IMPROVING TIMING ANALYZABILITY THROUGH SOURCE CODE ANALYSIS AND TESTABILITY TRANSFORMATIONS IN REAL-TIME SYSTEMS

David Barrientos Velasco
IMPROVING TIMING ANALYZABILITY THROUGH SOURCE CODE ANALYSIS AND TESTABILITY TRANSFORMATIONS IN REAL-TIME SYSTEMS

Thesis

submitted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

Software Engineering Group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

David Barrientos Velasco

30th June 2006
Author
David Barrientos Velasco

Title
IMPROVING TIMING ANALYZABILITY THROUGH SOURCE CODE ANALYSIS AND TESTABILITY TRANSFORMATIONS IN REAL-TIME SYSTEMS

MSc presentation
July 11th, 2006

Thesis Committee
prof. dr. ir. Arie van Deursen (chair) Delft University of Technology
Dr. Phil. Hans-Gerhard Gross Delft University of Technology
Dr. Kees van der Meer Delft University of Technology
Abstract

Timing analysis is a method for validating the temporal correctness of real-time systems. It gives the confidence of the timing behavior of a real-time system, by measuring the bounds of the execution time of all the tasks in a system. However, timing analysis is a very difficult to perform technique, and it encounters many problems at different levels of abstraction, from high-level code to machine code.

Timing analysis is in fact testing, where the test target is the main difference between them. In testing we want to generate enough test cases to find bugs in a system. While, in timing analysis we want to find input data that will generate the longest or shortest execution time of the program.

In this thesis, I take advantage of the similarities between testing and timing analysis, and introduce the concepts of testability in the timing analysis domain, so that, we can generate more information (e.g. number of simple paths) based on the source code that will help us to predict the effort that we will require to perform timing analysis in a real-time system. Therefore, we can identify different patterns (i.e. testability transformation) at the source code level that can help us to improve the timing analyzability of real-time systems that make use of these patterns.
## Contents

1 Introduction ............................... 1
   1.1 Outline of this thesis ................. 5

2 Concepts in Real-Time Systems and Scope of this Thesis 6
   2.1 Real-Time Systems ..................... 6
   2.2 Execution Time ......................... 7
   2.3 Timing Analysis ......................... 8
       2.3.1 Static Timing Analysis ............ 10
       2.3.2 Dynamic Timing Analysis .......... 11
   2.4 Predictability .......................... 11
   2.5 Analyzability ........................... 12
   2.6 Domains in timing analysis .......... 14
   2.7 Obstacles in timing analysis .......... 15
   2.8 Overview on testing and testability  . 16
   2.9 Objective of this thesis ............... 17
   2.10 Summary ............................... 18

3 Problems and Threats in Timing Analysis ................................ 20
   3.1 Problem Domains in Timing Analysis .... 20
       3.1.1 Characterization of executions paths 22
List of Figures

2.1 Representations of the relations between measures related to execution time [37] .............................................. 8
2.2 Relationship between the improvement of analyzability and time predictability in real-time systems .............................. 14
3.1 Three domains in timing analysis [28] .................................... 22
3.2 C-like syntax extensions for bounded loops [30] ........................ 24
4.1 Some prime flowgraphs. a) if-then b) if-then-else c) while-do d) switch-case e), f), g) sequences. Every black dot represents a statement in the source code .............................................. 34
4.2 Composed flowgraphs by applying sequencing and nested operations. a) sequence of two if-then-else statements b) a while-do statement nested on a if-else-statement ........................... 35
5.1 Order of execution of the operations of a) expression 5.2 and b) expression 5.1 [29] ................................................................. 51
5.2 Results of execution time of sorting 500 elements using Algorithm 5.6 ................................................................. 59
5.3 Results of execution time of sorting 500 elements using Algorithm 5.7 ................................................................. 60
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.4</td>
<td>Measurements of the execution times of Algorithm 5.9</td>
<td>66</td>
</tr>
<tr>
<td>5.5</td>
<td>Measurements of the execution times of Algorithm 5.10</td>
<td>67</td>
</tr>
<tr>
<td>5.6</td>
<td>Measurements of the execution times of the original implementations of a binary search algorithm from Algorithm 5.1</td>
<td>69</td>
</tr>
<tr>
<td>5.7</td>
<td>Measurements of the execution times of the original implementations of a binary search algorithm from Algorithm 5.11</td>
<td>70</td>
</tr>
<tr>
<td>5.8</td>
<td>Measurements of the execution time, in nanosecond, of Algorithm 5.15</td>
<td>76</td>
</tr>
<tr>
<td>5.9</td>
<td>Measurements of the execution time, in nanosecond, of Algorithm 5.16</td>
<td>77</td>
</tr>
</tbody>
</table>
List of Tables

4.1 Metric values of prime flowgraphs for different testing strategies . 40
4.2 Metric functions for sequencing operations . . . . . . . . . . . . 41
4.3 Metric functions for nesting operations . . . . . . . . . . . . . . 41

5.1 Execution times (in CPU cycles) of traditional and transformed
binary search algorithms [26] . . . . . . . . . . . . . . . . . . . . . . . 52
5.2 Formulas for the different conditions that can be present in an if-
statement. If the condition is false C will keep its old value . . . . 62
5.3 Formulas for the logical ‘and’ (&&) operator and the ‘or’ (||) ope-
rator . The ‘|’ represents the bitwise inclusive ‘or’ function. . . . . 63
5.4 Summary of the results of all the experiments carried out in the
previous section . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
List of Algorithms

4.1 Biggest value out of three numbers .......................... 43
4.2 Second implementation biggest value out of three numbers ... 44
4.3 Binary Search algorithm for sorted lists ....................... 45
4.4 Sequential Search algorithm for sorted lists .................. 46
4.5 Sequential Search algorithm for any list of numbers ......... 47
5.1 Traditional implementation of a binary Search algorithm ..... 53
5.2 Transformed implementation of Algorithm 5.1 using constant-time conditional expression pattern ....................... 53
5.3 Pattern to calculate the maximum value of two numbers .... 55
5.4 $maxValue$: Testability transformation for Algorithm 5.3 ...... 55
5.5 $minValue$: Testability transformation for minimum value of two numbers .................................................... 55
5.6 Traditional implementation of a bubble sort algorithm ....... 57
5.7 Implementation of a bubble sort algorithm applying min/max-value pattern ................................................... 57
5.8 Source code of the preprocessing phase for the single-path-if pattern ......................................................... 61
5.9 Traditional implementation of a sequential search algorithm ... 64
5.10 Transformed implementation of Algorithm 5.9 applying single-path-if pattern .............................................. 66
5.11 Binary search implementation using single-path-if pattern ... 68
5.12 If-statement example that is very hard to transform due the presence of a inner loop .............................................. 72
5.13 If-statement example that is impossible to transform due the presence of function calls ................................. 72
5.14 Transformation of Algorithm 5.12 using the single-path-if pattern ................................................................. 73
5.15 Traditional implementations of an insertion sort ......................................................................................... 74
5.16 Insertion sort algorithm that is free from input-data dependent control flow ................................................. 74
6.1 A piece of code using an if-statement ........................................ 81
6.2 Transformation of Algorithm 6.1 using the single-path-if pattern showing the worsening of its readability ............................... 82
Chapter 1

Introduction

Real-time systems have been growing quickly in importance and spreading in other areas of our daily activities, such as: houses, offices, etc. [20]. Nowadays, we do not only find these kinds of systems in the industry or research environments (e.g. aircrafts, power plants), but also they are becoming ubiquitous (e.g. microwaves, washing machines). As any other software system, a real-time system must fulfill all the functional requirements that were originally specified and provide a safe and correct operation. In addition, there are time-requirements that a real-time system must comply to. These time-requirements, which are known as deadlines, are as important as any other functional requirement [7].

In general, a real-time system executes a set of tasks, which are considered successfully executed if they provide the results that we expect, and if they finish their operation before a defined deadline. Therefore, to assure the temporal correctness of every task we must gather information about the execution time of each task, this is done through timing analysis. “Timing analysis is a method for validating the temporal correctness of real-time programs” [33]. There are two main methods to obtain the execution time of a task; these are static and dynamic timing analysis [19].
Chapter 1. Introduction

Static timing analysis calculates the execution time by analyzing the source code of a real-time program and evaluating additional information (e.g. possible execution paths, maximum number of iteration of loops, execution time of each instruction in the target machine); without executing it [19]. On the other hand, dynamic timing analysis measures the execution time of a task by running it on the target machine several times and with different data inputs. In this particular situation, dynamic timing analysis can be viewed as a special case of testing where the goal now is to find a test case that can traverse a path in the source code that yields the worst execution time. There are different strategies to generate test cases for dynamic timing analysis (e.g. manually, randomly, evolutionary testing) [12, 19].

We can gather more reliable knowledge about the execution time of a task by applying both static and dynamic analyses. It is possible to implement these two techniques to form a confidence interval, where the lower bounds are the results of dynamic timing analysis and upper bounds are the results of static timing analysis [1].

Deciding which of both timing analysis techniques is better or more efficient is not the main concern when we talk about timing analysis. The real problem is the difficulty and the effort needed to perform both techniques. For example, in the case of static timing analysis, besides the source code, we require extra information (e.g. path information, code annotations) to perform a good analysis. "This demands extensive user interaction and makes static analysis difficult, expensive and error-prone" [31]. In the case of dynamic timing analysis, we usually have to deal with an infinite set of paths in the source code, then we have to try to measure only a subset that contains the longest path, which certainly is not an easy task because we may never know if that subset has been found or not. Thus, we have to work with a very large input space of test cases from which maybe
only one test case can produce the worst execution time [1]. Thus, this process becomes long and hard, like looking for a "needle in the haystack". The same behavior occurs in any testing technique [4], where finding all the bugs in a program may be impossible, and moreover, we will never know if a program is fault-free.

Timing analyzability is an attribute of a real-time system that can help us to predict how the different components of the system contribute to ease the performance of a timing analysis in the system. In both timing analysis techniques, we can apply different methodologies that help us to make the systems better analyzable. For example, when we introduce extra information in static timing analysis, we are making the system, under scrutiny, easier to analyze; thus improving the system’s analyzability in order to make timing analysis easier.

In the same way as for static timing analysis, we can establish methodologies for dynamic timing analysis that can help us to improve a system’s timing analyzability. We can take advantage of the similarities between testing and dynamic timing analysis, so that we can introduce testing concepts, such as testability, as new analyzability methods to gather more information about a system. Testability is defined as the effort (e.g. number of test cases) needed to test a software program [2]. Therefore, we can state that the more effort we need to test a system the less analyzable in terms of timing analysis the system is expected to become. Furthermore, we can look for source code transformations, called testability transformations [3, 17], that can improve the performance of testing (i.e. reducing the number of test cases) and, therefore, increase the source code’s analyzability.

There are misunderstandings when developing real-time systems that make the timing analysis of programs more difficult. In [27], Puschnar attributes the difficulties presented in timing analysis to “the fact that designers of hard real-time systems use inadequate hardware and software architectures”. For example, developers tend to use software algorithms that have yielded good average perfor-
mance and speed for typical applications, where the variation of execution time usually is very large. While in a real-time system, those kinds of algorithms are not very suitable, instead, we are interested to implement algorithms that approximate to a constant execution time for every one of its test cases, which is the ideal scenario that real-time systems developers must aim to, because a system would be totally predictable, which means that we would know its state at any point in time, and we would have a system that will fulfill all its timing requirements (i.e. no missing deadlines).

In this thesis, I propose ways to increase the timing analyzability of real-time systems and ways to approach the system to the ideal scenario of a constant execution time; by means of identifying different testability transformations (i.e. patterns), which can help us to decrease the effort of timing analysis (e.g. using patterns with high testability). Later on, these patterns can be implemented in real-time systems assuring a system that is more analyzable and predictable in terms of time. To achieve this, my thesis will:

- Define a relationship between testability and (dynamic) timing analysis, from where we can extract information that can help us to estimate better the effort of timing analysis in real-time systems.
- Investigate metrics that, first, will help us to measure the testability of different pieces of code.
- Identify patterns, using testability metrics, in the source code with high testability that will decrease the effort of timing analysis in real-time systems.
- Study the timing behavior of those patterns implemented in different algorithms in order to verify the improvement of the algorithm’s analyzability and reduction of the costs in their timing analysis by decreasing the number of paths in the algorithm’s source code.
Chapter 1. Introduction

1.1 Outline of this thesis

The following chapter (chapter 2), first introduces some concepts in real-time systems that are important for the following chapters (sections 2.1 - 2.5). Then, it sets the context of the thesis by discussing the domains on which timing analysis can be studied and how they affect it (section 2.6). The general obstacles that we can encounter in timing analysis are given (section 2.7). An overview of testing and testability analysis is described (section 2.8). This chapter concludes with a description of the objective of this thesis and a summary of the chapter. Chapter 3 presents a deeper analysis of the different threats and problems that we can meet across when performing a timing analysis. Chapter 4 discusses the theory of testing and testability analysis and how they are related to timing analysis (section 4.1). Testability metrics are introduced and the techniques for their measurement are discussed (sections 4.2 and 4.3). This chapter ends with a study of testability transformations and some examples on how to apply them are given (section 4.4). Chapter 5 presents the main topic of this thesis, introducing and discussing all the identified patterns at the source code level to increase timing analyzability and the results of the performed experiments with these patterns. Finally, this thesis ends with a summary, conclusion of all the results that have been obtained and some considerations about further work on this topic.
Chapter 2

Concepts in Real-Time Systems and Scope of this Thesis

In this chapter, I will present some concepts of several terms that exist in a real-time development process. Then I will give an introduction of timing analysis, its problems and how it is related to testing. These definitions are important because they will help us to set up the context in this thesis and define clearly the objective of this thesis.

2.1 Real-Time Systems

According to [18], real time is “pertaining to a system or mode of operation in which computation is performed during the actual time that an external process occurs, in order that the computation results can be used to control, monitor, or respond in a timely manner to the external process”. Thus, [20] presents a definition of a real-time system:

- “Any system where a timely response by the computer to external stimuli is vital is a real-time system”
In other words, we can say that a real-time system is a system whose operations depend upon real time; a computer system with time requirements (deadlines).

### 2.2 Execution Time

A real-time system has time requirements, in other words it runs tasks that must meet their deadlines. Therefore, it is important to know the meaning of execution time of a task and how to measure it. In [21], execution time is defined as “the time interval between the start time and the termination time of a task”. That is to say, execution time is the elapsed time from the time instant when a task is activated until the time instant when the same task finishes its operation. In simple words, the execution time of a task is the time it takes a processor to execute that task, including wait times for external devices (e.g. memory, disks).

Determining the computation time of a task is crucial to a successful schedule of it in a real-time system. An overly pessimistic estimate of the execution time would result in wasted CPU cycles, whereas an under-approximation would result in missed deadlines, which of course would be the worst case scenario [7].

For real-time scheduling purposes, there are two other concepts that are related to execution time, these are the best case execution time (BCET) and the worst case execution time (WCET), although the latter one is more important that the former one, as we will see later.

The BCET of a task is its shortest possible execution time under all admissible system and environments conditions, and execution paths. This time, in general, is not important for real-time systems, because sometimes it is used for synchronization purposes and also to evaluate the results of the worst-case execution time [19], where big differences between BCET and WCET mean poor analyzability and predictability in terms of time in a real-time system.
Chapter 2. Concepts in Real-Time Systems and Scope of this Thesis

Figure 2.1: Representations of the relations between measures related to execution time [37]

The WCET of a task is the largest possible execution time under all admissible system and environments states, and execution paths. This time is required for safety real-time scheduling. At the source code level, the WCET is determined by finding the path that yields the longest possible execution time. In this sense, if we have many paths in the source code, we will require more time and effort to find the WCET.

In order to guarantee that the execution of a task meets its timing requirements (i.e. deadlines), the BCET and WCET must have bounded values. “Upper and lower bounds are quantities that bound the worst case and best case behavior” [37]. Usually these values are calculated statically and not during run-time of the system, by analysis, simulation or statistically.

In Figure 2.1, we can see the representation of the relations between BCET, WCET, upper and lower bounds, and the execution time of a task. Because of the importance of execution time in real-time systems, we require timing analysis techniques that can help us to measure these execution times.

2.3 Timing Analysis

“Timing analysis is a method for validating the temporal correctness of real-time programs” [33]. Thus, it gives the confidence of the timing behavior of a real-
time system, by computing the (upper) bounds for the execution time of the tasks in the program. “These bounds are needed for allocating the correct CPU time to the tasks of an application” [28].

In the development process of a real-time system, after timing requirements are defined and a timing analysis has been performed on all tasks, the next step is to assign all the tasks to (hardware) resources, so that each task satisfies its timing requirements; this activity is known as scheduling. In [20], the scheduling is defined as follows. “Given a set of tasks, task precedence constrains, resource requirements, task characteristics, and deadlines, we are required to devise a feasible allocation/schedule on a given computer”. There are many scheduling algorithms in the literature [20] that can automate the allocation of the tasks. The study of these algorithms is beyond the scope of this thesis.

Because of the importance to provide reliable information about the execution time of a program, timing analysis must fulfill two requirements [28]:

1. **Provision of safe bounds.** The computed execution time bounds must try not underestimate the worst case; the bounds must be safe.

2. **Tightness of computed WCET bounds.** A poorly performed timing analysis would lead to a waste of resource allocation, and, therefore, to a more expensive system. That is why, tight bounds are necessary to avoid these problems.

There are two main methods for timing analysis, these are: static timing analysis and dynamic timing analysis. We will have a close look at these techniques in the next subsections.
2.3.1 Static Timing Analysis

Static timing analysis estimates the WCET of a program without actually executing it. This is possible because, static timing analysis tries to consider “the effects of all possible inputs, including possible system states, together with the program’s interaction with the hardware” [34]. The analysis is based on models of the software and hardware involved. If these models are accurate enough, the result can be a safe timing estimate that is greater than or equal to the actual WCET.

Static timing analysis is usually divided in three phases [34]:

1. Flow analysis of the code. In this phase, which is machine-independent, the dynamic behavior of the program is extracted from the source code. Here we gather information such as: which functions are called, dependencies between if-statements, the number of times a loop can iterate, etc. Usually, this information is obtained through annotations provided by the user [30, 33], or integrated into the programming language [19].

2. Low-level analysis. In this phase, the timing behavior of a program is determined at the machine-code level, taking into consideration different architectural features that the target machine may present, such as: pipelines, caches, branch prediction etc.

3. Calculation phase. Based on all the information gathered in the two previous phases, the calculation phase proceeds to estimate the WCET of a program. A good technique used in this phase, is to transform the WCET problem in a maximization problem, and use Integer Linear Programming to find the maximum value of the execution time [32].
2.3.2 Dynamic Timing Analysis

Dynamic timing analysis measures the WCET of a program by running it several times with different input data, preferably on the target machine where the real-time system will be later implemented. Thus, the WCET will be the longest measured execution time.

Dynamic timing analysis can be viewed as a special case of testing, where the goal is to find an input data set that yields the worst/best case execution time. In this way, dynamic timing analysis can use techniques for data generation that are also used in testing, like: random testing and evolutionary testing.

- Random testing. Random testing uses, either manual or automated, random test data. It is the easiest method to generate test case that does not need a deep knowledge of the program’s structure [4].

- Evolutionary testing. It is based on the application of genetic algorithms to the search of an input data that can produce the worst execution time. A genetic algorithm performs on a population of binary strings (chromosomes), representing input data. These chromosomes are selected, combined and mutated to create “offspring” for subsequent generations. This process is executed continuously for many generations until a stopping criterion is satisfied (e.g. a number of generations is reached) [12].

2.4 Predictability

A definition of predictability is the following [39]:

- “Predictability refers to the degree that a correct prediction of a system’s state can be made either qualitatively or quantitatively”.
In real-time systems, timing predictability means that it should be possible to demonstrate that all tasks of a system met their timing-requirements [15]. However, offering a one hundred percent guarantee for all tasks is not necessary, sometimes even not possible, that is why implicitly, in this guarantee, we can stay that no failures occur, various execution times and deadlines are correct, and at any point in time the system can identify which tasks will miss their deadlines [36].

Accurate timing predictability depends on the measurement of the execution time of every task. Therefore, we can state that time predictability is referred to the difference between upper and lower bounds in execution time. In other words, the larger the difference between the best-case execution time and the worst-case execution time is, variation of the execution time (Figure 2.1), the lower the predictability of the system tends to be [37].

We will see in next section, how a high predictable real-time system is the result of its good timing analyzability.

### 2.5 Analyzability

In real-time systems, it is possible to have a case where the complexity of a system, either at a software level or hardware level, does not contribute to determine easily the lower and upper bounds of the execution time. Even worse, the complexity could be so high that there are no analysis methods available to predict the lower and upper bounds, thus, making timing analysis unfeasible [37].

Furthermore, we can also have limited knowledge about the dependency between the execution time of a task and some non-observed external behavior, which can be seen as interference. A typical example of this problem is the high dependency that control flow of different algorithms has with the input data.
Chapter 2. Concepts in Real-Time Systems and Scope of this Thesis

These two aspects, lack of analysis methods and limited knowledge, contribute to have a less analyzable system. Limited analyzability in a system is a dangerous drawback that can lead to worse bounds in the execution time of a program. Therefore if a task in a real-time system presents large differences between its best-case execution time and the worst-case execution time, then it is a symptom that the system has a poor timing analyzability. As a consequence, this large variance of execution time can produce some missed deadlines and, as said in section 2.4, decrease the time predictability of the system under analysis.

We can improve the timing analyzability, and as a consequence improve the predictability in a real-time system by [37]:

1. Reducing the interference from non-available information, for example by avoiding input-data dependent control decision in the source code.

2. Matching implementation concept with analysis techniques, in the case of this thesis I am studying different concepts of testability analysis to improve the analyzability of real-time systems.

In Figure 2.2, we can see the relationship that exists between the improvement of analyzability and time predictability in real-time systems. If we reduce the interference from non-available information and find new analysis technique, then we can improve the analyzability of a program. Improving timing analyzability means that we can decrease the variation of execution time (gap between BCET and WCET) and come close to the ideal scenario in real-time systems where we aim to have a constant execution time (BCET = WCET) in all tasks of the system. As a consequence, approaching a constant execution time also means that the time predictability of the system is increased.
Chapter 2. Concepts in Real-Time Systems and Scope of this Thesis

Figure 2.2: Relationship between the improvement of analyzability and time predictability in real-time systems

2.6 Domains in timing analysis

Timing analysis is a very difficult task that depends on the following factors [20]:

- Source Code: The actual implementation of the program is here, the selection of an algorithm will define if certain tasks take more or less time to execute. Since the source code contains a large number of different paths that a program can follow, its timing analysis becomes harder to accomplish.

- Compiler: The compiler maps the source-level code into the machine-level code. This mapping is not unique, depending on how a compiler being used is implemented; the paths that were identified at the source code level can be changed at the machine code level. This is due to the presence of optimization techniques that a compiler performs at compile-time.

- Machine Architecture: The different features of the hardware (e.g. processor, pipelining, caching, I/O devices) that a program is running on, will effect on a program’s execution time.
Chapter 2. Concepts in Real-Time Systems and Scope of this Thesis

- Operating System: The operating system determines the task scheduling and memory management, both of which have an impact on the execution time of a program.

Each of these factors represents a different domain, where we can identify different problems and threats that can increase the complexity of timing analysis.

In my thesis, I only focus on the source code domain. The control semantics defined in a (real-time) programming language can produce different sequence of actions. A sequence of actions is also is called an execution path, which starts with a valid initial store at the starting point, follows the semantics of the actions, and terminates at an end point of the program [29].

The main difficulty is to characterize the different execution paths that are present [28]. The more paths defined in the source code the more effort we will need to characterize the possible longest execution path, thus making the analysis of the paths much harder. This problem is analyzed in more detail later (section 3.1.1).

2.7 Obstacles in timing analysis

There have been many research projects in the past years, addressing and trying to solve different problems in the three domains discussed in section 2.6 [28]. However, despite all the efforts, there are still three main obstacles that make timing analysis a very hard task to perform [29]:

1. Limits of automatic path analysis. The high dependability of control flow on input data leads to a poor prediction of execution paths. Therefore, if we try to depict possible execution paths only through source code analysis, we will require additional information about the data inputs and the effect of these on the control flow.
2. Lack of hardware-timing data. Modern hardware architectures present mechanisms that enhance the average execution performance of instructions, such as: caches, pipelines, branch prediction, etc. These characteristics make the prediction of time execution a very hard task. Additionally, the scarce documentation about timing, provided by the manufacturers, makes timing analysis even worse.

3. Complexity of analysis. Timing analysis presents some similarities with testing, which makes it a hard and complex problem. An exhaustive analysis of all paths in a program is unfeasible [2], because of the presence of loops; a program may have infinite number of paths. Furthermore, the number of paths grows exponentially for each consecutive branch that is inserted in the analyzed code.

2.8 Overview on testing and testability

Software engineering is a field of computer science, which tries to apply different approaches (i.e. techniques, methods, tools), that are systematic, disciplined and quantifiable, to the development, operation, and maintenance of software. Among these approaches we can find software testing.

“Software testing is a widely used and accepted approach for verification and validation of a software system” [13]. Many authors have stated that testing requires at least half of the time needed to produce a program [4], and not always is very efficient, because there is a possibility that some faults could not been discovered during this phase.

This is where testability analysis comes in as a concept that can help to reduce the testing effort spent in software development. A testability analysis can tell us where faults are more likely to hide [38] or how many test cases (e.g. number of
paths) we will require to cover as much as possible the entire source code [2].

From the timing analysis’ point of view, we can state that the more paths (low testability) in the source code we need to cover; the less analyzable with respect to timing a program is expected to become. The concepts of testing and testability are explained later (Chapter 4).

2.9 Objective of this thesis

In sections 2.6 and 2.7, I showed how different domains and parts of a program under analysis contribute to make timing analysis a very difficult and hard task. Previous works [8, 25, 27, 29] have identified two methods to simplify timing analysis in different domains; these two methods are:

1. Single-Path Approach. This technique transforms, when possible, the code into one single execution path. It relies on special instructions at the hardware level, which have a constant, data-independent execution time. Unfortunately, these instructions are only present in few modern processors (Motorola M-Core, Alpha) [25, 29].

2. WCET-Oriented programming. This programming paradigm aims to use code that avoids input-data dependent control decision or restricts operations that are only executed for a subset of input-data space to a minimum [27].

These two quite new methods have yielded promising results that reduces the timing analysis effort [27]. It is in this way that the motivation for this work comes from the necessity to introduce more methodologies and contribute to the existing ones that enable us to reduce, perhaps not completely, the effort that developers
need to invest in timing analysis techniques and, at the same time, increase the
time predictability of real-time systems.

The aim of this thesis is to:

- Show how testability analysis information for path testing, can help to esti-
mate more accurately the effort needed to perform a dynamic timing analy-
sis of a software program, by counting the number of paths present in the
source code.

- Identify patterns in the source code with high testability for future imple-
mentations that can help us enhance a program’s timing analyzability, de-
crease the variance of a program’s execution time, and reduce, in this way,
the effort required for timing analysis.

- Measure the worst-case execution time and best-case execution time of dif-
ferent implementations applying the identified patterns, in order to verify
whether or not the patterns help to lessen the effort in the timing analysis
and to improve the time predictability of those implementations.

2.10 Summary

In this chapter, I have introduced all the concepts and terminology in real-time
systems that are important for this thesis and that help to set the context of this
thesis. First, I have mentioned the concept of a real-time system and how the
presence of timing requirements, besides functional requirements, make them dif-
ferent from typical software systems. Then, we have discussed the importance of
the different concepts of execution time (BCET and WCET), and mentioned the
two techniques, static and dynamic timing analysis, that are applied for the predic-
tion of the execution time of a task. After that, we have seen two very important
Chapter 2. Concepts in Real-Time Systems and Scope of this Thesis

corcepts, predictability and analyzability, the relationship between them and how they can help us to improve timing analysis. Finally, I have briefly mentioned the problems domains that we can encounter when performing a timing analysis, and how timing analysis has similarities with testing. All this has lead to define my motivation and objective in this thesis.

In the upcoming chapter, I will discuss more deeply the different problem domains in timing analysis and how different programming paradigms contribute to worsen the process of timing analysis.
Chapter 3

Problems and Threats in Timing Analysis

So far, I have stated that timing analysis is a very difficult task. In this chapter, I will explain in more detail why timing analysis is difficult, by depicting the different problems and threats that we can encounter when we carry out a timing analysis for a real-time system. I will mention the problems that are present in different domains, emphasizing those that are present at the source code domain.

3.1 Problem Domains in Timing Analysis

Timing analysis is a very important phase in a development process of a real-time system. It validates the temporal correctness of a real-time program, by predicting the best and worst execution times of the source code of a program and it sets the foundation on which scheduling and task assignment will be performed.

Measuring the worst/best case execution time can be derived from the information that we can extract from: the possible sequences of program actions during execution and the time needed for every action in a sequence [33]. However, these
two kinds of information have to be treated differently at three different levels. The first level is the high-level programming language. The second level is the machine level that models the source code into the target hardware where the program is going to run. The modeling of the source code into machine code is done by a compiler that influences the results of the execution time, which becomes a third level of analysis in timing analysis.

These three levels become three problem domains that are present in timing analysis, because they present different aspects that can lessen the analyzability of a real-time system. These three domains have to be addressed in order to meet the requirements that timing analysis must fulfill (section 2.3). These domains are:

1. Characterization of execution paths. An execution path is a sequence of actions beginning at a starting point, follows the semantics of each action, and finishes at an end point in the source code [29]. Due to the presence of a large number of paths in the source code, the identification of the path that yield the worst-case execution time is difficult, this problem is explained in detail later (section 3.1.1).

2. Translation of path information from source code level to machine code level. This translation is done by compilers that perform optimization operations, rearranging the source code, therefore defining new paths or eliminating others from the source code [28]. Section 3.1.2 explains better the problems that can occur in the translation of path information.

3. Hardware-level execution time analysis. In this domain we are interested in derive the execution time of every machine instruction. This execution time is composed of read/write memory access times, pipeline stages, and other hardware features that belong to the target machine where a real-time system is going to be implemented [33]. Section 3.1.3 will study this topic.
Figure 3.1: Three domains in timing analysis [28]

in detail.

Figure 3.1 shows a sketch of the three problem domains in timing analysis that are presented in this section, the highlighted part in yellow is the domain that is going to be analyzed in this thesis.

3.1.1 Characterization of executions paths

The source code of a program describes the different potential executions paths that a program may follow. An execution path is a sequence of actions that starts with a valid initial store at the starting point, obeys the semantics of the action and terminates at an end point of a the program [29].

Puschner [33] identifies two main problems that worsen the process of timing analysis due to path descriptions:

1. The limited expressiveness of the path description language. Some programming languages do not offer the opportunity to developers to introduce
extra information about the possible execution paths that a program may follow.

2. Lack of knowledge about the program execution characteristics. It is very hard to know in compile-time how many times a loop will be executed, or which path the program will follow in a decision branch (e.g. if-statement), because of the input data dependency (either directly or indirectly) of the branching expressions.

Four problems have been identified at source code level that make a program less analyzable:

1. The maximum number of iterations number of a loop, in general, cannot be extracted from the loop condition. Thus, the worst case execution time of a loop cannot be calculated in most cases using static analysis.

2. Recursive functions can lead to similar problems, as described above, because the maximal depth of recursive calls to a function cannot be determined statically.

3. Pointers to functions can reference functions with different execution times, where the knowledge of which function will be executed cannot be resolved in a static analysis.

4. The usage of goto that produces spaghetti code, which lacks of any structure. Furthermore, it is well known that its usage is not recommended for any kind of systems and is considered a bad coding practice in general [4].

These problems are more notorious in static timing analysis [30] than in dynamic timing analysis because when we execute the program the number of times a loop is traversed or a recursive function is called is not important anymore since
we know that usually at some point in time the process will end. To avoid these problems, real-time programming languages restrict the use of recursion and their syntax only accept bounded loops (e.g. Real-time Euclid) [15]. In other cases, some non-real-time languages are extended with new syntax that allows the programmer to insert extra information about the paths [19, 33]. For example, in Figure 3.2 we can see C-like new syntax structures that transform common loops into bounded loop, which guarantee that any loop will be executed a maximum number of times (represented by the keywords MAX_COUNT or MAX_TIME).

![C-like syntax extensions for bounded loops](image)

Even though these solutions have helped to increase a program’s analyzability and ease the effort needed to perform timing analysis of programs, we still require more methods and techniques that can contribute to continue facilitate the timing analysis process. For example, the authors in [37] suggest introducing new coding guidelines that restrict the implementation language to a disciplined subset. Following this suggestion, in this thesis I introduce a coding guideline that focus on the replacement of if-statements with calculation, in order to have less number of paths to characterize. This new approach is explained later (Chapter 5).
3.1.2 Translation of path information

The path information that is present in a high level language (e.g. Java) is translated by a compiler to the machine language. This task is not a simple one-to-one mapping, many compilers perform many optimizations to obtain more efficient code (speed execution, low CPU usage), by rearranging the source code.

This new rearrangement in this source can impact directly in the execution paths that were identified at the source code level, which can lead to new degrees of unpredictability of the program that may have been avoided when coding at a higher level.

Different solutions to this problem can be found in the literature, for instance the use of timing analysis techniques that are compiler-independent [31]. Another solution to this problem is to perform the timing analysis at an assembly-code level avoiding, in this way, the translation of path information performed by the compiler [28]. Further study of these problems are not the may concern in this thesis.

3.1.3 Hardware-level execution time analysis

This is the ideal domain where timing analysis can be performed, because it is the place where the execution times of program actions, that is a machine instruction, are more accurate. The aim of hardware-level execution time analysis is to extract the time required by a machine instruction to be executed and to compute the WCET bound for a given piece of code.

The Hardware-level execution time analysis for simple architectures, where only one instruction is executed at a time and memory access times do not change during execution, is straightforward. On the other hand, timing analysis for more complex architectures is more complex. Some features that we must take into
consideration when performing timing analysis on complex architectures are:

- **Caches.** The memory access time of data and instruction depends on cache mapping strategies, replacement strategies, cache size and level of the memory hierarchy.

- **Pipelines.** Execution time of an instruction relies on: the number of pipeline stages, interdependencies between different pipeline stages (e.g. result from an instruction not yet executed).

- **Branch prediction.** Some instructions are prefetched and loaded into the cache before the processor actually needs them for computation.

- **Parallelism.** Execution times of instructions can be influenced by the number, performance and synchronization of parallel units.

This domain is not my main interest for this thesis, therefore it will not be analyzed anymore.

### 3.2 Problems in Timing Analysis for Object-Oriented Programs

The problems, characterization and translation of execution paths and hardware level timing analysis, described in section 3.1 are the most common ones that we can find in any typical real-time system. Nevertheless, in other systems that follow other developments paradigms, one may encounter other drawbacks that increase the complexity to analyze them. For example, in object-oriented programs we have to pay attention to other features that are particular to these kinds of systems [14, 37]. Some of these characteristics are:
Chapter 3. Problems and Threats in Timing Analysis

- Dynamic allocation of memory. This operation is performed when creating new objects. It is not always predictable the time needed that a program will require to allocate a certain size of memory.

- Deallocation of memory. Either explicitly (class destructors, delete command in C++) or automatically (garbage collectors in Java or Smalltalk), deallocation of memory is also very unpredictable.

- Dynamic binding. Inheritance and polymorphism creates additional complexity to timing analysis, because it is not known at compile-time which method is going to be executed.

- Middleware. Object-oriented programs usually are combined with middleware, which offers different services. These services may reside in the local processor or in a remote server, affecting, in this way, the time response of such services, and the global execution time of the program in general.

Although, the problems in timing analysis for object-oriented programs are not going to be further studied in this thesis, they need to be presented to give a general perspective of what we have to deal with timing analysis.

3.3 Particular Problems in Dynamic Timing Analysis

One of the main characteristics of dynamic timing analysis is that we can look at it as a special case of testing, specifically white box testing. Because, instead of generating test cases to find bugs and hazards in the source code, by covering as much as possible the source code; now we are interested in finding a test case that is capable to discover a path that yields the worst/best execution time. In
other words, in dynamic timing analysis we generate test cases in order to cover as much as possible all the paths in a program’s source code. Therefore, the more paths that are covered in the source code, the higher our confidence will be about our measurements of the WCET/BCET of that piece of code.

Dynamic analysis, as a special case of testing, also inherits the disadvantages that testing presents. The main problems are:

- Testing itself is already a problem, because it consumes at least half of the time needed to produce a program [4].

- Exhaustive testing is impracticable, due to the presence of infinite number of paths that may be present in the source code [2].

- It is not completely reliable, because if a set of test cases has not found faults in a program, does not necessarily mean that the program is fault-free. From timing analysis’ point of view, if a set of test cases yields a WCET of a program, it does not necessary mean that is the “worst” WCET.

3.4 Summary

In this chapter, I have showed the three domain problems that are present in timing analysis, from high-level source code to hardware level. I have explained how each of these domains contributes to inhibit timing analysis in real-time systems, and described how research has been conducted to solve the problems in these domains. I have also described how similar dynamic timing analysis and testing are, and therefore how dynamic timing analysis can inherit the same problems that we have to deal when testing a program.

In the following chapter, I will discuss testing in detail, testability and how both can help us to improve timing analyzability in real-time systems.
Chapter 4

Testing, Testability and Testability transformations

Thus far, I have described the strong relationship between testing and dynamic timing analysis. I have stated that dynamic timing analysis can be viewed as a special case of testing. That is why, in this chapter, first I will introduce and analyze concepts in testing, such as testing techniques, that later will be the foundation to define testability and testability metrics. Once we have enough knowledge about testability and how we can measure it, I will introduce the concept of testability transformations, and show how these transformations can help us to reduce the effort in testing a program (i.e. reducing the number of paths), and therefore reduce the effort in dynamic timing analysis.

4.1 Testing

“Software testing is a verification process for software quality assessment and software quality improvement” [10]. The primary purpose of the testing phase is to assess two main characteristics or attributes of a software system:
Chapter 4. Testing, Testability and Testability transformations

- Correctness: To show whether or not the final product is in accordance to its original requirements specifications.

- To support the confidence in its safe and correct operation [13].

“Software testing is a widely used and accepted approach for verification and validation of a software system” [13]. These two activities are defined by the IEEE Standard Glossary of Software Engineering Terminology as following [18]:

**Validation.** It is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements. This can be represented by the question of whether we are building the right system.

**Verification.** It is the process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase. This may be represented by the question of whether we are building the system right.

In these two processes, software testing is performed to assess or estimate correctness and reliability of the system through discovering faults in order to correct them with test-debug-fix cycles [10]. The more test-debug-fix cycles are performed, the more reliable the system is expected to become. Although, there is a limitation to that rule, when no more defects will be found, even with the most sophisticated quality assurance techniques applied.

Furthermore, many authors have stated that testing and/or test design consumes at least half of the time needed to produce a program [4]. Testing is not only hard to do but also very inefficient, since when a set of test cases that could not find any fault in a program does not necessarily mean that the program is fault-free.

This is where software testability comes in as a concept that can help to reduce the testing effort spent in software development and estimate the effort required
to perform a dynamic timing analysis. The IEEE Standard Glossary of Software Engineering Terminology has two definitions of testability [18]:

1. The degree to which a requirement is stated in terms that permit the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

2. The degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met.

This leads to two points of view. The first one, based on the first definition, is to see testability as the effectiveness of a testing technique, in other words the ability of a testing technique to uncover faults. Some interesting results have been presented on this issue [4].

In this thesis, I look at testability from a different point of view that is based on the second definition. I see software testability as the effort needed to test a software program using structural testing strategies, and this, in fact, sees testability as the ability of a program to hide faults. In this way, we can try to optimize the source code of a program, so that it can tell us if we can improve its testability. For instance, by reducing the number of paths in the source code, so that we can have less number of paths look for the worst-case execution time of the program. To achieve this, we have to find a measure for this type of testability, then analyze a program’s source code and apply the measure, generalize this knowledge in patterns for future reference, and, eventually, change the program code.
4.2 Program Analysis Techniques

Prior to any attempt to define testability metrics, so that later we can optimize the source code, in my case to improve its testability and thus facilitate timing analysis, we need to study different techniques for source code analysis, so that the structure of the program can be well known before we can apply those testability metrics. In this section, I present static analysis techniques that first help us to learn the source code’s structure and second help us to define testing techniques that will be explained in the next section.

Static program analysis techniques are widely used especially in compiler optimization, program understanding, program slicing, program verification tools, etc [22]. Many of these techniques are based on the structure of the source code of a program. For this thesis, we can think of two parts of a program’s structure [9]:

- Control flow structure, which addresses the sequence in which instructions are executed in a program.

- Data flow structure follows the trail of a data item from the moment it is created until it is not used anymore.

In my case, I will only base my program analysis on control flow structure techniques, because in timing analysis at source code level, we are only interested in the execution paths and not in the behavior of data items throughout a program execution.

The analysis of each structure begins with the construction of an abstract data structure called flowgraph that represents a procedure or a program. A flowgraph is a directed graph $G = \langle N, E \rangle$ where each node $N$ represents a program statement and each edge indicates the flow of control from one node to another. There
are two special nodes the start node $N_s$ and a unique stop node $N_x$, with outdegree (i.e. number of edges leaving a specific node) zero, and all other nodes lying between them. There are two main types of nodes [2]:

- Selection nodes which have an outdegree of two or more. These represent all kind of branching of control.

- Procedure nodes which have an outdegree of one, representing inputs, outputs and assignments.

4.2.1 Control Flow Analysis (CFA)

One of the studies in CFA is the decomposition of flowgraphs, which is better explained by the opposite concept, composition [2]. The decomposition of flowgraphs will help us to define testability metrics for different testing techniques, how this is achieved is explained in section 4.4.1.

There are two operations that can be used to compose two or more flowgraphs in a bigger flowgraphs, these operations are [2]:

- Sequencing: Two flowgraphs are sequenced when the stop node of the first one is joined onto the start node of the second flowgraph.

- Nesting: One flowgraph $F_1$ is nested in another flowgraph $F_2$ when one of the procedure nodes of $F_2$ is replaced by $F_1$.

The definition of these two operations leads to another concept, prime flowgraphs. “A prime flowgraph is one which cannot be created by either sequencing or nesting other non-trivial flowgraphs” [2]. Prime flowgraphs are one of the most important concepts that we require to define testability metrics. All its operations, such us: composition and decomposition are going to be required later, when we will need to identify different testability transformations.
Chapter 4. Testing, Testability and Testability transformations

In Figure 4.1 we can see some prime flowgraphs that can be found in many imperative languages. In Figure 4.2 shows how two flowgraphs can be composed by applying either a sequencing operation (Figure 4.2a) or a nesting operation (Figure 4.2b).

![Figure 4.1: Some prime flowgraphs. a) if-then b) if-then-else c) while-do d) switch-case e), f), g) sequences. Every black dot represents a statement in the source code](image)

The first result of prime flowgraphs is that any flowgraph can be created from applying, as many times as needed, the sequencing and nesting operations to the prime flowgraphs, and this composition is unique. On the other hand, we can decompose uniquely a flowgraph into prime flowgraphs. More or less this decomposition resembles the factorization of algebraic expressions.

### 4.3 Software Testing Techniques

Testability depends strongly on testing techniques, and, together with prime flowgraphs, it will help us to define different testability metrics. In this section, I present path testing techniques, because they have similarities with dynamic timing
analysis. For instance, path testing can show us how to select different inputs in order to cover as many paths as possible that are in the source code, thus giving us a high confidence that one of this test cases will yield the worst-case execution time. As we will see in the next section, these techniques will contribute to define testability metrics that will help us to measure the testability of the source code of a program, and with these testability metrics we can estimate how much effort (e.g. number of paths) we will need to analyze the execution time of a piece of code.

One of the main purposes of testing is to uncover faults that may have been introduced in previous stages of the software development process. This is done by selecting a finite number of possible inputs to a program and comparing the outputs with the expected results for those inputs.

The process of how to select inputs is defined by testing techniques. There are two well known strategies for selecting data: black box (or functional) testing and white box (or structural) testing [2]. For this thesis, we will focus our study in
different testing techniques that can be found under the white box testing concept. Beizer divides structural testing into [4]:

- Path testing
- Transaction-flow testing
- Data-flow testing
- Domain testing

In this thesis, I am only interested in path testing, which is going to be explained in more detail in the following section.

4.3.1 Path Testing

Path testing is a family of testing techniques based on selecting a set of execution paths through a program [2]. It generates tests in such a way that we can achieve some measure of test thoroughness, such as: assuring that every statement has been executed at least one. I chose this testing technique, because dynamic timing analysis has to search for the path that will produce the worst-case execution time. In theory, if we cover all paths with the test cases generated by this testing technique we can be one hundred percent sure that one of this will find the path with the longest execution time (WCET).

Among the different testing criteria that we can find under this testing technique, path testing is the most suitable for this thesis. In path testing, our goal is to try to execute all possible control flow paths through the program. This criterion is the strongest and generally impossible to achieve in the path-testing strategy.

In general, the number of paths in the source code will be infinite, due to the possible presence of loops [24]. This drawback makes the definition of a metric for path testing impossible to achieve [2, 24]. However, we can obtain a finite
measure if we focus only on simple paths, as we will see in the next section. Simple paths are those for which no edge is traversed more than once [24].

There are other testing criteria that belong to the path testing family. For example, statement testing that looks to execute all statements in the program at least once; branch testing that tells us to use at least once every branch alternative, and others [4]. However, their studies for the purposes of this thesis are irrelevant.

4.4 Software Testability and Testability Measures

Software testing helps to assess two software attributes, correctness and reliability. Furthermore, software testability analysis combined with software testing can be a powerful tool to achieve high reliability in systems giving us a better picture of the “true reliability” of the system [38]. Software testing will still reveal as many faults as possible, while testability will tell us where faults are more likely to hide or how much testing effort we will need in some places of the system. From dynamic timing analysis’ point of view, testing or timing analysis will give us the worst/best case execution time of a program by executing it several times and with different data sets, while testability will tell us which parts of the program are more difficult to analyze in terms of time, for example, by estimating the number of paths that are required to analyze.

It is important to define clearly the difference between testing and testability. We can analyze how these techniques view the code. Testing is only interested in the relationship between inputs and outputs as in execution time, thus it has a black-box perspective to it. On the other hand, testability analyzes the source code, so it is classified as white-box. It is important to remark that these terms must not be confused with white and black-box testing [10].

The study of testability can be divided into two main concepts [38]:

37
1. Design for testability, which tries to show how testability can enhance testing and improve software testability in early stages of the software process development.

2. Testability metrics so we can measure the degree to which we have achieved it.

The first concept is well studied by Pettichord [23], where he defines testability as visibility, the ability to observe the states, output values and other information of the software under test; and control, as the ability to put inputs to the program under test or in any intermediate state. In his work he suggests how testability can be improved by the modification of some software features, such us: logging events (e.g. verbose output), monitoring that provides access to the internal workings of the code, user interface testability.

Binder, also, studies design for testability specifically for object-oriented systems and defines it as “a strategy to align the development process so that testing is maximally effective under either a reliability or resource-limited regime” [6].

Other authors put their effort on the second concept and try to come up with new methodologies to quantify the testability of the source code of a system. Following this criterion, [10] states that: "Software testability is a software metric that refers to the ease with which some formal or informal testing criteria can be satisfied". This means, if branch coverage, or any other testing technique mainly described by Boris Beizer [4], is used as a testing criterion and is easy to find test cases that match this criterion, then the piece of code under scrutiny can be considered to have a high degree of testability. Formally speaking, “the testability of a program P is the probability that if P contains fault(s), P will fail under test” [16].

Bertolino has a more concrete definition of testability, “The testability of a program is the probability that a test of a program on an input drawn from a specified probability distribution of the inputs is rejected, given a specified oracle
and given that the program is faulty” [5]. The oracle can be a human tester or a test program; in any case there is a probability that the oracle may have faults, which seems to be the concern of Bertolino, she includes it in her definition of testability.

Nevertheless, what Bertolino is forgetting, from my point of view, is that software testability is not about whether the source code is producing wrong results or not. The program is not compared to a specification, so no oracle is required. Its only concern is to know if the program is capable of revealing its faults during testing.

Another concept of software testability specifies that testability is a quality attribute that defines the effort needed for testing [2, 40]. Furthermore, Bach and Yeh state that testability measurements are strongly related to the test data selection strategies that are used for testing or the kind of program analysis technique we use to inspect the source code [2, 40]. That is to say, that testability measures will differ if we decide to choose a branch testing technique from a simple path testing. Based on these last concepts, I see software testability as the effort needed to test a software program, or in other words the effort needed to analyze a software program in terms of time. For example, we can count the number of paths that we will require to cover in order to find the one that will produce the worst-case execution time. To achieve this we will need to use structural testing strategies, specifically simple path testing.

### 4.4.1 Testability Measures for control flow testing strategies

An important part of testability is its measurement. Many authors have developed testability measures using different criteria [2, 10, 40]. However, [2, 40] have one thing in common, they base their measurements on a previous program analysis.

Software testability and testing strategies are strongly connected. Depending
on which testing strategy is used, the way to assess the testability metrics may differ from each other, for example the formulas to calculate these metrics may change. Many of these metrics will help us to assess the source code’s testability. However, I am only interested in testability metrics for simple path testing, because this metric is the most appropriate to estimate the effort need to perform timing analysis operations. This metric will later help us to find patterns with high testability that will reduce the number of simple paths in other programs that implement them.

The group of testability metrics that are more suitable for my thesis is based on control flow testing strategies. [2] defines these measurements as the minimum number of test cases required to provide total test coverage. Within this group we can find testability measures for the following testing strategies: all paths testing, visit-each-loop path testing, simple path testing, structured testing, branch testing and statement testing.

The process of the measurement begins with the decomposition of a flowgraph into primes, these primes are: IF-THEN ($D_0$), IF-THEN-ELSE $D_1$, WHILE ($D_2$), SWITCH-CASE $C_n$, and simple sequence $P_1$. Each of the decomposed prime flowgraphs receives a value. These values can be seen in Table 4.1 for each testing strategy.

<table>
<thead>
<tr>
<th>Test Strategy</th>
<th>$\mu(P_1)$</th>
<th>$\mu(D_0)$</th>
<th>$\mu(D_1)$</th>
<th>$\mu(D_2)$</th>
<th>$\mu(C_n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>All paths testing</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>n</td>
</tr>
<tr>
<td>Visit-each-loop-path testing</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>Simple path testing</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>Structured testing</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>n</td>
</tr>
<tr>
<td>Branch Testing</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>n</td>
</tr>
<tr>
<td>Statement testing</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>n</td>
</tr>
</tbody>
</table>

Table 4.1: Metric values of prime flowgraphs for different testing strategies

After every prime flowgraph has its metric value, we continue to calculate the
metrics of the nesting and sequencing functions using the functions depicted in Table 4.2 and Table 4.3.

<table>
<thead>
<tr>
<th>Test Strategy</th>
<th>$n \prod_{i=1} \mu(F_i)$</th>
<th>$n \prod_{i=1} \mu(F_i)$</th>
<th>$n \prod_{i=1} \mu(F_i)$</th>
<th>$\sum_{i=1} \mu(F_i) - n + 1$</th>
<th>$\max(\mu(F_1, \ldots, \mu(F_n)))$</th>
<th>$\max(\mu(F_1, \ldots, \mu(F_n)))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>All paths testing</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\sum_{i=1} \mu(F_i) - n + 1$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
</tr>
<tr>
<td>Visit-each-loop-path testing</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\sum_{i=1} \mu(F_i) - n + 1$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
</tr>
<tr>
<td>Simple path testing</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\prod_{i=1} \mu(F_i)$</td>
<td>$\sum_{i=1} \mu(F_i) - n + 1$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
</tr>
<tr>
<td>Structured testing</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\sum_{i=1} \mu(F_i) - n + 1$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
</tr>
<tr>
<td>Branch Testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
</tr>
<tr>
<td>Statement testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
<td>$\max(\mu(F_1, \ldots, \mu(F_n)))$</td>
</tr>
</tbody>
</table>

Table 4.2: Metric functions for sequencing operations

<table>
<thead>
<tr>
<th>Test Strategy</th>
<th>$\mu(D_1(F_1, F_2))$</th>
<th>$\mu(C_n(F_1, \ldots, F_n))$</th>
<th>$\mu(D_0(F))$</th>
<th>$\mu(D_2kF)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>All paths testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
</tr>
<tr>
<td>Visit-each-loop-path testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
</tr>
<tr>
<td>Simple path testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
</tr>
<tr>
<td>Structured testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
</tr>
<tr>
<td>Branch Testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
</tr>
<tr>
<td>Statement testing</td>
<td>$\mu(F_1) + \mu(F_2)$</td>
<td>$\sum_{i=1} \mu(F_i)$</td>
<td>$\mu(F) + 1$</td>
<td>$\mu(F) + 1$</td>
</tr>
</tbody>
</table>

Table 4.3: Metric functions for nesting operations

In Table 4.1 and Table 4.3, we can see a result of the impossibility to achieve full path-testing coverage; the metrics for the while-do prime are undefined. That is why the concept of simple path testing was introduced in section 4.3.1. The testability metric for simple path testing provides an extremely useful estimate of
the testing effort (i.e. number of paths) of a piece of code [24].

4.5 Source Code Improvement

[3, 17] present a notable study of source code optimization to improve its testability, or what they call testability transformations. “A testability transformation is a source-to-source program transformation that seeks to improve the performance of some chosen test data generation”.

Before any attempt of testability transformation is made, [17] defines a set of conditions that must be fulfilled in order to consider a transformation suitable for testability improvement.

- Testing-Oriented Transformation. Let \( P \) be a set of programs, and \( C \) a set of testing criteria. A Testing-Oriented Transformation is a partial function in \( (P \times C) \rightarrow (P \times C) \)

- Testability Transformation. A Testing-Oriented Transformation \( \tau \) is a testability transformation, if and only if, for all programs \( p \) and a testing criteria \( c \) (i.e. branch testing, statement testing) and \( \tau(p, c) = (p', c') \), all test sets \( T \) are adequate to \( p \) according to \( c \) and also adequate to \( p' \) according to \( c' \).

- \( c \)-Preserving Testability Transformation. Let \( \tau \) be a testability transformation. If for criterion \( c \), for all programs \( p \), there is a program \( p' \) such that \( \tau(p, c) = (p', c) \), then \( \tau \) is called a \( c \)-preserving testability transformation.

Usually a \( c \)-preserving testability transformation is ideal for applications, since the test data generated for the transformed program can also be used for the original program. On the other hand, a simple testability transformation will guarantee that the original testing strategy is achieved by a set of test cases.
Chapter 4. Testing, Testability and Testability transformations

4.5.1 Examples

The following examples show how to increase the testability through source code optimization. In the first example, I studied if merging two or more consecutive nested if’s can help to increase the testability of the program using branch coverage as the testing criteria.

Algorithm 4.1 Biggest value out of three numbers

Input: a, b, c:integer
Output: max

1: if a > b then
2:     if a > c then
3:         max = a
4:     else
5:         max = c
6:     end if
7: else
8:     if b < c then
9:         max = c
10:    else
11:        max = b
12:    end if
13: end if
14: return max

Two different implementations, to calculate the biggest value of three variables, are shown in Algorithm 4.1 and Algorithm 4.2. I assessed the testability of each implementation using the testability that has been already explained in section 4.4.1.
Chapter 4. Testing, Testability and Testability transformations

Algorithm 4.2 Second implementation biggest value out of three numbers

Input: \( a, b, c \) : integer

Output: integer

1: if \( a > b \) and \( a > c \) then
2: \hspace{1em} return \( a \)
3: else
4: if \( b > c \) and \( b > a \) then
5: \hspace{1em} return \( b \)
6: \hspace{1em} end if
7: \hspace{1em} end if
8: return \( c \)

• Testability measurement for Algorithm 4.1.

\[
T_1 = \max[\mu(D_1^1(D_2^2(P_3^{01}, P_5^{01})), D_3^8(P_9^{01}, P_1^{11}))), \mu(P_1^{14})]
\]

\[
T_1 = \max[\mu(D_1^2(P_3^{01}, P_5^{01}) + D_3^8(P_9^{01}, P_1^{11})), 1]
\]

\[
T_1 = \mu(P_3^{01}) + \mu(P_5^{01}) + \mu(P_9^{01}) + \mu(P_1^{11})
\]

\[
T_1 = 1 + 1 + 1 + 1 = 4
\]

• Testability measurement for Algorithm 4.2.

\[
T_2 = \max[\mu(D_1^1((P_2^{01}, D_0^4(P_5^{01}))), \mu(P_1^{08})] \]

\[
T_2 = \max[\mu(D_1^1(P_2^{01}, P_5^{01}) + 1, 1]
\]

\[
T_2 = \mu(P_2^{01}) + \mu(P_5^{01}) + 1
\]

\[
T_2 = 1 + 1 + 1 = 3
\]

The results show that testability is improved by 1 if we transform the implementation in Algorithm 4.1 to a more easily testable implementation, in this case Algorithm 4.2.

The previous example was performed to show how we can reduce the testing.
effort in general. For the following example (Algorithms 4.3, 4.4 and 4.5) I used a testability metric for simple path testing. This metric is going to be used in the next chapter to analyze different patterns that will help us to reduce the number of paths in a program so that we can improve the timing analyzability of real-time system. In this last example, I analyzed and measured the testability of different algorithmic implementations, showed in Algorithms (4.3, 4.4 and 4.5), for searching an item in an array of elements.

Algorithm 4.3 Binary Search algorithm for sorted lists

<table>
<thead>
<tr>
<th>Input:</th>
<th>key, myArray[] : integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output:</td>
<td>integer</td>
</tr>
</tbody>
</table>

1:  \( l = 0 \)
2:  \( r = myArray.length \)
3:  \( \text{do} \)
4:  \( i = (l + r)/2 \)
5:  \( \text{if} (key < myArray[i]) \text{ then} \)
6:  \( r = i - 1 \)
7:  \( \text{else} \)
8:  \( l = l + 1 \)
9:  \( \text{end if} \)
10: \( \text{while} (key! = myArray[i] && l < r) \)
11: \( \text{if} (key == myArray[i]) \text{ then} \)
12:  \( \text{return} i \)
13: \( \text{end if} \)
14: \( \text{return} -1 \)

I assessed the testability of each implementation using the simple path coverage, as a testing criterion. The results are the following:

- Testability measurement for Algorithm 4.3.

\[
T_1 = P^1_1 \cdot P^2_1 \cdot \mu(D^3_2(P^4_1, D^5_1(P^6_1, P^8_1))) \cdot \mu(D^{11}_0(P^{12}_1)) \cdot P^{14}_1
\]

\[
T_1 = 1 \cdot 1 \cdot (\mu(P^4_1, D^5_1(P^6_1, P^8_1)) + 1) \cdot 2 \cdot 1
\]

\[
T_1 = ((1 \cdot 2) + 1) \cdot 2 = 6
\]
Chapter 4. Testing, Testability and Testability transformations

Algorithm 4.4 Sequential Search algorithm for sorted lists

**Input:** key, myArray[] : integer

**Output:** integer

1: \( i = 0 \)
2: \( \textbf{while} \ (\text{key} > \text{myArray}[i]) \ \textbf{do} \)
3: \( i = i + 1 \)
4: \( \textbf{if} \ i == \text{myArray}.\text{length} \ \textbf{then} \)
5: \( \textbf{return} \ -1 \)
6: \( \textbf{end if} \)
7: \( \textbf{end while} \)
8: \( \textbf{if} \ (\text{key} == \text{myArray}[i]) \ \textbf{then} \)
9: \( \textbf{return} \ i \)
10: \( \textbf{end if} \)
11: \( \textbf{return} \ -1 \)

- Testability measurement for Algorithm 4.4.

\[
T_2 = P_1 \cdot \mu(D_2^2(P_1^3; D_0^4(P_1^5))) \cdot \mu(D_0^8(P_1^9)) \cdot P_{11}^1
\]

\[
T_2 = 1 \cdot \mu(D_2^2(1 \cdot 2)) \cdot 2 \cdot 1
\]

\[
T_2 = (2 + 1) \cdot 2 = 6
\]

- Testability measurement for Algorithm 4.5.

\[
T_3 = P_1 \cdot \mu(D_2^2(D_0^3(P_1^4); P_1^9)) \cdot P_{11}^8
\]

\[
T_3 = 1 \cdot \mu(D_2^2(1 \cdot 2)) \cdot 1
\]

\[
T_3 = (2 + 1) \cdot 1 = 3
\]

These results are very interesting. Firstly, the result between the two sequential implementations, Algorithms 4.4 and 4.5, shows that while Algorithm 4.4 is more efficient in time than Algorithm 4.5 [35]; testing Algorithm 4.5 will be less hard than testing Algorithm 4.4. Secondly, the testability results between Algorithm 4.4 and Algorithm 4.3 are the same, which, from my point of view, makes it
Chapter 4. Testing, Testability and Testability transformations

Algorithm 4.5 Sequential Search algorithm for any list of numbers

**Input:** key, myArray[] : integer  
**Output:** integer  
1: \( i = 0 \)  
2: while \( (i < myArray.length) \) do  
3: \( \text{if } key == myArray[i] \text{ then} \)  
4: \( \text{return } i \)  
5: \( \text{end if} \)  
6: \( i = i + 1 \)  
7: \( \text{end while} \)  
8: \( \text{return } -1 \)

easier to choose Algorithm 4.3 over Algorithm 4.4, because the former algorithm is, again, more efficient in time than the later [35]. Furthermore, applying a simple path testing technique we can view how analyzable in terms of time an algorithm can be. In this particular case, Algorithm 4.5 supposed to be easier to analyze than the other two searching algorithms, because it has less number of paths that we require to find test cases for them.

### 4.6 Summary

In this chapter, I have presented an analysis about testing and testability and how they are related with timing analysis and how they can help to improve timing analyzability in real-time systems. I began with a description of testing, its importance as a validation and verification technique for software systems and described why testing is difficult and usually does not cover all the code. In order to measure difficulty and effort in testing, I have introduced the concept of testability as the effort needed to test a system or component and how this system or component facilitates the performance of tests.

In order to define testability metrics that are going to be used in the next chapter, first I had to talk about source code analysis, specifically control flow analysis,
and how we can decompose our source code in prime flowgraphs. And, second I discussed testing techniques, and defined simple path testing as the most suitable testing technique that can help us to measure the effort we require for a dynamic timing analysis. Both, prime flowgraphs and testing techniques have helped us to define the testability metrics and the formulas to measure the effort (i.e. number of test cases) we require to test of a piece of code. Finally, I have mentioned testability transformations and how they can help us to improve the performance of testing.

In the next chapter, I will present the main research topic of this thesis, which is introduce patterns (i.e. testability transformations) through source code analysis and I will show how these can help us improve the timing analyzability of a real-time system.
Chapter 5

Source Code Analysis and Pattern Identification

In chapter 4 we have studied the importance of testability analysis and how we can reduce the effort in software testing by identifying patterns in the source code with high testability, where high testability means source code with a low number of simple paths to cover. In case of dynamic timing analysis of real-time systems, we can apply the same principles. Because dynamic timing analysis is in fact testing with a slightly different test target (execution time violations), consequently we can reduce the effort needed for timing analysis of a real-time system (i.e. improving timing analyzability), by applying patterns that will reduce the number of paths in a program. In this chapter, I will present different patterns that I have found for this research, which can help us to reduce the number of simple paths, and, thus, increase a program’s timing analyzability and, at the same time, increase its time predictability.

As testability metric for the analysis, I chose a testability metric for simple path testing described in [2], because in dynamic timing analysis, our main concern is to find a test case that can traverse a path that yields the worst case execution time.
Therefore, if we can find patterns in the source code that decrease the number of simple paths, then we can reduce the effort to find the worst/best case execution of a program, because we need to analyze/execute fewer paths in the source code.

## 5.1 Patterns

Patterns in computer science were first introduced by Gamma [11]. In his book, Gamma shows the concept of a pattern in architecture depicted by Christopher Alexander, where he states that a pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over. This concept was successfully applied by Gamma in object-oriented design.

In my case, I use the term pattern to identify those pieces of source code that describe a problem (i.e. code that produces more paths than other code), occur over and over again, and will help us to improve the timing analyzability of a real-time system (i.e. reducing the number of simple paths in a program), so that we can use them in different applications.

In the next subsection, I will present a pattern that has already been studied. Afterwards, I will present two patterns that I have found for this thesis.

### 5.1.1 Constant-Time Conditional Expression Pattern

This pattern was introduced by Peter Puschner and Alan Burns [29], they call it “The single Path approach”. They propose a new “constant-time conditional expression operation”. This new operator tries to replace an if-statement in order to reduce the number of paths in the source code (e.g increase the source code
testability). The notation for this new operator is the following:

\[ \text{Cond} \ # \ Expr_1 : Expr_2; \]  \hspace{1cm} (5.1)

It consists of a boolean expression (Cond) and two expressions (Expr1, Expr2). First, it evaluates the two expressions and returns Expr1 if Cond is true or Expr2 if Cond is false. This operator resembles the already well known ternary operator that is present in some languages (e.g. C, Java):

\[ \text{Cond} \ ? \ Expr_1 : Expr_2; \]  \hspace{1cm} (5.2)

The difference between these two expressions is that operator 5.2 works like an if-then-else statement, evaluating the condition first and then executing one of two branches. On the other hand, operator 5.1 computes first both expressions in the branches and then evaluates the condition and returns one of the two expressions. This semantic is illustrated in Figure 5.1.

![Figure 5.1: Order of execution of the operations of a) expression 5.2 and b) expression 5.1 [29]](image)

51
Evaluation: Constant-Time Conditional Expression Pattern

The constant-time conditional expression pattern can be implemented in different algorithms [25, 26, 27, 29]. As an example I will show the work presented in [26], and how this pattern can be used for increasing the analyzability of the binary search algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Traditional</th>
<th>Transformed</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average Case</td>
<td>WCET</td>
</tr>
<tr>
<td>Binary Search</td>
<td>94</td>
<td>124</td>
</tr>
</tbody>
</table>

Table 5.1: Execution times (in CPU cycles) of traditional and transformed binary search algorithms [26]

In Algorithm 5.2, we can see the transformation of the “traditional” binary search algorithm (Algorithm 5.1) after applying the constant-time conditional expression. Table 5.1 shows the results obtained in [26]. A disadvantage of the code in Algorithm 5.2 is that it always returns the index of the smallest element that is greater than or equal the key, therefore this implementation is not capable to tell us if an element is not present in the array.

We can perform a testability analysis on both Algorithms 5.1 and 5.2 to demonstrate how by implementing the pattern proposed by Puschner, we can increase the timing analyzability of the binary search algorithm. The results of the testability analysis are the following:
Algorithm 5.1 Traditional implementation of a binary Search algorithm

**Input:** key, myArray[] :integer  
**Output:** integer  
1: found = 0  
2: l = 0  
3: r = myarray.length  
4: do  
5: \( i = (l + r) \gg 1 \)  
6: if (key == myArray[i]) then  
7:     found = 1  
8: end if  
9: if (key < myArray[i]) then  
10:     r = i − 1  
11: else  
12:     l = l + 1  
13: end if  
14: while (!found&&r >= l)  
15: if (found) then  
16:     return i  
17: end if  
18: return −1  

Algorithm 5.2 Transformed implementation of Algorithm 5.1 using constant-time conditional expression pattern

**Input:** key, myArray[] :integer  
**Output:** integer  
1: left = 0  
2: right = myarray.length  
3: idx = (right + left) >> 1  
4: for (inc = SZ; inc > 0; inc = inc >> 1) do  
5:     right = (key < myArray[idx])#idx − 1; right  
6:     left = (key > myArray[idx])#idx + 1; left  
7:     idx = (right + left) >> 1  
8: end for  
9: return idx
• Testability measurement for Algorithm 5.1

\[
T_1 = \mu(P_1^1) \cdot \mu(P_2^2) \cdot \mu(P_3^3) \cdot \mu(D_2^5(\mu(P_5^5) \cdot \mu(D_6^6(P_7^7)) \cdot \\
\mu(D_4^9(P_1^{10}, P_1^{12})))) \cdot \mu(D_6^{15}(P_1^{16})) \cdot \mu(P_1^{18})
\]

\[
T_1 = 1 \cdot 1 \cdot 1 \cdot (\mu(D_2^5(1 \cdot 2 \cdot 2)) \cdot 2 \cdot 1)
\]

\[
T_1 = (1 + 4) \cdot 2 = 10
\]

• Testability measurement for Algorithm 5.2

\[
T_2 = \mu(P_1^1) \cdot \mu(P_2^2) \cdot \mu(P_3^3) \cdot \mu(D_2^5(\mu(P_5^5) \cdot \mu(P_6^6) \cdot \mu(P_7^7))) \cdot \mu(P_1^9)
\]

\[
T_2 = 1 \cdot \mu(D_2^5(1 \cdot 1 \cdot 1))
\]

\[
T_2 = (1 + 1) = 2
\]

This transformation proposed by Puschner was designed to be implemented at the translation path information and hardware levels (see Figure 3.1). One of the disadvantages is that they look at object code, certainly this depends strongly on the features of the target machine, requiring special instructions at the machine-code level that is only present in some processors (e.g. Motorola M-core, Alpha, Pentium P6, SPARC 96) [29], and a compiler that can map high-level source code into those low-level machine instructions. This indeed means a limitation of the usage of the constant-time conditional expression operator. In my thesis, I only look at high-level source code, proposing programming conventions that can give us the same result, reduction of paths and small gap between BCET and WCET, that is to say reduce the effort in dynamic timing analysis.

If we want to identify patterns that are independent from the hardware level, we will have to work at the source code level, which is what this thesis is proposing. The following patterns were identified taking into consideration the inde-
pendency of such patterns from the hardware level. Thus, these patterns can be implemented in any program, without worrying about the features of the hardware architecture on which that program is going to be executed.

5.1.2 Min/Max-Value Pattern

The first pattern that I have identified is used to calculate either the maximum or minimum value of two numbers. A typical implementation is presented in Algorithm 5.3, the proposed patterns to replace Algorithm 5.3 are depicted in Algorithms 5.4 and 5.5.

**Algorithm 5.3** Pattern to calculate the maximum value of two numbers

| Input: | a, b : integer |
| Output: | integer |
| 1: | max = 0 |
| 2: | if (a > b) then |
| 3: | max = a |
| 4: | else |
| 5: | max = b |
| 6: | end if |
| 7: | return max |

**Algorithm 5.4** maxValue: Testability transformation for Algorithm 5.3

| Input: | a, b : integer |
| Output: | integer |
| 1: | return \((a + b) + \text{abs}(a - b))/2\)

**Algorithm 5.5** minValue: Testability transformation for minimum value of two numbers

| Input: | a, b : integer |
| Output: | integer |
| 1: | return \((a + b) - \text{abs}(a - b))/2\)

In order to know if such a transformation can reduce the effort of timing analysis, we need to perform a testability analysis on Algorithms 5.3 and 5.4. Since
Algorithms 5.4 and 5.5 have similar structure, it is not necessary to do the analysis for Algorithm 5.5.

- Testability measurement for Algorithm 5.3

\[ T_1 = \mu(P_1^1) \cdot \mu(D_1^2(\mu(P_3^1), \mu(P_5^5))) \cdot \mu(P_7^7) \]
\[ T_1 = 1 \cdot 2 \cdot 1 = 2 \]

- Testability measurement for Algorithm 5.4

\[ T_2 = \mu(P_1^1) \]
\[ T_2 = 1 \]

The results of testability analysis show that the implementation in Algorithm 5.4 requires only one test case to cover all simple paths, in fact one only path, in the program. Although, the reduction from two simple paths to only one simple path does not seem a big improvement at this moment, applying this pattern many times in other implementations where comparisons of this kind are countless, the reduction of simple paths will be much more significant. Therefore, if we can implement this pattern in other programs we can also increase their testability, or in other words, reduce the effort to make a worst-case execution time estimate (i.e. improve timing analyzability).

**Evaluation: Min/Max-Value Pattern**

This pattern, although very simple, is used in many applications. Sorting algorithms are good examples where min/max functions are required. Continuing with my analysis, I have implemented the min/max-value pattern in three different sorting algorithms, bubble, selection and insertion sorting algorithms.
Algorithm 5.6 Traditional implementation of a bubble sort algorithm

**Input:** \( a, myArray[], SZ : integer \)

**Output:** sorted list

```plaintext
1: tmp = 0
2: for (i = 0; i < SZ; i + +) do
3:   for (j = i + 1; j < SZ; j + +) do
4:     if (myArray[i] >= myArray[j]) then
5:       tmp = myArray[j]
6:       myArray[j] = myArray[i]
7:       myArray[i] = tmp
8:     end if
9:   end for
10: end for
```

Algorithm 5.7 Implementation of a bubble sort algorithm applying min/max-value pattern

**Input:** \( a, myArray[], SZ : integer \)

**Output:** sorted list

```plaintext
1: for (i = 0; i < SZ; i + +) do
2:   for (j = i + 1; j < SZ; j + +) do
3:     max = maxValue(myArray[i], myArray[j])
4:     min = minValue(myArray[i], myArray[j])
5:     myArray[j] = max
6:     myArray[i] = min
7: end for
8: end for
```
Chapter 5. Source Code Analysis and Pattern Identification

In Algorithm 5.6, we can see a common implementation of a bubble sort algorithm. In Algorithm 5.7, I calculate the maximum and minimum value of every pair of elements of the array, using the min/max-value pattern.

To demonstrate whether this pattern is helping to reduce the number of simple paths in the algorithm, I carried out a testability analysis of both implementations.

- Testability measurement for Algorithm 5.6

\[
T_1 = \mu(P^1_1) \cdot \mu(D^2_2(\mu(D^3_2(\mu(D^4_0(\mu(P^5_1) \cdot \mu(P^6_1) \cdot \mu(P^7_1)))))))
\]

\[
T_1 = 1 \cdot \mu(D^2_2(\mu(D^3_2(\mu(D^4_0(1 \cdot 1 \cdot 1))))))
\]

\[
T_1 = \mu(D^2_2(1 + 1)) = \mu(D^2_2(1 + 2)) = 3 + 1 = 4
\]

- Testability measurement for Algorithm 5.7

\[
T_2 = \mu(D^1_2(\mu(D^2_2(\mu(P^3_1) \cdot \mu(P^4_1) \cdot \mu(P^5_1))))))
\]

\[
T_2 = \mu(D^1_2(\mu(D^2_2)(1 \cdot 1 \cdot 1)))
\]

\[
T_2 = \mu(D^1_2(1 + 1)) = 2 + 1 = 3
\]

The testability analysis results show how implementing the min/max value pattern in the bubble sort algorithm, reduces the number of simple paths for the timing analysis. As a second step, I inspected both implementations in a real time environment system (Linux OS extended with RTAI 3.1\(^1\)). I run both implementations to sort an array of 500 elements and applied dynamic timing analysis to measure the worst/best case execution time of both implementations.

In this particular case, the best and worst cases are easy to find and can be done manually. As we can see in line 3 of Algorithm 5.6, the expression in the

\(^1\)http://www.rtai.org
Chapter 5. Source Code Analysis and Pattern Identification

Figure 5.2: Results of execution time of sorting 500 elements using Algorithm 5.6

if-statement look for two consecutive items in the list where the first one is larger than the second one, therefore:

1. The best case will be a list of elements that is already sorted (e.g. [1,2,3,4]), so that the condition of the if-statement will always be false and it will not be required to evaluate the statements in Lines 5, 6, and 7 of Algorithm 5.6.

2. The worst case will be a sorting list that is inversely sorted (e.g. [4,3,2,1]), so that every pair of elements that is been evaluated will have to execute Lines 5, 6 and 7 of Algorithm 5.6.

In Figure 5.2 and Figure 5.3, we can see the results of the dynamic timing analysis of Algorithm 5.6 and Algorithm 5.7 respectively. These results present some interesting data that need to be pointed out:

1. This pattern causes a big increase in the execution time of the algorithm, almost by a factor of 6.

2. Figure 5.3 clearly shows how this pattern reduces the gap between BCET and WCET, where even in same cases the results overlap from each other.
On the other hand, in Figure 5.2 clearly shows how different the execution times are for the best and worst cases in a typical implementation of the bubble sort algorithm.

![Bubble Sort 2](image)

Figure 5.3: Results of execution time of sorting 500 elements using Algorithm 5.7

### 5.1.3 Single-Path-If Pattern

The pattern presented in section 5.1.1 is a good example on how to reduce paths in the object code of a program. Unfortunately, this pattern can only be implemented in few hardware architectures, which certainly is a disadvantage. This is the reason why I came up with a new way to replace the if-then-else statement by means of introducing calculations to evaluate the truth of an if-expression. These patterns give an alternative of the pattern in section 5.1.1 at a high-level programming language, and can be implemented in any hardware architectures, because the calculations, to be introduced later, do not depend on any specific feature at the hardware level.

This pattern replaces if-statements of the form:
if $A \Theta B$ then $C := \text{expr1}$ [else $D := \text{expr2}$]

where $\Theta$ represents any relational operator ($<$, $>$, $<=$, $>=$, $==$), $A$ and $B$ are any numeric expression, variable or constant. The pattern is implemented differently for each comparison operator. First, it requires a preprocessing phase of the numeric expressions, variables or constants present in the condition expression, in order to obtain values that are greater than zero, since later we will perform some divisions. Then, we calculate the maximum and minimum number of all these new values. With all these calculations we proceed to transform all the assignments statements that belong to the if-statement, using different formulas defined for each relational operator.

**Preprocessing phase**

This preprocessing stage is required for two purposes:

1. Transform the values of $A$ and $B$ to numbers that are greater than zero, because later calculations include some divisions and, therefore, we need to avoid divisions by zero.

2. Calculate the maximum and minimum numbers of those transformations, for this step we use the min/max value pattern described in section 5.1.2.

Algorithm 5.8 shows the source code that performs the steps above described.

**Algorithm 5.8** Source code of the preprocessing phase for the single-path-if pattern

1: $max = \maxValue(\text{abs}(A), \text{abs}(B))$
2: $q = A + (max + 1)$
3: $r = B + (max + 1)$
4: $max = \maxValue(q, r)$
5: $min = \minValue(q, r)$
Assignment phase

After the preprocessing phase has been performed, we can continue with the transformation of all the assignment statements ("C := expr1"), that belong to the if-statement.

Different formulas have been depicted for each relational operator, so that depending of the truth value of the condition C will keep his old value or will receive the new value that expr1 yields. In Table 5.2, we can see each formula that is used for every relational operator.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>if A &gt; B then C := expr1</td>
<td>C := (expr1) * (q/max-min/max) + C * (r/max)</td>
</tr>
<tr>
<td>if A &gt;= B then C := expr1</td>
<td>C := (expr1) * (q/max) + C * (r/max-min/max)</td>
</tr>
<tr>
<td>if A &lt; B then C := expr1</td>
<td>C := (expr1) * (r/max-min/max) + C * (q/max)</td>
</tr>
<tr>
<td>if A &lt;= B then C := expr1</td>
<td>C := (expr1) * (r/max) + C * (q/max-min/max)</td>
</tr>
<tr>
<td>if A == B then C := expr1</td>
<td>C := (expr1) * (min/max) + C * abs(q/max-r/max)</td>
</tr>
<tr>
<td>if A != B then C := expr1</td>
<td>C := (expr1) * abs(q/max-r/max) + C * (min/max)</td>
</tr>
</tbody>
</table>

Table 5.2: Formulas for the different conditions that can be present in an if-statement. If the condition is false C will keep its old value

This replacement is performed for every assignment statement that is present in the then part of the if-statement, certainly using the corresponding expression in the right-hand side of the assignment. These formulas are easy to follow, if the truth value of the conditional expression is true, then the formulas in bold of the “true” column in Table 5.2 will yield a value of one, while formulas in bold of the “false” column will yield a value of zero, meaning that C will receive the value of expression expr1, which is what we were expecting to happen. On the other hand, if the truth value of the conditional expression is false, then the opposite occurs; the formulas in bold of the “true” column in Table 5.2 will yield a value of zero, while formulas in bold of the “false” column will yield a value of one, and C will
Chapter 5. Source Code Analysis and Pattern Identification

Table 5.3: Formulas for the logical ‘and’ (&&) operator and the ‘or’ (||) operator. The ‘|’ represents the bitwise inclusive ‘or’ function.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>if (c_1&amp;&amp;c_2) then (C := expr_1)</td>
<td>(C := (expr_1) \ast (FfT_{c1})(FfF_{c2}) + C \ast (FfF_{c1})(FfF_{c2}))</td>
</tr>
<tr>
<td>if (c_1</td>
<td></td>
</tr>
</tbody>
</table>

keep his old value.

There could be a case when the if-statement has an else part. In that case, we again replace every assignment statement with the formula that corresponds to the opposite relational operator of the condition in the if-statement. For instance, if we have a relational operator ‘\(>\)’ in the condition expression, then we have to replace all the assignment statements, in the else part, with the formula that belongs to the relational operator ‘\(<\leq\)’.

We need to complement the formulas in Table 5.2, so that they can also support the presence of logic operators (e.g. and, or). In this case, we have to deal with more than one condition in an if-statement.

In Table 5.3, we can see the formulas required to transform the assignment statements when multiple conditions are present in an if-statement. The use of these formulas is very simple, we only need to replace the formulas for true \((Fft)\) and formulas for false \((Fff)\) of each condition, with the formulas that are in bold from Table 5.2. A minor inconvenience in these formulas is that, for each condition, we are required to define the variables q, r, min, max that were explained in section 5.1.3.1 (see Algorithm 5.8).

As a small example, we will consider the following if-statement:

\[
\text{if } (a < b \&\& b \geq c) \text{ then } ans := 2 \ast a \ast b
\]

Looking in Table 5.3, we see that we have to use the following formula:
\[ ans = (2 \ast a + b) \ast (F_fT_c1) \ast (F_fT_c2) + ans \ast (F_fF_c1) | (F_fF_c2) \]

Now, we replace the formulas for each condition with the ones explained in Table 5.2. Finally, the transformation is:

\[ ans = 2 \ast a + b \ast (q1 / max1) \ast (r2 / max2) + ans \ast ((r1 / max1 - min1 / max1) | (q2 / max2 - min2 / max2)) \]

**Evaluation: Single-path-if pattern**

In order to demonstrate the advantages of the simple-path-if pattern, I transformed two searching algorithms where the use of relational operators is very common. In the first example, I changed the traditional sequential search algorithm implementation replacing every if-statement with the single-path-if pattern. Both implementations are shown in Algorithms 5.9 and 5.10.

**Algorithm 5.9** Traditional implementation of a sequential search algorithm

**Input:** \( a, myArray[], SZ : integer \)

**Output:** integer

1. \( i = 0 \)
2. \( \textbf{while} \ (i < SZ) \ \textbf{do} \)
3. \( \textbf{if} \ (key == myArray[i]) \ \textbf{then} \)
4. \( \quad \textbf{return} \ i; \)
5. \( \quad \textbf{end if} \)
6. \( i = i + 1 \)
7. \( \textbf{end while} \)
8. \( \textbf{return} \ -1 \)

As a first step, I perform a testability analysis of both implementations.
Testability measurement for Algorithm 5.9

\[ T_1 = \mu(D_2^1(\mu(D_3^2(\mu(P_0^1)))))) \]
\[ T_1 = \mu(D_2^1(\mu(D_3^2(1)))) \]
\[ T_1 = \mu(D_2^1(1 + 1)) = 2 + 1 = 3 \]

Testability measurement for Algorithm 5.10

\[ T_2 = \mu(P_1^1) \cdot \mu(D_2^1(\mu(P_4^1) \cdot \mu(P_5^1) \cdot \mu(P_7^1) \cdot \mu(P_8^1) \cdot \mu(P_9^1)))) \]
\[ T_2 = 1 \cdot \mu(D_2^1(1 \cdot 1 \cdot 1 \cdot 1 \cdot 1)) \]
\[ T_2 = 1 + 1 = 2 \]

The results of testability analysis, shown above, clearly reveal that implementing the single-path-if pattern in a sequential sort algorithm reduces the number of simple paths of the implementation. I tested and measured the execution time of these two implementations with arrays of 5000 elements. There were two measurements for each implementation:

1. Best case, searching an element that is present in the first position of the array.
2. Worst case, searching an element that is not present in the array at all.
Chapter 5. Source Code Analysis and Pattern Identification

**Algorithm 5.10** Transformed implementation of Algorithm 5.9 applying single-path-if pattern

**Input:** a, myArray[], SZ :integer

**Output:** integer

1: \( \text{answer} = -1 \)
2: \textbf{for} \( i = SZ - 1; i >= 0; i + + \) \textbf{do} 
3: \( \text{max} = \text{maxValue}(\text{abs(key)}, \text{abs(myArray}[i]]) \)
4: \( q = \text{key} + (\text{max} + 1) \)
5: \( r = \text{myArray}[i] + (\text{max} + 1) \)
6: \( \text{max} = \text{maxValue}(q, r) \)
7: \( \text{min} = \text{minValue}(q, r) \)
8: \( \text{answer} = (i) \ast (\text{min}/\text{max}) + \text{answer} \ast \text{abs}(q/\text{max} - r/\text{max}) \)
9: \textbf{end for} 
10: \textbf{return} \ \text{answer}

Every algorithm was executed 100 times for each case. The results for these
for measurements are depicted in Figure 5.4 and Figure 5.5.

The charts in Figure 5.4 and Figure 5.5 clearly shows the advantages for the timing behavior of algorithms that the single-path-if pattern provides. The differences between the best and worst execution time in Algorithm 5.5 is very small, still there are some cases when BCET is larger than the WCET. The smaller the difference between BCET and WCET of an algorithm is, the more predictable the algorithm becomes.

As a second example, I decided to replace the constant-time conditional expressions used in the binary search algorithm that is shown in Algorithm 5.2, with the single-path-if pattern (Algorithm 5.11), so that this new algorithm can be implemented in any hardware architecture, and not as the constant-time conditional expression pattern in section 5.1.1.

Again, I start with a testability analysis of the implementation of Algorithm 5.11 in order to compare the results obtained in section 5.1.1.
Chapter 5. Source Code Analysis and Pattern Identification

Algorithm 5.11 Binary search implementation using single-path-if pattern

**Input:** $a, myArray[], SZ : integer

**Output:** integer

1: $left = 0$
2: $right = SZ - 1$
3: $idx = (left + right) >> 1$
4: for $inc = SZ; inc > 0; inc = inc >> 1$ do
5:  $max = maxValue(abs(key), abs(myArray[idx]))$
6:  $q = key + (max + 1)$
7:  $r = myArray[idx] + (max + 1)$
8:  $max = maxValue(q, r)$
9:  $min = minValue(q, r)$
10: $left = (idx + 1) * (q/max - min/max) + (left) * (r/max)$
11: $right = (idx - 1) * (r/max - min/max) + (right) * (r/max)$
12: $idx = (left + right) >> 1$
13: end for
14: $idx = (idx) * (min/max) + (−1) * abs(q/max − r/max)$
15: return $idx$

- Testability measurement for Algorithm 5.11

$$T_1 = \mu(P_1^1) \cdot \mu(P_1^2) \cdot \mu(P_1^3) \cdot \mu(D_2^3(\mu(P_1^5) \cdot \mu(P_1^7) \cdot \mu(P_1^8)) \cdot \mu(P_1^9) \cdot \mu(P_1^{10}) \cdot \mu(P_1^{11}) \cdot \mu(P_1^{12})) \cdot \mu(P_1^{14}) \cdot \mu(P_1^{15})$$

$$T_1 = 1 \cdot 1 \cdot 1 \cdot \mu(D_2^3(1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1 \cdot 1)) \cdot 1 \cdot 1$$

$$T_1 = \mu(D_2^3(1)) = 1 + 1 = 2$$

The testability results are the same as those in section 5.1.1, when we use the Puchner’s pattern. As a second step, I executed 100 times and measured the execution time of the two binary search implementations (Algorithms 5.1 and 5.11) with arrays of 5000 elements. There were two measurements for each implementation:

1. Best case, searching an element that is present in the $(SZ − 1)/2$ position
of the array.

2. Worst case, searching an element that is not present in the array at all.

![Binary Search 1](image)

Figure 5.6: Measurements of the execution times of the original implementations of a binary search algorithm from Algorithm 5.1

The results I obtained after running each implementation (see Figure 5.6 and Figure 5.7) are not the ones I expected. Before measuring each execution time, I expected that differences between the BCET and WCET for the traditional binary search algorithm were going to show a clearer gap between these two execution times, and not as overlapped as Figure 5.6 shows. However, applying the single-path-if pattern in the binary search algorithm, improves its time predictability, because the gap between the BCET and WCET of the traditional implementation (1500-2500 ns) is smaller than the one for the algorithms using the single-path-if pattern (160-2150 ns).
5.2 Summary of experiments and discussion

In this chapter I have presented different patterns that were created to improve the timing analyzability of a real-time systems in order to reduce the effort required when we do dynamic timing analysis in such systems. As a basis, I presented the constant-time conditional expression that was presented by Puschner [29], and showed that is possible to find patterns in the source code that help us to improve timing analysis, although his work was oriented to object code, which needs special hardware and possibly different compilers to fully take advantage of this pattern. On the other side, my thesis focused on introducing patterns (i.e. programming conventions) that can reduce the number of simple paths in a program’s source code.

The patterns I have shown reduce the number of simple paths in the source code by replacing if-statements with calculations and formulas, so that we can reduce the gap between BCET and WCET as a sign of timing analyzability im-
Table 5.4: Summary of the results of all the experiments carried out in the previous section

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Testability Metric</th>
<th>BCET</th>
<th>WCET</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>Traditional</td>
<td>4</td>
<td>20196513</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>3</td>
<td>238376778</td>
</tr>
<tr>
<td>Sequential Search</td>
<td>Traditional</td>
<td>3</td>
<td>477</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>2</td>
<td>411705</td>
</tr>
<tr>
<td>Binary Search</td>
<td>Traditional</td>
<td>8</td>
<td>632</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>2</td>
<td>2736</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>Traditional</td>
<td>3</td>
<td>78183</td>
</tr>
<tr>
<td></td>
<td>Transformed</td>
<td>2</td>
<td>1783337</td>
</tr>
</tbody>
</table>

Reducing the number of simple paths is directly related to the transformation of if-statements in single-path statements using calculation. However, another control flows that contribute to increase the number of paths are loops (e.g. do-while). Unfortunately, it is impossible to transform these loops in single-path statements, because of the need to execute some processes many times. On this issue, the only solution so far is to work with bounded loops, as the ones presented in section 3.1.1.

The replacement of if-statements is not always easy for the patterns in this thesis. Even more, in some cases it is impossible. For instance, the example that we can see in Algorithm 5.12 is very hard to transform using the single-path-if pattern. The presence of an inner loop (Line 5) makes the replacement of the if-statement (Line 1) a little bit more complex, although the transformation is still possible as we can see in Algorithm 5.14. Lines 1, 2 an 3 of Algorithm 5.12 can
be easily transformed (Lines 6, 7 and 8 of Algorithm 5.14). However, the \textit{while} loop needs a special treatment. As we can see in Line 8 of Algorithm 5.14, I have added the condition expression of the if-statement in Algorithm 5.12, to the condition of the \textit{while} loop.

In Algorithm 5.13, the difficult emerges when we need to call a routine (Lines 2, 4) that depends on the truth value of the condition of the if-statement (Line 1). In this case the transformation using the single-path-if pattern is impossible because there cannot be an assignment to the call to a routine.

\textbf{Algorithm 5.12} If-statement example that is very hard to transform due the presence of a inner loop
\begin{verbatim}
1: if \((a > b)\) then
2: \quad \(b = 3 \times b - a\)
3: \quad \(r = 5 \times \text{altitude}\)
4: \quad \(x = TT - \text{altitude}\)
5: \quad \textbf{while} \((r < 7000)\) \textbf{do}
6: \quad \quad \(x = TT - \text{altitude}\)
7: \quad \quad \(r = r - 500\)
8: \quad \textbf{end while}
9: \textbf{end if}
\end{verbatim}

\textbf{Algorithm 5.13} If-statement example that is impossible to transform due the presence of function calls
\begin{verbatim}
1: if \((3 \times b == \text{maxV})\) then
2: \quad \textit{callFunctionX} \((b, \text{pIni}, \text{pFin})\)
3: \textbf{else}
4: \quad \textit{callFunctionY} \((a, \text{pIni}, \text{pFin})\)
5: \textbf{end if}
\end{verbatim}

In other implementations, it is also very hard to reduce the number of simple paths, for instance when an algorithm only has loops. Therefore, in these cases, it is impossible to apply the patterns presented in this thesis, or to find new ones that can increase its analyzability and its timing predictability by reducing the number of paths in the source code. However, these implementations can still be
Algorithm 5.14 Transformation of Algorithm 5.12 using the single-path-if pattern
1: max = maxValue(abs(a), abs(b)) \{Preprocessing Phase Starts\}
2: q = a + (max + 1)
3: r1 = b + (max + 1)
4: max = maxValue(q, r1)
5: min = minValue(q, r1)
6: b = (3 * b - a) * (q/max - min/max) + (b) * (r/max) \{Assignment Phase Starts\}
7: r = (5 * altitude) * (q/max - min/max) + (r) * (r/max)
8: x = (TT - altitude) * (q/max - min/max) + (x) * (r/max)
9: while (r < 7000 && a > b) do
10:   x = TT - altitude
11:   r = r - 500
12: end while

transformed so that they can be more suitable for real-time systems. For instance, we can produce code that is free from input-data dependent control flow. This approach is called “WCET-Oriented programming” [26].

The insertion sort algorithm is a very good example that shows how we can increase its timing analyzability without reducing the number of simple paths and eliminating some input-data dependent control flow. As we can see in Algorithm 5.15, this implementation has only loops and no decision statements (i.e. if-statements). Nevertheless, the expression myArray[j - 1] > tmp in the inner loop makes this loop an input-data dependent control flow, and eliminating this expression will guarantee a better analyzable algorithm (Algorithm 5.16), just as the results will show later.

As a first step, I perform a testability analysis of both implementations.
Algorithm 5.15 Traditional implementations of an insertion sort

**Input:** $myArray[]$, $SZ$ :integer  
**Output:** sorted list

1. $for (i = 1; i < SZ; i++)$ do
2. $tmp = myArray[i]$
3. $for (j = i; j > 0 && myArray[j - i]; j--)$ do
4. $myArray[j] = myArray[j - 1]$
5. $end for$
6. $myArray[j] = tmp$
7. $end for$

Algorithm 5.16 Insertion sort algorithm that is free from input-data dependent control flow

**Input:** $myArray[]$, $SZ$ :integer  
**Output:** sorted list

1. $for (i = 0; i < SZ; i++)$ do
2. $for (j = i + 1; j > 0; j--)$ do
3. $min = minValue(myArray[j - 1], myArray[j])$
4. $max = maxValue(myArray[j - 1], myArray[j])$
5. $myArray[j] = max$
6. $myArray[j - 1] = min$
7. $end for$
8. $myArray[j] = tmp$
9. $end for$
Chapter 5. Source Code Analysis and Pattern Identification

- Testability measurement for Algorithm 5.15

\[ T_1 = \mu(D_1^2(\mu(P_2^1) \cdot \mu(D_2^3(\mu(P_4^1))) \cdot \mu(P_6^1))) \]
\[ T_1 = \mu(D_1^2(1 \cdot 2 \cdot 1)) \]
\[ T_1 = 2 + 1 = 3 \]

- Testability measurement for Algorithm 5.16

\[ T_2 = \mu(D_1^2(\mu(D_2^3(\mu(P_3^1) \cdot \mu(P_4^1) \cdot \mu(P_5^1) \cdot \mu(P_6^1))) \cdot \mu(P_8^1))) \]
\[ T_2 = \mu(D_1^2(\mu(D_2^3(1 \cdot 1 \cdot 1 \cdot 1)))) \]
\[ T_2 = \mu(D_1^2(1 + 1)) = 2 + 1 = 3 \]

We can see from the results of the testability analysis, that both implementations (Algorithms 5.15 and 5.16) have the same testability metric (3 simple paths). The key that makes Algorithm 5.16 more analyzable is to remove the expression `myArray[j - 1] > tmp` in the inner for-statement of Algorithm 5.15, and replace it with the min/maxValue pattern, as shown in Algorithm 5.16. Both implementations were tested to sort an array of 500 elements and applied a dynamic timing analysis, all the measurements that I gathered form this experiment are depict in Figure 5.8 and Figure 5.9.

Figure 5.8 and Figure 5.9, clearly show the advantage of avoiding input-data dependent control decision in the source code. The Algorithm 5.15 presents some input-data dependent control flow, yields a wider range of execution times, which certainly is not what we really need from an implementation in a real-time environment. On the other hand, applying a pattern that increases the algorithm’s testability and removing input-data dependent control in the implementation (see Algorithm 5.16), we can reduce the gap between the BCET and WCET of Algo-
Figure 5.8: Measurements of the execution time, in nanosecond, of Algorithm 5.15.

This example shows how different techniques can contribute to increase the timing analyzability of different algorithms, and at the same time reducing the gap between the BCET and WCET of such implementations. This behavior is depicted in Figure 2.2, where, on one side, we can avoid input-data dependent control decision in the source code, and on the other side, we introduce new analysis techniques. Combining both, we can improve the timing analyzability of real-time systems.

The last point that is important to mention is the big increase in the execution time for all the algorithms where the patterns were implemented (see Table 5.4). These results are very understandable, because of the introduction of a significant number of calculations to the algorithms. It is possible that these calculations, especially multiplications and divisions, can make timing analysis more diverse and difficult at the machine-code level, because some processors may have different execution times according to input parameters and compiler optimization to
Chapter 5. Source Code Analysis and Pattern Identification

Figure 5.9: Measurements of the execution time, in nanosecond, of Algorithm 5.16

those operations. This may lead to further research of the patterns of this thesis at a lower level (i.e. hardware analysis), and analyze whether these extra calculations impact negatively on the hardware-level execution time analysis or not.
Chapter 6

Summary, Conclusions and Future Work

The main motivation in this thesis was to reduce the effort developers need to invest on performing timing analysis in real-time systems by improving the timing analyzability.

Furthermore, I have also mentioned that if we are able to reduce the gap between the WCET and BCET of a system, then we can say that we have improved the timing analyzability in a real-time system; approaching in this way to the ideal scenario in real-time systems, where we can achieve a constant execution time of all tasks. Simultaneously, by reducing this gap, we will be increasing the time predictability of the system.

We have seen the resemblance between dynamic timing analysis and testing, and how testability analysis can help us to predict the effort needed in timing analysis. This is why, I stated that the number of simple paths that exist in the source code is a good testability metric that can help us to estimate the effort we require to perform a timing analysis in a piece of code. As a consequence, we can say that more paths in a program increase the effort we will need to carry out a
Chapter 6. Summary, Conclusions and Future Work

timing analysis for that program.

In order to assess the effort in timing analysis, I used a testability metric for simple path testing [24], because the number of paths in the source code can be infinite, due to the possible presence of loops [24] and a metric for path testing is impossible to achieve [2, 24]. This metric gives a useful estimate of the testing effort and helped me to find the patterns presented in this thesis.

Based on “the single path approach” presented by Puschner [29], but in my case oriented to source code level, I tried to find patterns in the source code with high testability (i.e. low number of simple paths), which by applying them many times in different implementations can help to reduce the number of simple paths in the source code, and therefore reduce the effort in timing analysis.

After I identified some patterns, I implemented them in different algorithms, compare the execution times between the traditional implementation and the transformed implementations, and check if the gap between BCET and WCET of the algorithms using the patterns, is smaller than the gap of the traditional implementations. Experiments with these patterns have shown the following results:

- All the found patterns decrease the number of simple paths in the source code by replacing if-statements with different formulas. Another way to reduce the number of paths is to replace loops. Unfortunately, this last option was impossible to achieve, because some algorithms present processes that have to repeat several times and they cannot be reduced to a simple path.

- All the algorithms where I implemented the identified patterns have produced good results in their execution times. The variation of execution of these algorithms was reduced significantly in many cases.

- In order to improve the analyzability of the chosen algorithms for the experiments, we have to pay the price of decreased performance of the algo-
rithms. For example, sorting 500 hundred algorithms using the traditional bubble sort algorithm had an average execution time of 20196513ns. However, after implementing the patterns in the algorithm its average execution time rises up to 238376778ns.

- There is another problem when we try to use the patterns in different algorithms, readability. Unfortunately, after changing an algorithm with the patterns, we downgrade its readability. Since all patterns that I have found help us to replace if-statements, we cannot longer easily distinguish the statements that belonged to the true part or to the false part in the if-statement, which of course is a very hard problem when we need to maintain that piece of code.

- The combination of this technique, using patterns with high testability (e.g. low number of simple paths), with other techniques, such as Worst-Case Oriented Programming and the Single-Path Approach is highly recommendable. As we have seen in section 5.2, there are cases when it is impossible to reduce more the number of simple paths in the code, in this case we can introduce the other techniques in order to keep increasing the analyzability, and, therefore, predictability of the system.

The results I obtained from the experiments show that testability measures are a good technique to estimate the timing analysis effort. The experiments have shown that finding patterns in the source code have helped to reduce the number of simple paths of the algorithms that implement them, and therefore, they reduce their gap between BCET and WCET; which is a sign that time analyzability is improved. Finally, we can state that the more number of paths a source code of an algorithm has, the less analyzable with respect to time the algorithm is expected to become.

80
The problem with the algorithms’ readability affected by the implementation of the patterns is a really big issue. As a small example, in Algorithm 6.2 we can see the transformation using the single-path-if pattern of the piece of code in Algorithm 6.1. At a first glance, and without looking at the code in Algorithm 6.1, is it possible to distinguish which assignments statements belong to the ‘if’ part or to the ‘else’ part of if-statement? Is it easy to maintain codes that are similar to the one presented in Algorithm 6.2? I think that the answer to both questions is NO, and it is very probable that the effort of timing analysis that we could have saved with the use of these patterns, will be transformed into more effort that we could need to maintain this kind of code.

Algorithm 6.1 A piece of code using an if-statement

1: if \((a > 10)\) then
2: \(x = x * 30\)
3: \(y = (r - 50) * x\)
4: \(z = a + 95/y\)
5: else
6: \(x = x - 30\)
7: \(j = j + 1\)
8: \(u = y - z + 42\)
9: end if

The easiest solution is let the developers deal with it by themselves, for example by using more comments in their source code. But, again we will be moving the problem to coding time which certainly will be increased.

A good solution could be, to develop a “translator” that can analyze the code from the developers using their usual programming conventions (e.g. Algorithm 6.1) and “translate” all the parts of code that fit to the patterns that were identified in this thesis, and other new patterns that later can be created.

This translator will be transparent to the developer and it will lie between the developer and the compiler. The developer can continue coding in the style he
Algorithm 6.2 Transformation of Algorithm 6.1 using the single-path-if pattern showing the worsening of its readability

1: \( \textit{max} = \textit{maxValue}(\textit{abs}(a), \textit{abs}(10)) \) \{Preprocessing Phase Starts\}
2: \( q = a + (\textit{max} + 1) \)
3: \( r = 10 + (\textit{max} + 1) \)
4: \( \textit{max} = \textit{maxValue}(q, r) \)
5: \( \textit{min} = \textit{minValue}(q, r) \)
6: \( x = (x \ast 30) \ast (q/\textit{max} - \textit{min}/\textit{max}) + (x - 30) \ast (r/\textit{max}) \) \{Assignment Phase Starts\}
7: \( y = ((r - 50) \ast x) \ast (q/\textit{max} - \textit{min}/\textit{max}) + (y) \ast (r/\textit{max}) \)
8: \( z = (a + 95/y) \ast (q/\textit{max} - \textit{min}/\textit{max}) + (z) \ast (r/\textit{max}) \)
9: \( j = (j) \ast (q/\textit{max} - \textit{min}/\textit{max}) + (j + 1) \ast (r/\textit{max}) \)
10: \( u = (y) \ast (q/\textit{max} - \textit{min}/\textit{max}) + (y - z + 42) \ast (r/\textit{max}) \)

prefers the most. Once his coding is done, and before compiling it, the translator will take the developer’s code, analyze it and transform it using all the patterns I have found and others that can be found later. In this way, the patterns do not affect the program’s readability and we still have an improved time analyzability and high time predictable source code for our real-time systems.
Bibliography


