[6];

  m_a2c->m[1] = m_b2c->m[0] * m_a2b->m[1] +
     m_b2c->m[1] * m_a2b->m[4] + m_b2c->m[2] * m_a2b->m[7];

  m_a2c->m[2] = m_b2c->m[0] * m_a2b->m[2] +
    m_b2c->m[1] * m_a2b->m[5] + m_b2c->m[2] * m_a2b->m[8];

  m_a2c->m[3] = m_b2c->m[3] * m_a2b->m[0] +
    m_b2c->m[4] * m_a2b->m[3] + m_b2c->m[5] * m_a2b->m[6];

  m_a2c->m[4] = m_b2c->m[3] * m_a2b->m[1] +
     m_b2c->m[4] * m_a2b->m[4] + m_b2c->m[5] * m_a2b->m[7];
```

```
22
23   m_a2c->m[5] = m_b2c->m[3] * m_a2b->m[2] +
24     m_b2c->m[4] * m_a2b->m[5] + m_b2c->m[5] * m_a2b->m[8];
25
26   m_a2c->m[6] = m_b2c->m[6] * m_a2b->m[0] +
27     m_b2c->m[7] * m_a2b->m[3] + m_b2c->m[8] * m_a2b->m[6];
28
29   m_a2c->m[7] = m_b2c->m[6] * m_a2b->m[1] +
30     m_b2c->m[7] * m_a2b->m[4] + m_b2c->m[8] * m_a2b->m[7];
31
32   m_a2c->m[8] = m_b2c->m[6] * m_a2b->m[2] +
33     m_b2c->m[7] * m_a2b->m[5] + m_b2c->m[8] * m_a2b->m[8];
34
35 }
```

Listing 1: Example of a piece of software where the execution is not driven by code coverage metrics. It corresponds to navigation system software listing from an open source code for Unmanned Aerial Vehicles (UAV) [111].

Appendix B

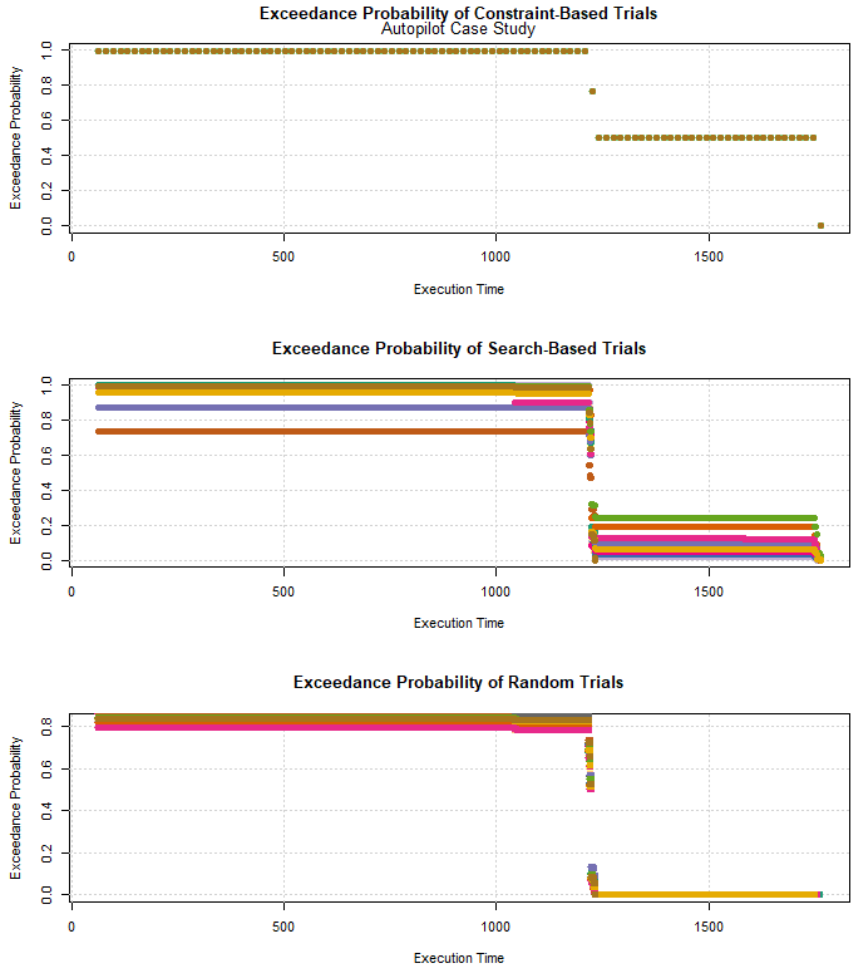


Figure 1: Execution time profile. Each trial corresponds to a different color.

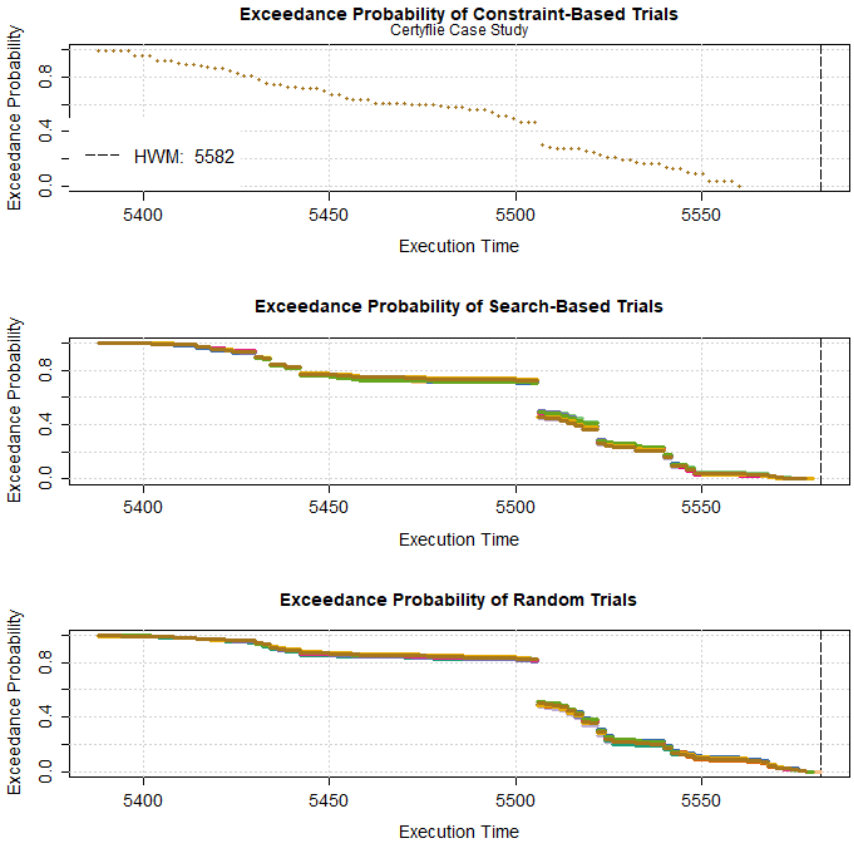


Figure 2: Execution time profile. Each trial corresponds to a different color.

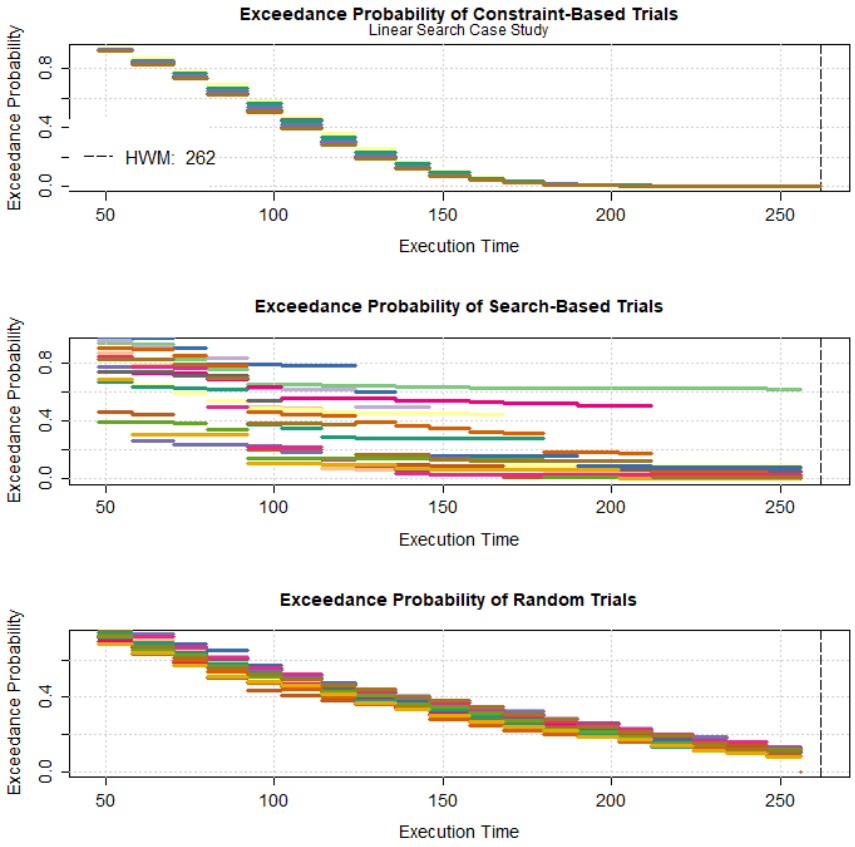


Figure 3: Execution time profile. Each trial corresponds to a different color.

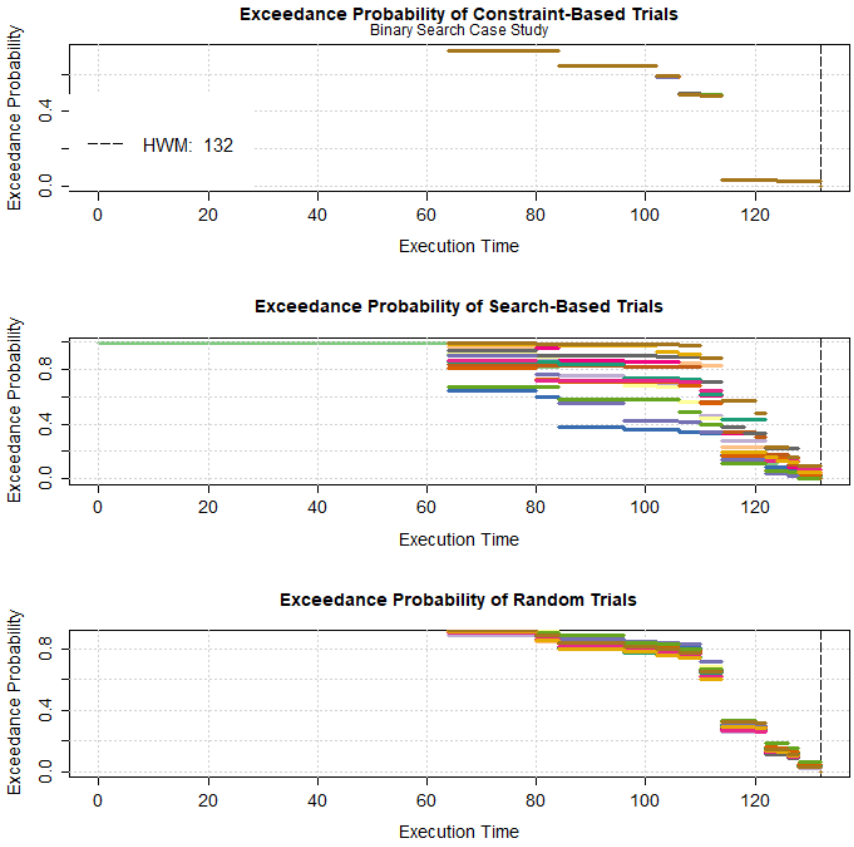


Figure 4: Execution time profile. Each trial corresponds to a different color.

List of Abbreviations

BFS Best-First Search. iii, vii, 27, 36, 41, 42, 54, 57, 63, 65, 66, 69, 74, 75, 79, 80, 82, 88–90, 95, 96, 106, 121, 123, 129, 134, 163, 171, 213–216, 223, 224

BM Block Maxima. 30, 185

CBT Constraint-Based Testing. i, 11–14, 23, 26–29, 35, 36, 39–41, 43, 49, 103, 122, 126–128, 172–174, 212, 213, 216, 227, 228

CBTG Constraint-Based Test Generator. iii, v, viii, ix, xi–xiii, 23, 27, 35–37, 42, 50, 54, 63, 64, 91, 95, 97, 98, 102–106, 109, 110, 112, 113, 115–120, 122–124, 126, 127, 129, 134–139, 141, 142, 144–152, 154, 155, 157, 161–163, 165–169, 171–173, 176, 177, 199, 203, 204, 207, 208, 210, 213–219, 222, 223

CCDF Complementary Cumulative Distribution Function. 184, 187, 194, 204, 208, 209, 211

CFA Control Flow Automaton. 26, 27, 41, 46–49

CFG Control-Flow Graph. 16, 17, 26, 27

CG Constraint-Graph. vii, 46–49, 54, 65, 66, 69, 71, 74–77, 79, 81, 82, 84, 88, 216

CNS Cost Node Structure. 69, 71, 72, 78

CSP Constraint Satisfaction Problem. 11, 12, 26, 40, 41, 43, 44, 46, 47, 50–52, 67, 68

CTN Cost Tree Node. 69–72, 77

CWCET Computed Worst-Case Execution Time. vi, 6–13, 15, 17, 18, 20, 21, 28, 29, 35–37, 41, 127, 173–175, 177, 179, 216, 217, 228

DFS Depth First Search. 27, 36, 41, 47

ETP Execution Time Profile. iv, 33, 179, 182, 183, 199, 204, 207–211, 220, 221

ETT Exponential Tail Test. 32–34, 190, 194, 195, 199

EVT Extreme Value Theory. 29–34, 37, 38, 180–191, 194, 195, 197, 200, 207, 209, 210, 213, 214, 220, 221, 227

GEV Generalised Extreme Value Distribution. 30, 32, 34, 187, 188, 190, 195, 210

GoF Goodness-of-Fit. 33, 34, 180, 182, 187, 189, 195, 197–199, 206, 208, 210, 220

GPD Generalised Pareto Distribution. 30, 32, 187, 191

GPDT Generalized-Pareto Distribution Test. 34, 190, 193, 195

HWM High-Water Mark. vi–xii, 5–11, 18, 21, 22, 24, 29, 33, 36–38, 41, 92, 94, 95, 98, 100–103, 109–111, 113, 115, 116, 122–124, 132, 134–137, 139, 142, 145–149, 151–155, 158–161, 165, 166, 170, 171, 175–180, 196, 197, 199, 204, 206, 207, 215–220, 222–224, 226–228

IPET Instruction Path Enumeration Technique. 28, 41

MBPTA Measurement-Based Probabilistic Timing Analysis. iv, xiv, 10, 11, 13–15, 29, 33, 34, 37, 38, 91, 180–184, 197, 200, 204–208, 210, 211, 213, 214, 219–221, 226

MBTA Measurement-Based Timing Analysis. i, 8, 10–13, 15, 20, 22–25, 28, 29, 35, 36, 41, 52, 64, 94, 96, 100, 172, 173, 180, 181, 212, 213, 215, 218, 228

MDA Maximum Domain of Attraction. 189, 190, 193, 210

PoT Peaks-over-Threshold. 30, 185, 190

pWCET Probabilistic Worts-Case Execution Time. 31, 32, 37, 38, 42, 180, 181, 184, 187, 209, 213, 214, 219, 220, 227

RT Random Testing. 11, 23, 24, 26, 36, 122, 123

RTG Random Test Generator. ix, xiii, xiv, 36, 66, 91, 93, 95, 99, 100, 102–105, 109–112, 115–118, 121, 124, 125, 134–139, 142–146, 148–151, 154, 155, 157, 161, 162, 165–167, 172, 182, 199, 201, 202, 204–206, 208–210, 213, 217–219, 221, 227

SBF Search-Based Fitting. 34, 38, 181, 195, 197, 200, 203, 204, 206, 210, 214, 221, 229

SBT Search-Based Testing. 11, 23, 24, 26, 28, 29, 35, 36, 39, 40, 91, 97, 98, 120, 122, 123, 132, 170, 208, 212, 218, 228

SBTG Search-Based Test Generator. ix, xiv, 23, 25, 36, 64, 66, 91, 93, 95, 97–100, 102–105, 109–112, 115–118, 121, 124, 125, 134–139, 142–145, 148–151, 154, 155, 157, 161, 162, 165–167, 172, 203, 205–208, 210, 213, 217–219, 221, 223, 228

STA Static Timing Analysis. 9, 11, 17–20, 22, 35, 57, 212

SUT Software Under Test. 11, 12, 17, 18, 20, 21, 24–27, 30, 35–37, 44, 46–49, 54, 63, 64, 66–68, 74–76, 94, 100, 108, 126, 129, 133, 144, 149, 172–174, 177, 213, 215, 217, 225, 227, 228

TG Test Generator. 112, 113, 115, 127, 138, 139, 144, 147–149, 155, 157, 162, 172, 177, 178, 180, 181, 184, 196, 209–211, 214, 218–220, 226, 229

WCET Worst-Case Execution Time. i, vi, ix, 3–9, 12–15, 17, 19–23, 32, 33, 35, 41, 58, 98, 126, 132–135, 148, 179–181, 184, 212–214, 222, 227

WNMT Wilcoxon-Nemenyi-McDonald-Thompson Test. vii–xii, 96, 101, 104, 109, 110, 115, 116, 142, 144, 148, 149, 153, 159, 165

References

[1] Franck Wartel, Leonidas Kosmidis, Chieh Lo, Benoit Triquet, Eduardo Quinones, Jaume Abella, Adriana Gogonel, Andrea Baldovin, Enrico Mezzetti, Lucian Cucu, et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *8th IEEE International Symposium on Industrial Embedded Systems*, pages 241–248. IEEE, 2013. vi, 1

[2] Peter H. Feiler Jörgen Hansson, Steve Helton. Roi analysis of the system architecture virtual integration initiative. Technical report, Software Engineering Institute, 2018. vi, 1

[3] Abderrahmane Brahmi, David Delmas, Famantanantsoa Randimbivololona, Abdellatif Atki, Thomas Marie, and Mohamed Habib Essoussi. Formalise to automate: deployment of a safe and cost-efficient process for avionics software formalise to automate: deployment of a safe and cost-efficient process for avionics software. In *9th European Congress on Embedded Real Time Software and Systems (ERTS 2018)*, 2018. vi, 1

[4] Lucas Bang, Abdulbaki Aydin, and Tevfik Bultan. Automatically computing path complexity of programs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 61–72, New York, NY, USA, 2015. ACM. vi, 22, 52, 53

[5] AdaCore, 2020. viii, 108

[6] AdaCore, 2020. viii, 114

[7] Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. Proartis: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(2s):94, 2013. 1, 181, 182

[8] Lisanne Bainbridge. Ironies of automation. *Automatica*, 19(6):775–779, 1983. 2

[9] Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. 2, 16, 21, 43

[10] Roderick Chapman, Alan Burns, and Andy Wellings. Combining static worst-case timing analysis and program proof. *Real-Time Systems*, 11(2):145–171, 1996. 2, 18, 57, 177, 224

[11] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009. 2, 3

[12] Patrick Graydon and Iain Bate. Safety assurance driven problem formulation for mixed-criticality scheduling. In *Proceedings of the Workshop on Mixed-Criticality Systems*, pages 19–24, 2013. 2, 43

[13] Roderick Chapman. *Static Timing Analysis And Program Proof*. PhD thesis, 1995. 3, 4, 9, 18, 19, 133, 171

[14] Paul Lokuciejewski, Daniel Cordes, Heiko Falk, and Peter Marwedel. A fast and precise static loop analysis based on abstract interpretation, program slicing and polytope models. In *2009 International Symposium on Code Generation and Optimization*, pages 136–146, March 2009. 3, 16, 17, 18, 129

[15] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006. 3, 4

[16] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):36, 2008. 4, 5, 6, 12, 15, 20

[17] Mark Bartlett, Iain Bate, and Dimitar Kazakov. Accurate determination of loop iterations for worst-case execution time analysis. *IEEE Transactions on Computers*, 59(11):1520 – 1532, nov 2010. 4, 19, 57

[18] John Barnes. *Spark: The Proven Approach to High Integrity Software*. Altran Praxis, http://www.altran.co.uk, UK, 2012. 4, 16, 48, 97, 122, 224, 225

[19] Nigel James Tracey. *A Search-Based Automated Test-Data Generation Framework for Safety Critical Software*. PhD thesis, 2000. 4, 5, 9, 10, 11, 24, 35, 37, 126, 132, 212

[20] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 333–339. IEEE, 2008. 4, 5, 15, 20, 22, 52

[21] Stephen Law and Iain Bate. Achieving appropriate test coverage to support measurement-based timing analysis. *28th Euromicro Conference on Real-Time Systems (ECRTS), Toulouse*, 2016. 5, 6, 10, 11, 12, 20, 25, 27, 35, 40, 91, 92, 96, 132, 172, 208, 213

[22] Linda Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. Taylor & Francis, 2013. 5, 18, 21, 22, 43, 93, 120, 172, 183, 184, 224

[23] Benjamin Lesage, David Griffin, Frank Soboczenski, Iain Bate, and Robert I. Davis. A framework for the evaluation of measurement-based timing analyses. *23rd International Conference on Real-Time Networks and Systems (RTNS)*, 2015. 5, 10, 20, 32, 33, 37

[24] Patrick Graydon and Iain Bate. Realistic safety cases for the timing of systems. *The Computer Journal*, pages 759–774, 2013. 8, 9, 20, 172

[25] Guillem Bernat, Antoine Colin, and Stefan Petters. pwcet: A tool for probabilistic worst-case execution time analysis of real-time systems. *Report-University of York Department of Computer Science YCS*, 2003. 9

[26] Robert Ian Davis and Liliana Cucu-Grosjean. A survey of probabilistic timing analysis techniques for real-time systems. *LITES: Leibniz Transactions on Embedded Systems*, 6(1):1–60, 5 2019. 10, 13, 30, 32, 33, 34, 37, 180, 182, 186, 209, 213

[27] Jean Diebolt, Myriam Garrido, and Stéphane Girard. *A Goodness-of-fit Test for the Distribution Tail*, pages 95–109. 09 2007. 10, 13, 34, 180, 188, 189, 190, 191, 192, 197, 198, 209, 214

[28] Sven Bünte, Michael Zolda, and Raimund Kirner. Let's get less optimistic in measurement-based timing analysis. *Industrial Embedded Systems (SIES), 2011 6th IEEE International Symposium on*, pages 204–212, 2011. 10, 21, 29, 35, 228

[29] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quinones, and Francisco J Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *2012 24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101. IEEE, 2012. 11, 31, 32, 33, 184, 221

[30] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2009. 11, 12, 13, 23, 26, 27, 46, 47, 54, 55, 91, 132, 223

[31] Gang-Ryung Uh, Robert Cohn, Bharadwaj Yadavalli, Ramesh Peri, and Ravi Ayyagari. Analyzing dynamic binary instrumentation overhead. In *WBIA Workshop at ASPLOS*. Citeseer, 2006. 11, 35

[32] Phil McMinn. Search-based software test data generation: A survey. *Software Testing Verification and Reliability*, 14(2):105–156, 2004. 11, 24, 26, 98, 212

[33] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008. 11, 26, 129, 133, 164, 170, 171, 214

[34] Usman Khan and Iain Bate. Wcet analysis of modern processors using multi-criteria optimisation. In *1st International Symposium on Search Based Software Engineering*, pages 103–112. IEEE, 2009. 11, 25, 223

[35] Michael Zolda, Sven Bünte, and Raimund Kirner. Towards adaptable control flow segmentation for measurement-based execution time analysis. In *Proceedings of the International Conference on Real-Time and Network Systems*, pages 35–44, 2009. 11, 12, 20, 21, 28, 29, 41, 172

[36] Sebastian Altmeyer, Björn Lisper, Claire Maiza, Jan Reineke, and Christine Rochange. Wcet and mixed-criticality: What does confidence in wcet estimations depend upon? In *OASIcs-OpenAccess Series in Informatics*, volume 47. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015. 11

[37] Jordy Ruiz and Hugues Cassé. Using SMT Solving for the Lookup of Infeasible Paths in Binary Programs. In Francisco J. Cazorla, editor, *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, volume 47 of *OpenAccess Series in Informatics (OASIcs)*, pages 95–104, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 11, 18, 173

[38] Rapita Systems Ltd. Rapitime explained. `https://www.rapitasystems.com/downloads/brochures-white-papers/rapitime-explained`. 11, 12, 20, 25, 35, 172, 174, 177, 216, 217

[39] Arnaud Gotlieb. Constraint-based testing. an emerging trend in software testing. *Advances in Computers*, 99:67–101, 12 2015. 12, 33, 182

[40] Patrice Godefroid. Higher-order test generation. *SIGPLAN Not.*, 46(6):258–269, June 2011. 12, 26

[41] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):8290, February 2013. 12, 27, 41, 44, 214

[42] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 58–70, New York, NY, USA, 2002. ACM. 13, 26, 27, 35, 46, 47

[43] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. Fshell: Systematic test case generation for dynamic analysis and measurement. In *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pages 209–213, 2008. 13, 27, 35, 41, 173, 213, 215

[44] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer Publishing Company, Incorporated, 2010. 16

[45] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Department of Computer Engineering, Mälardalen University, Box 883, S-721 23 Västerås, Sweden, and Department of Computer Systems, Information Technology, Uppsala University, Box 325, S-751 05 Uppsala, Sweden, May 2000. 16, 18, 129

[46] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), 2018. 17

[47] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010. 17, 22, 222

[48] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2):159–176, 1989. 18

[49] ait. ait: Static timing analysis tool. 18

[50] Gilles Brassard and Paul Bratley. *Fundamentals of Algorithmics*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996. 19, 27, 54, 65

[51] Guiem Bernat, Antoine Colin, and Stefan M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, pages 279–288, Austin, Texas, USA, Dec 3–5 2002. 19

[52] Adam Betts and Guiem Bernat. Tree-based wcet analysis on instrumentation point graphs. In *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 8 pp.–, April 2006. 20, 172

[53] Adam Betts Stefan M. Petters and Guillem Bernat. A new timing schema for wcet analysis. 2004. 20, 173

[54] Marco Ziccardi, Jaume Abella Enrico Mezzetti, Tullio Vardanega, and Francisco J. Cazorla. Epc: Extended path coverage for measurement-based probabilistic timing analysis. In *Proceedings of the Real-Time Systems Symposium (RTSS 2015)*, 2015. 20

[55] Adam Nellis, Pascal Kesseli, Philippa Ryan Conmy, Daniel Kroening, Peter Schrammel, and Michael Tautschnig. Assisted coverage closure. *CoRR*, abs/1509.04587, 2015. 21

[56] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997. 22, 93, 223

[57] Joachim Wegener, Harmen Sthamer, Bryan F Jones, and David E Eyres. Testing real-time systems using genetic algorithms. *Software Quality Journal*, 6(2):127–135, 1997. 24

[58] Joachim Wegener, Roman Pitschinetz, and Harmen Sthamer. Automated testing of real-time tasks. In *Proceedings of the First International Workshop on Automated Program Analysis, Testing and Verification, Limerick, Ireland*, 2000. 24

[59] Hans-Gerhard Groß. A prediction system for evolutionary testability applied to dynamic execution time analysis. *Information and Software Technology*, 43(14):855–862, 2001. 24

[60] Mark Harman, Lin Hu, Rob Hierons, Joachim Wegener, Harmen Sthamer, Andre Baresel, and Marc Roper. Testability transformation. *IEEE Transactions on Software Engineering*, 30(1):3–16, 2004. 24, 40, 44, 120, 170, 212

[61] Mark Harman. Open problems in testability transformation. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 196–209, 2008. 25, 44

[62] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. 26

[63] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Fitness-guided path exploration in dynamic symbolic execution. In *2009 IEEE/IFIP International Conference on Dependable Systems Networks*, pages 359–368, 2009. 27, 129

[64] Juan Pablo Galeotti, Gordon Fraser, and Andrea Arcuri. Improving search-based test suite generation with dynamic symbolic execution. In *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, pages 360–369, 2013. 27, 28, 35, 228

[65] Arthur Baars, Mark Harman, Youssef Hassoun, Kiran Lakhotia, Phil McMinn, Paolo Tonella, and Tanja Vos. Symbolic search-based testing. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, page 5362, USA, 2011. IEEE Computer Society. 27, 28, 35, 40, 63, 120, 228

[66] Michael Zolda. *Precise measurement-based worst-case execution time estimation.* PhD thesis, 2012. 28

[67] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, 16(12):1477–1487, Dec 1997. 28

[68] Andreas Holzer, Christian Schallhart, Michael Tautschnig, and Helmut Veith. How did you specify your test suite. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 407–416, New York, NY, USA, 2010. ACM. 28

[69] Sven Bünte, Michael Zolda, Michael Tautschnig, and Raimund Kirner. Improving the confidence in measurement-based timing analysis. *Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), 2011 14th IEEE International Symposium on*, pages 144–151, 2011. 28, 29, 41

[70] George Lima and Iain Bate. Valid application of evt in timing analysis by randomising execution time measurements. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium IEEE Real-Time and Embedded Technology and Applications Symposium*, 2017. 29, 33, 182, 209

[71] Emil Julius Gumbel. *Statistics of extremes*. Courier Corporation, 2012. 29, 31, 186

[72] Stuart Coles. *An introduction to statistical modeling of extreme values*. Springer series in statistics. Springer, 2001. Autres tirages : 2004, 2007, 2009. 29, 31, 185, 209

[73] Paul Embrechts, Claudia Kluppelberg, and Thomas Mikosch. *Modelling extremal events for insurance and finance*. Applications of mathematics. Springer, 1997. 29, 186, 187, 188, 190

[74] Samuel Jiménez-Gil, Iain Bate, George Lima Luca Santinelli, Adriana Gogonel, and Liliana Cucu-Grosjean. Open challenges for probabilistic measurement-based worst-case execution time. *Embedded Systems Letters*, 2017. 30, 33, 34, 179, 186, 194, 213

[75] Carl J. Scarrott and Anna MacDonald. A review of extreme value threshold estimation and uncertainty quantification. *Revstat Statistical Journal*, 10:33–60, 03 2012. 30, 186, 188

[76] Cláudia Neves and M. Isabel Fraga Alves. Testing extreme value conditions — an overview and recent approaches. In *REVSTAT*. 30

[77] Petra Friederichs and Thordis L Thorarinsdottir. Forecast verification for extreme value distributions with an application to probabilistic peak wind prediction. *Environmetrics*, 23(7):579–594, 2012. 30, 34, 37, 180, 195, 196

[78] Stuart Edgar and Alan Burns. Statistical analysis of wcet for scheduling. In *Proceedings IEEE RTSS 2001*, 2001. 31

[79] Guiem Bernat, Martin J. Newby, and Alan Burns. Probabilistic timing analysis: an approach using copulas. *Journal of Embedded Computing*, 1(2):179–194, 2005. 31, 174, 179

[80] Alessandra Melani, Eric Noulard, and Luca Santinelli. Learning from probabilities: Dependences within real-time systems. In *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–8. IEEE, 2013. 31

[81] Jeffery Hansen, Scott A Hissam, and Gabriel A Moreno. Statistical-based wcet estimation and validation. In *Proceedings of the 9th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009. 31

[82] David Griffin and Alan Burns. Realism in statistical analysis of worst case execution times. In *OASIcs-OpenAccess Series in Informatics*, volume 15. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2010. 31, 33, 210

[83] Leonidas Kosmidis, Charlie Curtsinger, Eduardo Quiñones, Jaume Abella, Emery Berger, and Francisco J Cazorla. Probabilistic timing analysis on conventional cache designs. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 603–606. EDA Consortium, 2013. 32

[84] Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J Cazorla. A cache design for probabilistically analysable real-time systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 513–518. EDA Consortium, 2013. 32

[85] Cazorla Francisco J. Padilla María Abella Jaume, del Castillo Juan. Extreme value theory in computer sciences: The case of embedded safety-critical systems. In *6th International Conference on Risk Analysis (ICRA)*. ICRA, 2015. 32

[86] Edna Barros George Lima, Dàrio Dias. Extreme value theory for estimating task execution time bounds: A careful look. In *ECRTS 2016*. IEEE, 2016. 32, 37, 179, 181, 183, 207

[87] Joan Del Castillo, Jalila Daoudi, and Richard Lockhart. Methods to distinguish between polynomial and exponential tails. *Scandinavian Journal of Statistics*, 41(2):382–393, 2014. 32, 187, 188, 210

[88] Irina Fedotova, Bernd Krause, and Eduard Siemens. Applicability of extreme value theory to the execution time prediction of programs on socs. 2017. 32, 187, 210, 221

[89] George Lima. Facts and myths around applying evt in timing analysis, June 2017. 33, 185, 210

[90] Jean Diebolt, Myriam Garrido, and Stéphane Girard. Le test ET : test d'adéquation d'un modèle central à une queue de distribution. Research Report RR-4170, INRIA, 2001. 33, 34, 180, 188, 189, 190, 192, 197, 198

[91] Alberto Luceno. Maximum likelihood vs. maximum goodness of fit estimation of the three-parameter weibull distribution. *Journal of Statistical Computation and Simulation*, 78(10):941–949, 2008. 34, 180, 195

[92] Stephen J. Maher, Tobias Fischer, Tristan Gally, Gerald Gamrath, Ambros Gleixner, Robert Lion Gottwald, Gregor Hendel, Thorsten Koch, Marco E. Lübbecke, Matthias Miltenberger, Benjamin Müller, Marc E. Pfetsch, Christian Puchert, Daniel Rehfeldt, Sebastian Schenker, Robert

Schwarz, Felipe Serrano, Yuji Shinano, Dieter Weninger, Jonas T. Witt, and Jakob Witzig. The scip optimization suite 4.0. Technical Report 17-12, ZIB, Takustr.7, 14195 Berlin, 2017. 50, 65, 96, 132, 164, 225

[93] Alberto Prieto Julio Ortega, Mancia Anguita. *Arquitectura de Computadores*. Thomson, 2005. 58

[94] STMicroelectronics. *RM0090 - Reference manual*, 2019. 58

[95] AdaCore, Ada Drivers Library. 59

[96] Richard PG Collinson. *Introduction to avionics systems*. Springer Science & Business Media, 2013. 63, 101, 108

[97] Patrice Godefroid. Test Generation Using Symbolic Execution. In Deepak D'Souza, Telikepalli Kavitha, and Jaikumar Radhakrishnan, editors, *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2012)*, volume 18 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24–33, Dagstuhl, Germany, 2012. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 64, 129, 133, 164, 175

[98] Antonio G. Carrillo and Joaquín. Fernández-Valdivia. *Abstracción y estructuras de datos en C++*. Delta, 2006. 77, 79, 133, 141

[99] Microsoft. Working set reference. 95

[100] Microsoft. Process memory counters windows api. 95

[101] AdaCore, Ada Drivers Library. 97, 108

[102] AdaCore, Ada Drivers Library. 97, 115

[103] Claire Maiza, Hamza Rihani, Juan M. Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I. Davis. A survey of timing verification techniques for multi-core real-time systems. *ACM Comput. Surv.*, 52(3), June 2019. 122, 222

[104] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The mälardalen wcet benchmarks: Past, present and future. In Björn Lisper, editor, *WCET*, volume 15 of *OASICS*, pages 136–146. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany, 2010. 126, 128

[105] Rober I. Davis, Tullio Vardanega, Jan Andersson, Francis Vatrinet, Mark Pearce, Ian Broster, Mikel Azkarate-Askasua, Frank Wartel, Liliana Cucu-Grosjean, Glenn Farrall Mathieu Patte, and Francisco J. Cazorla. Proxima: A probabilistic approach to the timing behaviour of mixed-criticality systems. *Ada User Journal*, (2):118–122, 2014. 181

[106] Stuart Coles, Joanna Bawa, Lesley Trenner, and Pat Dorazio. *An introduction to statistical modeling of extreme values*, volume 208. Springer, 2001. 186

[107] Myriam Garrido and Jean Diebolt. The et test, a goodness-of-fit test for the distribution tail. In *Methodology, Practice and Inference, second international conference on mathematical methods in reliability*, pages 427–430, 2000. 189, 190

[108] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2017. 197

[109] Yang Xiang, Sylvain Gubian, Brian Suomela, and Julia Hoeng. Generalized simulated annealing for efficient global optimization: the GenSA package for R. *The R Journal Volume 5/1, June 2013*, 2013. 197

[110] Jean Diebolt, Myriam Garrido, and Stéphane Girard. Asymptotic normality of the et method for extreme quantile estimation. application to the et test. 2002. 198

[111] Paparazzi. Paparazzi open source software package for unmanned (air) vehicle systems, 2018. 231