Graduate School ETD Form 9
(Revised 12/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Scott Edward McNeany

Entitled
Characterizing Software Components Using Evolutionary Testing and Path-Guided Analysis

For the degree of Master of Science

Is approved by the final examining committee:

Dr. James Hill
Chair

Dr. Rajeev Raje

Dr. Mohammad Hasan

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): Dr. James Hill

Approved by: Dr. Shiaofen Fang 03/21/2013
Head of the Graduate Program Date

CHARACTERIZING SOFTWARE COMPONENTS USING EVOLUTIONARY TESTING AND PATH-GUIDED ANALYSIS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Scott Edward McNeany

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science

May 2013

Purdue University

Indianapolis, Indiana

This work is dedicated to my loving and patient wife, Terri.

ACKNOWLEDGMENTS

I am sincerely thankful to my thesis advisor, Dr. James Hill, for making me work hard and strive to reach my full potential. Your guidance and encouragement have been invaluable.

I also want to thank Dr.Rajeev Raje and Dr.Mohammad Hasan for being a part of my thesis committee and contributing to this work.

Thank you to my wife, Terri, and my entire family for your continued support.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

McNeany, Scott Edward M.S., Purdue University, May 2013. Characterizing Software Components Using Evolutionary Testing and Path-Guided Analysis. Major Professor: James H. Hill.

Evolutionary testing (ET) techniques (*e.g.*, mutation, crossover, and natural selection) have been applied successfully to many areas of software engineering, such as error/fault identification, data mining, and software cost estimation. Previous research has also applied ET techniques to performance testing. Its application to performance testing, however, only goes as far as finding the best and worst case execution times. Although such performance testing is beneficial, it provides little insight into performance characteristics of complex functions with multiple branches.

This thesis therefore provides two contributions towards performance testing of software systems. First, this thesis demonstrates how ET and genetic algorithms (GAs), which are search heuristic mechanisms for solving optimization problems using mutation, crossover, and natural selection, can be combined with a constraint solver to target specific paths in the software. Secondly, this thesis demonstrates how such an approach can identify local minima and maxima execution times, which can provide a more detailed characterization of software performance. The results from applying our approach to example software applications show that it is able to characterize different execution paths in relatively short amounts of time. This thesis also examines a modified exhaustive approach which can be plugged in when the constraint solver cannot properly provide the information needed to target specific paths.

1 INTRODUCTION

Performance testing [1] is an important aspect of testing any software system. Through performance testing, software system stakeholders learn how the system performs under different operating conditions, such as peak time vs. non-peak time. Likewise, performance testing can be used to characterize the behavior of a software system. For example, performance testing can be used to identify best and worst-case execution times of a software system.

When executing a performance test, is it critical that software testers select good input values for their tests. This is because different test input values will produce different performance results. For example, evolutionary testing (ET) [2], which is a concept of software testing that allows new test cases to be derived from existing test cases without human intervention, and genetic algorithms (GAs) [3], which are specific algorithms for carrying out evolutionary testing, have been used to generate input values for performance testing of software systems. In such cases, ET has been primarily used to characterize best-case and worst-case execution times of a software system (*i.e.*, high-level, global performance properties of a software system) [4].

Although it is important to characterize systemic performance properties of a software system, is also important to characterize local performance properties of a software system. For example, software systems usually contain many control branches and loops. Each control branch and loop will exhibit different performance properties, which is typically reachable by only a specific set of input values [5]. In order to truly characterize the performance of a software system, it is necessary to understand both global and local performance properties.

Unfortunately, it can be both tedious and time-consuming to evaluate both global and local performance properties—especially local performance properties of complex software systems. This thesis therefore presents an approach for addressing this

challenging problem. More specifically, this thesis presents an approach called Path-guided, Parameterized Performance Testing (PPPT) that combines ET and GAs with path constraint-logic to characterize local performance properties of a software system.

PPPT operates by analyzing the branch and loop conditions of the software system to determine the constraints necessary to target a specific path in a software function. Once the constraints that target the specific control path are known, the ET portion of PPPT generates a suite, or initial population, of test cases. These test cases are then run against the target software component, and the set of input parameters resulting in the worst (or best, if that's what is being tested) performance is used to generate the next population of test cases. This process continues—with each round getting closer to the worst-case performance of a specific path —until PPPT is confident it has found the parameters necessary to generate the worst case for that branch. Once each branch is completed, the next branch is analyzed in the same fashion until all branches are complete.

There are, however, cases where a modern constraint solver is not capable of providing input values that target a specific path. In such cases, PPPT uses a modified version of an exhaustive approach that instruments source code to help target specific paths. The software is modified in two ways: first, the source code is instrumented with counters to track the path taken during each iteration of the input variables; and second, the source code is cleansed of any computationally-intensive or out of process call that are not critical in determining the path. For example, this may be an out of process call to the database or a web service, or a system call to the operating system. By removing these expensive calls, we can exhaustively search the input parameter space without executing the core logic of the application—thereby reducing the overall execution time of each test.

The main contributions of this thesis therefore are as follows:

- It presents a novel approach called Path-guided Parameterized Performance Testing (PPPT) that allows for performance analysis without specifying exact

inputs, and targets specific branches of code to provide information about local minima and maxima execution times;

- It illustrates how PPPT allows for rapid analysis, modeling, and comparison of a software system's performance characteristics; and
- It discusses how PPPT was applied to several challenge problems, which highlighted the limitations of existing tool sets (*e.g.*, modern constraint solvers for targeting specific paths) and offers an alternative method that uses a modified exhaustive approach for building the necessary input parameter-path mapping.

Results from applying PPPT to sample problems show that PPPT can be highly effective in analyzing branch performance. In one case demonstrated in the results section, PPPT resulted in 78% fewer iterations over traditional exhaustive testing. Likewise, PPPT can successfully separate the conditions necessary to target a specific branch and—using existing ET methodologies—determine the best and worst case execution times of each branch.

1.1 Thesis Organization

The remainder of this thesis is organized as follows: Chapter 2 discusses how PPPT relates to other existing works; Chapter 3 provides background information needed to understand PPPT's solution approach; Chapter 4 discusses our approach; Chapter 5 presents the results of applying our approach to several software systems; and Chapter 6 provides concluding remarks and future research directions.

2 RELATED WORKS

This chapter discusses existing work that relates to our work on PPPT. More specifically, this chapter covers related works from the area of genetic algorithms, input test data generation, and path-guided exploration.

2.1 Genetic Algorithms

The first application of GA on performance analysis can be credited to Wegener et al. [3]. Wegener's research focused solely on locating the minimum and maximum execution times. Their fitness function determined the "best fit" candidates by analyzing the execution time of the previous test runs and taking the best or worst execution time, depending on the goal of that particular test. Based on a simple C-function sample application, Wegener was able to find the worst case execution time in just 20 generations compared to 4603 generations using random testing. Even more impressive, Wegener found a better best case execution time than was found using random testing.

Wegener's results demonstrate that GAs are a more suitable optimization strategy than hill-climbing [6], which is a searching technique that takes incremental steps to compare two input parameter sets. The approach alters a single parameter at a time and if the result is closer to the optimum, the new test is used to generate additional tests. The process is repeated until no further optimizations can be made. Genetic algorithms are more suitable because a large population is used in GA to derive new inputs when compared to using hill climbing.

Although Wegener did show branch coverage statistics, their work only offered the recommendation for adding structural capability by stating that "Another idea for further improvement is to combine our approach with structural testing. The

fitness function could be expanded in such a way that individuals that execute a new program branch get a high fitness value to ensure their survival in the next generation. The diversity of the population therefore is not only maintained with respect to the temporal behaviour of individuals, but also in consideration of the test object's internal structure." [3]

More recent applications of genetic algorithms include Oyama's Real-coded Adaptive Range Genetic Algorithm (ARGA) [7], which was a parallel genetic algorithm for three-dimensional aerodynamic shape optimization, which ultimately lead to better wing design. A GA was also used to study direct pattern synthesis and impedance matching for the Juno radiometer antennas [8]. In 2010, Kim et al. [9] made use of a previously developed adaptive hybrid genetic algorithm search simulator (to solve the resource-constrained project scheduling problem in the area of civil and construction management.

2.2 Test Data Generation

Korel et al. [10] provides both a comprehensive overview of software test data generation techniques. Korel also introduces a new technique for dynamic test data generation. This approach relies on data flow analysis, which allows the execution path of the program under test to be monitored, to build the constraints necessary to target a specific path. While executing the application, if an undesired path constraint condition is reached, the path condition can be flipped on the next execution. The dynamic approach has limitations in determining path infeasibility and will result in a large number of attempts before determining that the path cannot be reached. The author states, however, that this is not an issue with standard symbolic execution [11] (*i.e.*, the process of replacing actual variables with symbols thereby allowing their path constraints to be recorded) This is why Korel proposes a combination of the two methods.

2.3 Combining Instrumentation and Genetic Algorithms

Maragathavalli et al. [12] uses an approach similar to Korel's approach for identifying bugs in software systems called a Path-Reuse Method (PRM). The PRM approach differs slightly from Korel's approach in that it executes the target software prior to determining path constraints. PRM then use the path result, which is determined by instrumenting the source code, as the fitness function for determining the next generation of inputs. This approach is viable for identifying bugs because it provides high branch coverage. It, however, is infeasible in characterizing the performance of software because it is not guaranteed to find every parameter combination that targets a specific branch. It only guarantees that *some* set of inputs will be found that target a specific branch.

3 BACKGROUND

This section is meant to introduce concepts that are key in understanding and implementing PPPT. We will walk through the process behind evolutionary testing which is one of the core components in PPPT. We will also discuss how evolutionary testing can be combined with other well-known software methods, such as constraint solvers and software code instrumentation, to target specific paths.

3.1 Evolutionary Testing

Evolutionary Testing (ET) [2], also known as GAs as introduced in Section 1, is a concept of software testing that allows new tests to be derived from existing tests based on a fitness function. The fitness function used in an ET represents a goal of the software, such as worst case execution time. In the first round of input parameter selection, a random sample of test cases is chosen N times where N represents the initial population size. The test cases are all run independently against the target function and the result is then run through the fitness function. Lastly, the test case with the best "fitness" for the given test is chosen to survive to the next round and produce offspring.

Each subsequent round in ET begins with the generation of new tests that are closely related to the "best" fit test case from the previous round with some degree of separation. The degree of separation is random within a specified proximity range. For example, assume there is one integer variable with the range (0, 100) and has a proximity value of the offspring 0.10. If the initial best fit test case has a value $X = 45$, then all derived offspring will have a random value of 45 ± 10.

Once enough offspring is generated to fill the population size for the next round of testing, then the new round of tests are run against the target function and an-

alyzed in the same manner as the previous test run. This process repeats until the desired goal is reached, or the maximum number of rounds set by the user is reached. Lastly, if ET gets "stuck" in a local minima or maxima while trying to reach a global minimum/maximum, a new set of completely random variables can be chosen. There is, however, no way to automatically detect that a local minima or maxima has been reached, so this random regeneration may result in a false positive. For the purpose of this research, we've chosen not to do a random regeneration at any point because the function is already is already split into the various paths, which is most likely what would cause a local minima/maxima performance.

3.2 Path-Guided Testing

Branch coverage [13], which is a test method that aims to ensure each possible branch from each decision point is executed at least once, thus ensuring that all reachable code is executed, is an important aspect a good testing library. This is because it is "hard", if not impossible, to determine if a bug exists in different parts (*i.e.*, branches) of the software when it is not covered by a test. Uncovering bugs is just one instance of why branch coverage is important.

To further demonstrate the concept of branch coverage, we'll use a common problem called the Triangle Problem. As shown in Figure 3.1, there are three branches that determine the type of a triangle (*i.e.*, equilateral, isosceles, or scalene) based on a comparison of the sides a, b, and c. A triangle with three equal sides is considered equilateral; a triangle with two equal sides is isosceles, and a triangle with no equal sides is scalene.

We then write two unit test functions, both shown in Figure 3.2. The function `TestEquilateral` validates that a triangle with three equal sides is classified as "Equilateral". The function `TestIsosceles` validates that a triangle with two equal sides is classified as "Isosceles". There is currently no test that demonstrates the proper classification of the scalene triangle.

```
public string TypeOfTriangle(int a, int b, int c)
{
  if (a == b
    && b == c)  //All sides are equal
  {
    return "Equilateral";
  }
  else if (a == b
    || a == c
    || b == c)  //Two sides are equal
  {
    return "Isosceles";
  }
  else            //No sides are equal
  {
    return "Scalene";
  }
}
```

Figure 3.1.: Triangle Problem

```
[TestMethod]
public void TestEquilateral()
{
  int a = 5;
  int b = 5;
  int c = 5;

  string result = TypeOfTriangle(a, b, c);
  Assert.AreEqual("Equilateral", result);
}

[TestMethod]
public void TestIsosceles()
{
  int a = 5;
  int b = 5;
  int c = 10;

  string result = TypeOfTriangle(a, b, c);
  Assert.AreEqual("Isosceles", result);
}
```

Figure 3.2.: Triangle Problem Unit Tests

In this example, there are three branches that could occur based on the input to the function, but only two of these branches are tested in our test suite. We therefore state that the "branch coverage" of this function as 66% (*i.e.*, 2 out of 3). If we were to write an additional test where a, b, and c were all different, the function would return "Scalene" and our branch coverage would become 100%.

In this thesis, the goal of covering all branches is to learn more about the performance characteristics of all branches individually. Previous performance testing

research [3] [14] puts most of its emphasis on finding the global minimum and maximum regardless of the path. It, however, is relevant to find the local minima and maxima by branch if the goal is to compare the branches, or report the data to the client based on the actual execution path. It is also very meaningful to characterize the performance of the system in terms of the execution path.

For example, this research looks at an example of the RSA cryptosystem where the keys are either reused, or new keys are generated prior to performing the encryption. This decision determines the path that is taken and, as will be shown, has a significant impact on the performance of the system. Without path-specific information, this comparison would require two separate tests. If a specific path can be targeted, then a single test can be written and the paths can be determined at runtime, which greatly reduced testing time. The constraint solver framework discussed in the next section is an integral part of being able to target specific paths for certain types of functions.

3.3 Constraint Solvers

A constraint solver [15] is a mathematical tool for determining the correct inputs that solve a given problem. Many applications exist for which constraint solvers are used, such as real-time supply-chain optimization [16–18], scheduling and resource assignment [19–23], graphics and modeling [24–26], machine learning [27, 28], and decision optimization [29–33]. Our research on PPPT takes advantage of using constraint solvers for decision optimization.

The Microsoft Constraint Solver Foundation (MCSF) [34] is the constraint solver chosen we selected when implementing PPPT. We selected MCSF because it integrates well with our existing C# code and because it has a rich high-level API. Our work on PPPT, however, is not limited to MCSF and C# applications. The MCSF provides a rich API that can be used by any of the .NET programming language (*i.e.*, C#, C++, Visual Basic, F#, and IronPython).

MCSF operates be taking an input string that contains the input parameters, path constraints, and goals the user wishes to have solved. For example, Figure 3.3 illustrates some simple code that contains multiple branches and each branch sleeps for a different period of time. This is called the *Sleep Test*, and we use it throughout this thesis.

```
Expression<Func<int, int, string>> e = (x, y) =>
  x == 0 ? "Invalid x Value"
  : x == 1 ? Sleep(y, "Linear")
  : x == 2 ? Sleep(y * 2, "Linear times 2")
  : x == 3 ? Sleep(y * y, "Quadratic")
  : "Invalid x Value";
```

Figure 3.3.: Sleep Test

Figure 3.4 shows four strings for a single path in the Sleep Test written as a string accepted by the MCSF. That path has a single path constraint of $x == 1$, and individual goals of maximizing x, minimizing x, maximizing y, and minimizing y.

```
Model[
  Decisions[Integers[0,100],x],
  Decisions[Integers[0,100],y],
  Constraints[x == 1],
  Goals[Minimize[x]]]

Model[
  Decisions[Integers[0,100],x],
  Decisions[Integers[0,100],y],
  Constraints[x == 1],
  Goals[Maximize[x]]]

Model[
  Decisions[Integers[0,100],x],
  Decisions[Integers[0,100],y],
  Constraints[x == 1],
  Goals[Minimize[y]]]

Model[
  Decisions[Integers[0,100],x],
  Decisions[Integers[0,100],y],
  Constraints[x == 1],
  Goals[Maximize[y]]]
```

Figure 3.4.: Sleep Test Constraint Strings

The MCSF parses the string above and determines if the goal can be met with the given constraints. If the goal can be met, it returns another string to the user

specifying what constraints must be added to target the specific path. For example, Figure 3.5 shows the results for the four goals in the previous Figure 3.4.

```
1 Minimize: x = 1
2 Maximize: x = 1
3 Minimize: y = 0
4 Maximize: y = 100
```

Figure 3.5.: Sleep Test Constraint Solver Results

As shown in the example above, the constraint solver returned four strings specifying that to reach this path, x must be exactly 1 and y must range from (0, 100). The reason y ranges from (0, 100) is because the original input to the constraint solver specified that it should be within those bounds. The range (0, 100) is arbitrary and can easily be plugged in as -2147483648 (*i.e.*, Int32.MinValue in C#) and 2147483647 (*i.e.*, Int32.MinValue in C#) to get the full range of integers. It, however, should be noted that passing in extremely large values for the decision fields causes poor performance in the constraint solver. This is because the solver must iterate through all possible values within the range. It is therefore important to determine whether the a goal can be solved without passing a large range of values to the constraint solver.

3.4 Source Code Instrumentation

Source code instrumentation [35] is a common practice in software systems for tracing and performance analysis. This practice usually involves instrumenting production systems to find bugs in actively running software. This thesis, however, does not require instrumentation of production systems, and instead uses instrumentation solely for the purpose of creating an input parameter-path map that will be used by the evolutionary component for targeting specific branches. An example of source code instrumentation can be seen in Figure 3.6. This example shows a very basic implementation in which a logging function is called in between each statement. Normally, the logging mechanism would take in additional information like the component

name, class name, function name, line number, and any variety of other information that could be useful to view.

```
1  public string TypeOfTriangle(int a, int b, int c)
2  {
3    if (a == b
4      && b == c)  //All sides are equal
5    {
6      InstrumentationLogger.Log("Got to line 6");
7      return "Equilateral";
8    }
9    else if (a == b
10     || a == c
11     || b == c)  //Two sides are equal
12   {
13     InstrumentationLogger.Log("Got to line 13");
14     return "Isosceles";
15   }
16   else            //No sides are equal
17   {
18     InstrumentationLogger.Log("Got to line 18");
19     return "Scalene";
20   }
21 }
```

Figure 3.6.: Instrumented Triangle Problem

4 THE DESIGN AND FUNCTIONALITY OF PPPT

This chapter explains the design and functionality of PPPT, which characterizes software components and provides a detailed overview of each software path's performance. There are several components that work together to accomplish these goals, which are discussed in detail within this chapter. More specifically, Section 4.1 provides a high-level summary of the steps required to implement PPPT. The intent of this section is to provide the reader with the design constructs independent of the specific implementation. Section 4.2 then describes in detail the process flow and decisions necessary for implementing PPPT. Lastly, Section 4.3 provides a simple example to help illustrate the process of the various components of PPPT.

4.1 Approach

The design of PPPT works as follows: first, we must choose a mechanism for constructing an input parameter-path map. The input parameter-path map is a table of input parameter values that target a specific path. The input parameter-path map is necessary for providing the genetic algorithm component a list of input parameters that target a specific path. This way, the algorithm can simply pick a set of inputs from the list and guarantee that the execution will follow the desired program path. Figure 4.1 shows a sample input parameter-path map for a specific path of an imaginary software component. It shows the values for input parameters x and y that satisfy the path constraints of path $1 - 2 - 3 - 4$.

Once PPPT knows which input parameter values are able to target specific paths, it can then use this data to begin generating tests. This is where step 2 begins. The goal in step 2 is to find the minima and maxima execution times of each specific path in the software component. A series of tests need to be generated in succession until

	PathValue	VariableName	VariableValue
1	1-2-3-4	x	0
2	1-2-3-4	y	64
3	1-2-3-4	x	0
4	1-2-3-4	y	32
5	1-2-3-4	x	0
6	1-2-3-4	y	34
7	1-2-3-4	x	0
8	1-2-3-4	y	66
9	1-2-3-4	x	0
10	1-2-3-4	y	67

Figure 4.1.: Sample Input Parameter-Path Map

PPPT is confident that it has found the minima and maxima execution times or it has reached the maximum number of testing rounds specific by the tester.

After PPPT has iterated over each path and found the minima and maxima execution times for each path, it can then begin to analyze the data and display it to the user in a useful format. This data will provide the user with a concise overview of the various paths and provide insight into any problem areas that occur in the software. We will delve further into the output of PPPT in the coming sections.

4.2 Implementation

Figure 4.2 shows a decision tree for determining what method is used to build the input parameter-path map. If the function contains loops, it is not possible to use the constraint solver to build the map. A major limitation of constraint solvers is their ability to execute inside of a loop. In order to utilize the constraint solver, the loop must be unfolded completely causing very large, complex decisions which negatively affect the performance of the solvers. This leads us to the need for a secondary approach when loops are present.

Therefore, PPPT will employ the modified exhaustive approach with code instrumentation to determine the execution path of all input parameter combinations. It

is modified in such a way that any statement that does not directly affect the program path of the software is removed during this step. Next, new statements that record the path are inserted in each unique branch and the parameter combinations are executed iteratively.

For the purpose of this research, this process is done manually. It is left to future work to provide an automated solution using existing instrumentation technology. Once the input parameter-path map is built, it can then be run through the evolutionary component described below to record the actual execution time of the function while targeting specific paths.

In step 1, if PPPT finds that there are no loops, then PPPT can build the input parameter-path map using a constraint solver. For this, we use the Microsoft Constraint Solver Foundation. This generates a data structure that provides the necessary information about the expression to allow for each specific path to be targeted. The constraint solver expects a string, the first step in the process is to build that string. To accomplish this, we sub-class the built-in .NET framework `ExpressionVisitor` class that allows the LINQ expression tree to be visited and analyzed as nodes in a binary tree. This sub-class is the `ConstraintSolverExpressionVisitor` shown in Figure 4.3. During the visiting process, the constraint solving string can be built based on the nodes that determine the path (less than, greater than, equal to, not equal to, etc.). When one of these nodes is visited, a new path is added to the list. Then when a `ParameterExpression` node is visited, that parameter is added to the path to form a complete path constraint. Each path is represented by an `ExpressionPath` object and that object contains a list of `ExpressionParameter` objects, which then contains a list of `PathConstraint` objects.

Regardless of the method in step 1 for building the input parameter-path map, in step 2 we run the function through the evolutionary testing component. The component performs evolutionary testing in the classical sense, but constrains its initial and offspring parameters to meet the criteria of the path constraints so as not to target a different path. An initial population of tests is chosen for each path

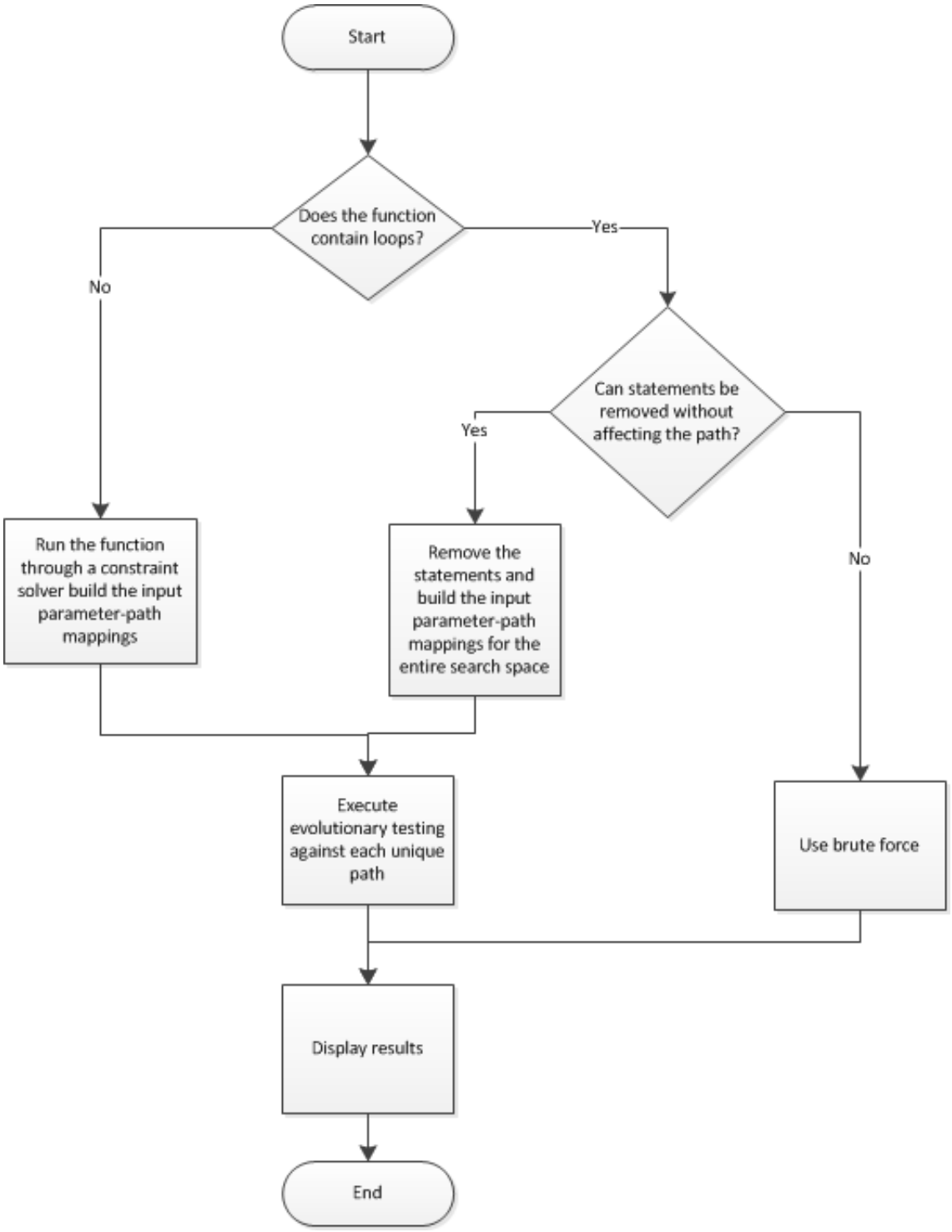

Figure 4.2.: Process Flow

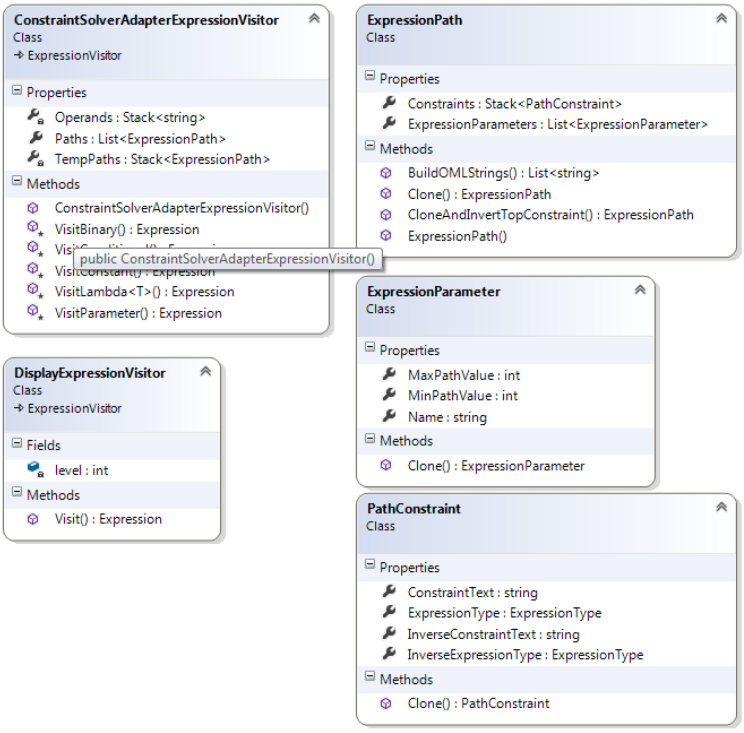

Figure 4.3.: Class Diagram - Constraint Solver Logic

with the population size determined by the tester. At the same time the tester must specify the maximum number of rounds to test, as well as the type of test (minimum or maximum). The path constraints, test function, population size, maximum number of rounds, and type of test are then passed to the fitness service. The service then handles generating inputs for the initial population, executing the function with those inputs, and storing the execution time for each round. Once the entire population is executed, the fitness service can compare the execution times among both the current population and all previous populations to determine the "best fit" test case, which will either be the one with the minimum or the maximum execution time from the previous round.

In each round, the fitness service class generates a new round of input parameters represented by the base class `ApplicationVariable`, shown in Figure 4.4. As you can see, each variable type has a specific sub-class which is responsible for generating new offspring based on the proximity of the previous round.

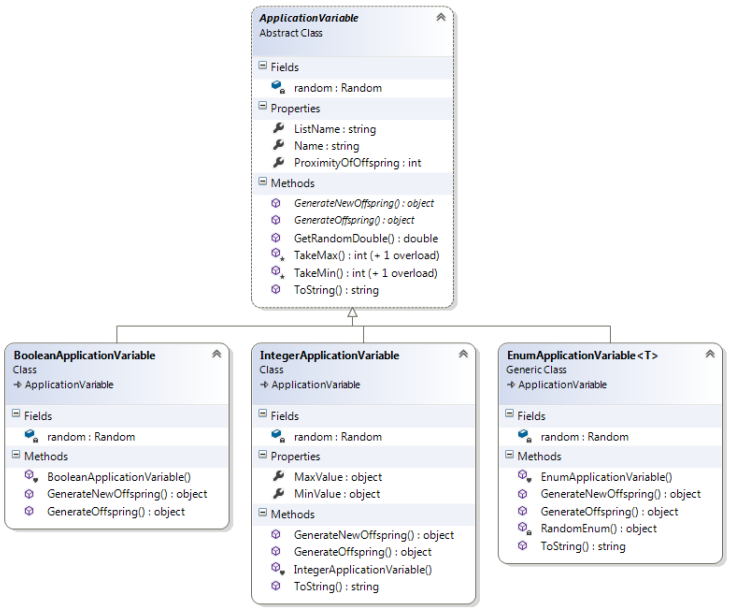

Figure 4.4.: Class Diagram - Application Variables

After each round of testing, meaning the current population has been fully executed and a new best fit has been selected, a new population must be generated. This new population is generated based on a close proximity to the current best fit test case. To do this, all input parameters are inspected independently and a new one is generated that has only a small degree of variance to the existing one, but still meets the path constraints. Of course, if the path constraint requires a parameter to be equal to a constant value, as is the case with `x` in the sleep expression, it will simply be copied to all offspring `x` values. However, if a range of values still target a specific path, then a "proximity of offspring" value is used. This proximity of offspring is simply a percentage of the total range of the offspring from 0 to 100. If the value is 10, for example, the new offspring will all be the current best fit value +/- 10, so long as that still falls within the range necessary to target the given path.

It's worth noting that the proximity concept works quite well with integer and decimal values, but some creativity has to be used when dealing with other data types. For enumerated types and boolean, the proximity percentage can still be used but in a slightly different way. If a proximity percentage of 10 is specified, this can

mean that 10% of the new offspring will have a new enumerated or boolean value, while the other 90% will have the same value. In a multi-parameter function where other inputs are integers, this could be very useful because it would allow the integer parameters to slide across a range while periodically varying the enumerated types and booleans to test whether a different value might cause a different performance. Chances are a single boolean value would not cause a change in performance unless some out of process I/O call is made based on that value or a new path is taken. If dealing with lists, one could vary the list size based on the proximity value.

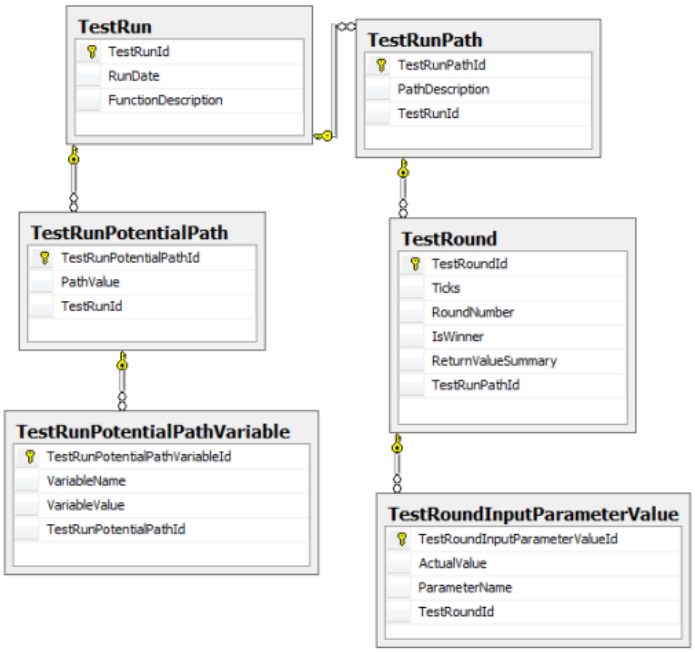

Figure 4.5.: Database Diagram

Once the maximum number of rounds is met, the data is then stored into a local database so that it can be analyzed immediately or at some later date. The database structure itself consists of only five tables, allowing the test number, execution date, path, round, and parameter information to be stored. The full database diagram is shown in Figure 4.5. The master table that stores a single test is `TestRun`. During the input parameter-path map stage, potential paths and input parameters are stored

in the `TestRunPotentialPath` and `TestRunPotentialPathVariable` tables. After this stage is complete, values can be pulled from this table to generate new offspring for testing. The actual results of the evolutionary testing phase are stored in the `TestRunPath`, `TestRound`, and `TestRoundInputParameterValue` tables, which store the execution time of each test and the actual values of each parameter during the test.

4.3 Application of PPPT to a Simple Problem

The first example shown in Figure 4.6 demonstrates a purely hypothetical and simple function that is designed specifically to highlight the potential capability of PPPT. It is a LINQ expression that consists of four unique paths, each with different performance characteristics. x determines what path is taken and y is passed to the sleep function, either as-is or after being multiplied. The sleep function simply causes the current thread to wait for a certain number of milliseconds before proceeding. This provides a mechanism for demonstrating the order of magnitude of certain paths in a controlled and predictable environment.

```
1 Expression<Func<int, int, string>> e = (x, y) =>
2   x == 0 ? "Invalid x Value"
3   : x == 1 ? Sleep(y, "Linear")
4   : x == 2 ? Sleep(y * 2, "Linear times 2")
5   : x == 3 ? Sleep(y * y, "Quadratic")
6   : "Invalid x Value";
```

Figure 4.6.: Sleep LINQ Expression

This function lends itself well to a discussion on why the analysis needs to be done on a path-specific basis. Without path-specific information, the results would show global minimum and maximum execution time for all combined paths. This would essentially only provide real information for the quadratic path where $x = 3$ and provide little (or no) value about other paths. This example is obviously simple enough that one could predict the performance of each path without actually running the tests. If, however, we assume that one of these paths represents a failed

or exception case while other paths represent a successful execution, It would be beneficial to state that when an exception occurs, the maximum execution time will never exceed n msec.

5 RESULTS FOR APPLYING PPPT TO SOFTWARE COMPONENTS

This chapter demonstrates our findings from applying PPPT to four test components. These test functions were chosen to effectively highlight the challenges we faced and how PPPT was designed to overcome those challenges. This chapter is laid out as follows: First, Section 5.1 defines the system requirements for reproducing the environment used to generate our results. Section 5.2 shows the results when applied to the Sleep LINQ Expression shown in Figure 4.6. Section 5.2 also does a comparison of the genetic algorithm approach and a purely random test data generation. Section 5.3 discusses the results when PPPT is applied to a hypothetical function with an exception pathway. This is meant to highlight the ability to target different branches and identify the branches with potential issues. Section 5.4 shows the results of PPPT when applied to the RSA cryptographic algorithm. This demonstrates the limitations of the constraint solver and the need for a modified exhaustive approach. Finally, Section 5.5 applies PPPT to a non-recursive version of the Euclidean GCD algorithm, which further highlights the limitations of constraint solvers, as well as the limitations of the modified exhaustive approach.

5.1 Experimental Setup

All tests in this thesis were generated using a 64-bit Windows 7 Enterprise edition© laptop. The machine has a 2.50 GHz Intel® Core™ i5-2520M processor and 8.00 GB of RAM.

Note, however, that results will vary slightly based on the current usage of the processor and memory for other tasks. There is no guarantee that context switching will not occur in the middle of a test while the timer is running. These outlying

test cases are rare and are discussed in more detail in Section 6. A mechanism for handling or removing outliers is left for future work.

5.2 Analysis of Sleep LINQ Expression

The first function that will be analyzed is the Sleep LINQ Expression shown in Figure 4.6. There are two input parameters, x and y. The x value determines the path and the y value is used in the equation and directly affects the performance. Even though this is clear to any user looking at the function, the software doing the analysis is given no hints about the available values or which parameters affect performance.

The application first determines the values needed to target a specific path. This process is described more in the previous section. Once that information is determined, the paths are targeted to determine either the minimum execution time or the maximum execution time. Figure 5.1 below shows the maximum execution times for the path $x == 1$, which is the "Linear" branch. The population size of each round is 10 and 10 rounds were performed for a total of 100 overall tests.

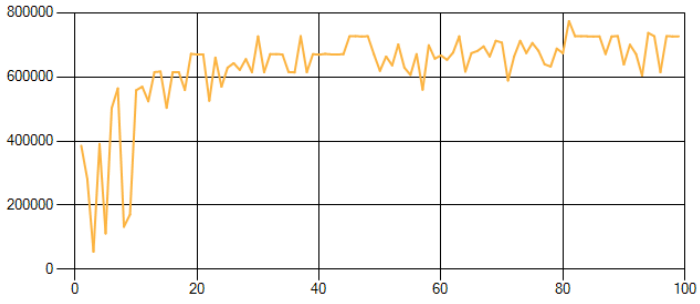

Figure 5.1.: Maximum Execution Time of Linear Sleep Expression in Ticks (10 nS)

Figure 5.2 below shows the corresponding values that produced the graph in Figure 5.1. It is clear that the application determines that the x value must remain constant at 1 while a larger y value produces larger execution times. It is difficult to see in the image, but there is a single line showing the y parameter staying constant at $x == 1$.

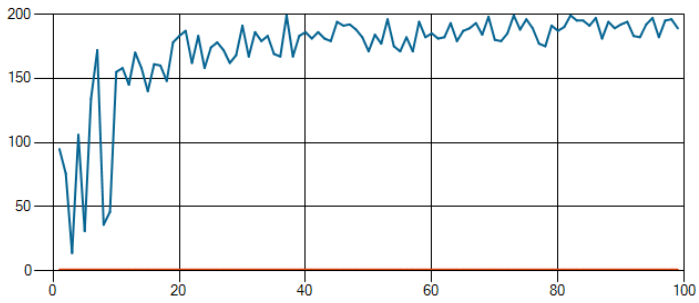

Figure 5.2.: Maximum Values of Linear Sleep Expression

From this graph, it is shown that the actual maximum execution time is 200 and that is found in round 3, or test #38. The tests will continue until the maximum number of rounds is reached to ensure that there are no other values that cause a greater execution time. This is one area that is a potential for future work. It may be possible to short circuit the testing if it can be determined that no other value will cause a higher execution time using statistical analysis or some other mechanism. For this work, the tests simply run out the pre-determined number of rounds.

A comparison of these results to a random selection of input parameters was also performed. This test was not strictly random, as it still targeted a specific path so that the results would be meaningful. Figure 5.3 and Figure 5.4 show the execution time and values, respectively, of the linear branch. It is interesting that test #17 in the 2nd round comes very close to finding the y that causes the maximum execution time. However, since the application does not build on that knowledge, the actual maximum value of y is not found until test #74.

For brevity, the remaining branches' results will be left out of this paper. The results are very similar to the linear branch results, except the produce much higher execution times.

5.3 Analysis of Exception Pathways

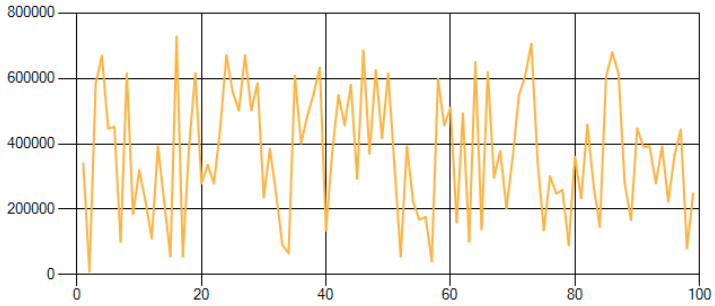

Figure 5.3.: Random Execution Time of Linear Sleep Expression in Ticks (10 nS)

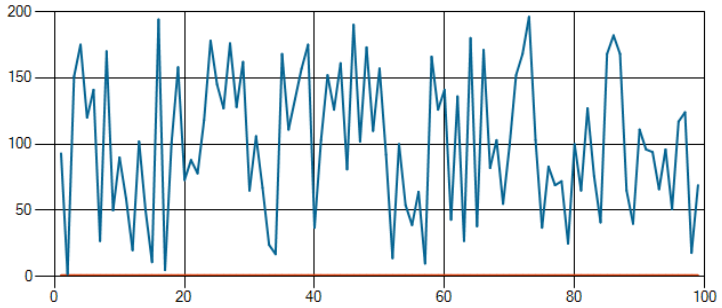

Figure 5.4.: Random Values of Linear Sleep Expression

The next example, shown in Figure 5.5, demonstrates the cost of throwing exceptions. In this case, a C# developer might choose to throw an `ArgumentException` or `ArgumentNullException` for each of the parameters in a function before executing the main path of the function. This is considered best practice regardless of the programming language. However, one must consider the performance consequences in doing so. The following LINQ expression shows a simple function with one integer parameter, arg.

```
1 Expression<Func<int, int, string>> e = (x, y) =>
2   x == 0 ? "Invalid x Value"
3   : x == 1 ? Sleep(y, "Linear")
4   : x == 2 ? Sleep(y * 2, "Linear times 2")
5   : x == 3 ? Sleep(y * y, "Quadratic")
6   : "Invalid x Value";
```

Figure 5.5.: Exception LINQ Expression

If the parameter is equal to 0, the function immediately throws an `ArgumentException`. If the value is anything other than 0, it simply prints its own value and returns. The body of the function itself is not important, except to show that the simplest of function still returns in less time than an exception.

The results are shown below. Figure 5.7 shows the function under normal circumstances, in which it performs two consecutive Console.WriteLine() function calls, one to print the words "Print Arg:" and the other to print the value of the *arg* variable. Not surprisingly, there is no curve in the performance. It is simply constant at around 900 ticks, regardless of the value of *arg*.

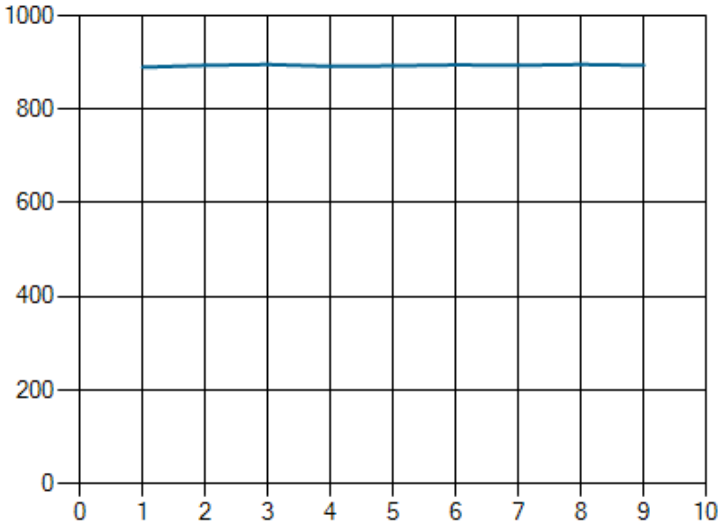

Figure 5.6.: Execution Time of Console.WriteLine() in Ticks (10 nS)

5.7 shows the second path, the exception path. In this case, the application forces the value of *arg* to 0 for every test case, as it did in the previous example. Again, the execution time is linear since it generally takes the operating system constant time to throw the same exception. However, what's interesting here is that the execution time is constant around 4,000 ticks, nearly 4.5 times the normal path.

These results would provide a developer the necessary data to determine whether throwing an exception is the appropriate action. It is easily demonstrated that a small change to this application to return error codes would greatly improve the

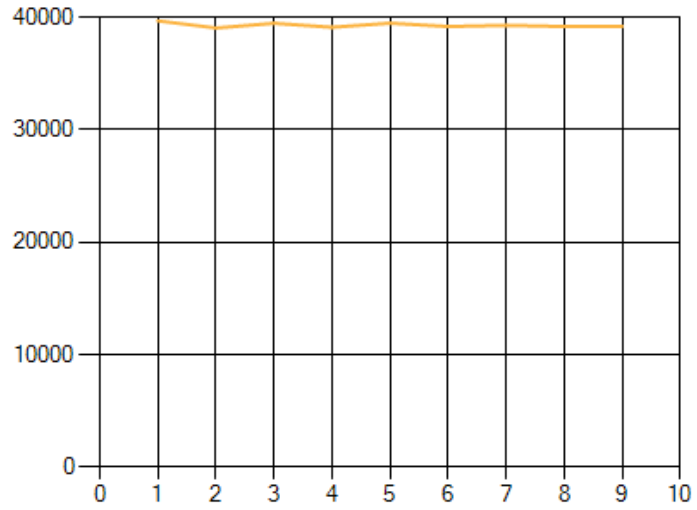

Figure 5.7.: Execution Time of Exceptions in Ticks (10 nS)

performance of the application. However, it then goes against the standard best practice and becomes more difficult to maintain large amounts of error codes. It depends heavily on the nature of the application. Systems with real-time requirements might consider using error codes, while standard business applications might continue to use exceptions.

5.4 Analysis of RSA Cryptographic Algorithm

This leads us to our first real example. We will examine the RSA public-key cryptosystem, most commonly used in SSL and digital signatures. The RSA cryptosystem was designed by Ron Rivest, Adi Shamir, and Leonard Adleman in 1978 [36], thus the acronym "RSA". It's security relies heavily on the difficulty of factoring large integers. The algorithm is described in more detail below.

To encrypt a message M with our method, using a public encryption key (e, n), proceed as follows. (Here e and n are a pair of positive integers.) First, represent the message as an integer between 0 and n - 1. (Break a long message into a series of blocks, and represent each block as such an integer.) Use any standard represen-

tation. The purpose here is not to encrypt the message but only to get it into the numeric form necessary for encryption. Then, encrypt the message by raising it to the e^{th} power modulo n. That is, the result (the ciphertext C) is the remainder when M^{e} is divided by n. To decrypt the ciphertext, raise it to another power d, again modulo n. The encryption and decryption algorithms E and D are thus:

$C = E(M) = M^{\text{e}}(modn)$, for a message M.
$M = D(C) = C^{\text{d}}(modn)$, for a ciphertext C.

In our example, we will see a scenario where keys are precomputed, but the caller of the application has the ability to regenerate the key sequence before computing the ciphertext. This could be a requirement if the key sequence was compromised or if has been a certain number of days since the last key regeneration. Alternatively, perhaps a new client is introduced in which the key sequence has not yet been generated. In this instance, a new key sequence would also need to be generated.

For illustration purposes, we will also create a new string based on the length parameter passed in. A more sophisticated system will need to allow the string itself to be passed in, but for the purpose of demonstration, we will simply pass in the string length and create the string based on that length. This will suffice for the purposes of testing the RSA algorithm.

The full example is shown in Figure 5.8. The function begins by extracting the parameters from the object array. There are three parameters, *stringLength*, *blockLength*, and *shouldRecomputeKeys*. The first parameter, as discussed previously, determines the length of the string to test. The second parameter determines the block length. Various block lengths could potentially have an impact on the mathematical modulo operations. The last parameter, *shouldRecomputeKeys* tells the function whether to recompute the keys prior to executing the algorithm. The contents of the *RecomputeKeys* function is left out for brevity.

```
1  public object ExecuteFunction(object[] parameters)
2  {
3    //Branch 1
4    System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();
5
6    int stringLength = (int)parameters[0];
7    int blockLength = (int)parameters[1];
8    bool shouldRecomputeKeys = (bool)parameters[2];
9
10   string plainTextString = new string('A', stringLength); //Fixed string with length
         as a parameter.
11
12   if (shouldRecomputeKeys)
13   {
14     //Branch 2
15     RecomputeKeys();
16   }
17
18   StringBuilder cipherTextStringBuilder = new StringBuilder();
19   for (int i = 0; i < plainTextString.Length - 1; i += blockLength)
20   {
21     //Branch 3
22     int endValue = (i + blockLength >= plainTextString.Length) ? plainTextString.
           Length - 1 : i + blockLength;
23
24     Byte[] plainTextBytes = encoding.GetBytes(plainTextString.Substring(i, endValue -
           i));
25     BigInteger plainText = new BigInteger(plainTextBytes);
26
27     //Compute cipherText
28     BigInteger cipherText = BigInteger.ModPow(plainText, e, n);
29     Byte[] cipherTextBytes = cipherText.ToByteArray();
30
31     cipherTextStringBuilder.Append(System.Text.Encoding.ASCII.GetString(
           cipherTextBytes));
32   }
33
34   //Branch 4
35   string cipherTextString = cipherTextStringBuilder.ToString();
36
37   return null;
38 }
```

Figure 5.8.: Customized RSA Implementation

As you can see, this program is written in standard C# rather than in LINQ. This is a good time to segue into a discussion of the limitations of modern constraint solvers. The constraint solver that was used in the previous two examples, Microsoft Solver Foundation, requires that all constraints be known ahead of time and not derived from previous conditions. This poses a very difficult problem when dealing with loops in our case since we want to know *all* parameter combinations that satisfy a given path, and we consider each loop iteration a different path. This is one area where our research differs from the typical use of constraint solvers for path-guided exploration. Previously, most research uses constraint solvers to ensure that a branch

is tested, not necessary guaranteeing that it will be hit the maximum number of times. It is imperative that our approach know exactly how many times a loop condition will be met and that we can group input parameter combinations by their final program execution path.

This means, for the introduction of looping conditions, we must use instrumentation instead of a constraint solver. The form of instrumentation used is a modified form of standard code instrumentation. The first step is an analysis to determine what operations can be removed without affecting the final program execution path. This gives us an idea of which statements can be safely removed, while still allowing us to build a library of input parameter-path mappings without the requirement of doing a full execution of every input parameter combination. Figure 5.9 shows the RSA algorithm instrumented in this fashion.

As you can see, we've managed to remove almost the entire core of the algorithm without affecting the path conditions in any way. For example in branch #2, we now only have a statement indicating that the branch condition was met, not the actual call to the *RecomputeKeys* function. Similarly, the logic for performing the encryption was left out of branch #3. This is a significant gain in performance over executing the entire search space to determine the input parameter-path mappings.

The next step in this process is to use the library of input parameter-path mappings that we've built and run it through our evolutionary test generator to quickly determine the best and worst case execution times of each path. This process is no different from the previous two examples, where the system starts with an initial population targeting a specific path, executes the function in its original form, records the execution time of each iteration, then uses that information to generate a new round of tests that target the same path in hopes that the execution time will start to converge on the desired result. The results from the execution of the RSA cryptographic algorithm are shown in Figure 5.10.

In its raw form, several things are clear. First and foremost, there is a great distinction in performance among certain paths. There is clearly a connection between

```
1  public List<PotentialPathValue> GetAllPotentialPathValues()
2  {
3    var values = new List<PotentialPathValue>();
4    for (int stringLength = 0; stringLength < 20; stringLength++)
5    {
6      for (int j = 1; j <= 20; j++)
7      {
8        bool shouldRecomputeKeys = true;
9        int loops = 0;
10       while (loops < 2)
11       {
12         System.Text.ASCIIEncoding encoding = new System.Text.ASCIIEncoding();
13         string plainTextString = new string('A', stringLength); //Fixed string with
               length i.
14         int blockLength = j;
15
16         //Log the possible x and y values.
17         var value = new PotentialPathValue();
18         values.Add(value);
19         value.Add(stringLength);
20         value.Add(blockLength);
21         value.Add(shouldRecomputeKeys);
22
23         //Log that we took this path.
24         value.PathDescription += "-1";
25
26         if (shouldRecomputeKeys)
27         {
28           //Log that we took this path.
29           value.PathDescription += "-2";
30         }
31
32         StringBuilder cipherTextStringBuilder = new StringBuilder();
33         for (int i = 0; i < plainTextString.Length - 1; i += blockLength)
34         {
35           //Log that we took this path.
36           value.PathDescription += "-3";
37         }
38
39         string cipherTextString = cipherTextStringBuilder.ToString();
40
41         //Log that we took this path.
42         value.PathDescription += "-4";
43
44         //Run the loop a second time with it being false.
45         shouldRecomputeKeys = false;
46         loops++;
47       }
48     }
49   }
50   return values;
51 }
```

Figure 5.9.: Instrumented RSA Implementation

all of the paths in the 800,000 to 1,000,000 ticks range. Second, no individual path appears to be affected by the evolutionary portion of the testing. What this tells us is that the internal performance of each does not vary much based on parameter combinations.

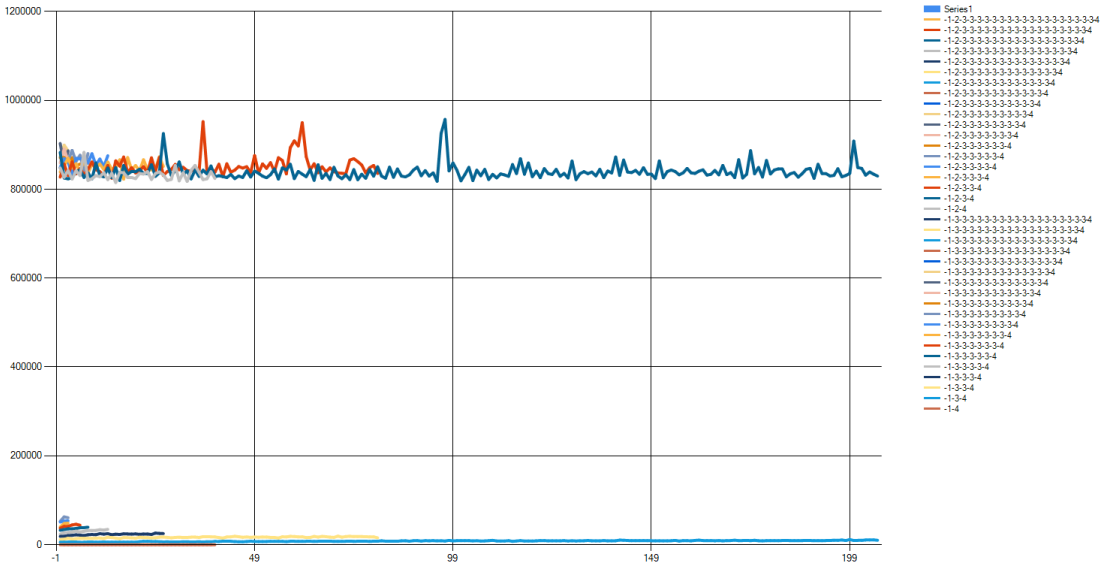

Figure 5.10.: RSA Results Showing All Paths in Ticks (10 nS)

What's not so clear by the results shown in Figure 5.10 is what branch conditions are causing that poor performance. This is the critical knowledge that a developer will need to remedy the system, so our work is not done. We then must take the same data set and group it by which branches were met, excluding how many times that it was met. If we do this, we get much clearer results (shown in 5.11). The results show definitively that input parameters resulting in the execution of branch #2 have far greater execution times than those that skip that branch.

Branch #2, of course, is the one where the function determines whether it should recompute the keys. This tells the developer which area to focus on when choosing to spend time improving the performance of the system. In this case, the developer will need to rethink the method used to recompute the keys. If the algorithm for recomputing the keys is already optimized to full potential, perhaps they can compute a block of them at the beginning of the day and choose one from that precomputed list. If they were to do this, the *RecomputeKeys* method would then be replaced with a call to the database to get a fresh set of keys, rather than doing the computation on the fly. This would result in a serious performance gain depending on the key size. For

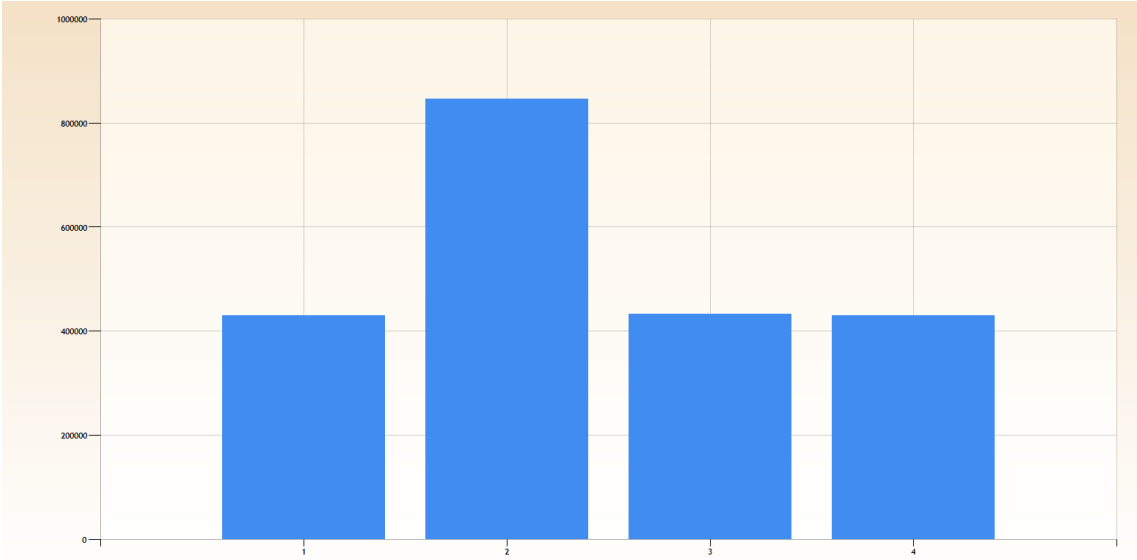

Figure 5.11.: RSA Results Grouped by Branch in Ticks (10 nS)

any standard key size in real cryptosystems (1024-bit and above), the performance savings will be dramatic.

Operation	Time in Ticks (10 ns
Execution Time to Build Input Parameter-Path Map	10,444
Actual Execution Time of Evolutionary Testing	77,754,559
Combined Execution Time	**77,765,003**
Total Brute Force Execution Time for All Paths	351,519,155
Difference (Total Time Savings over Brute Force)	**273,754,152**

Figure 5.12.: RSA Results Compared to Brute Force

The actual execution times are compared to a brute force/purely exhaustive approach in Figure 5.12. The time it takes to build the input parameter-path map is minimal. We combine that with the actual execution time of the evolutionary testing process and it is still significantly lower than running a brute force approach, and this is not even for the entire search space; it is only for string lengths and block lengths of up to 20. When applied to a larger search space of entire documents worth of text, it would become impossible to use a brute force approach, but it would still be feasible to use PPPT.

Incorporating this approach into PPPT shows real potential for reducing the number of overall tests for targeting specific paths. We no longer need to use brute force to run all values. We can quickly build a library of input parameter-path combinations, and use evolutionary testing beyond that to characterize the performance and, if they exist, find minima and maxima execution times for every path.

However, no good example would be complete without a corresponding example where this technique breaks down. We will see in the next section an example where this technique falls short because there are simply no calls that can be removed during the input parameter-path discovery stage. Each statement in the function is dependent upon the previous one and the constraint library cannot be built without each call being present. If we try to use the same approach, we actually experience worse performance than if a brute force approach were to be used. This is obviously not the desired behavior.

5.5 Analysis of Euclidean GCD Algorithm

The final example we will cover is the non-recursive Euclidean GCD algorithm [37], as seen in Figure 5.13. This is a well-known algorithm for factoring two integers to find there greatest common factor. It is named after the Greek mathematician Euclid, who first described it in Books VII and X of his Elements. The algorithm has two modes, recursive and non-recursive. The non-recursive version simply uses a while loop and temporary variables to simulate the recursion.

You will quickly notice that the algorithm relies heavily on loop conditions, therefore we cannot use the constraint solver as we did in the first two examples. We will then attempt to use the instrumentation. The instrumented code can be seen in Figure 5.14. This, too, poses a problem. There simply are no places where code was able to be removed during instrumentation. Each statement in the function is dependent upon the previous one. This is an example where this technique falls short and cannot be used.

```
1  public object ExecuteFunction(object[] parameters)
2  {
3    //Branch 1
4
5    int x = (int)parameters[0];
6    int y = (int)parameters[1];
7
8    if (x < y)
9    {
10     //Branch 2
11     x += y;
12     y = x - y;
13     x -= y;
14   }
15
16   if (y == 0)
17   {
18     //Branch 3
19     return x;
20   }
21
22   while (x % y != 0)
23   {
24     //Branch 4
25     x += y;
26     y = x - y;
27     x -= y;
28     y %= x;
29   }
30
31   //Branch 5
32   return x;
33 }
```

Figure 5.13.: Non-Recursive Euclidean GCD Algorithm

The performance of this is actually worse than a brute force approach; up to 2x worse. This is because we essentially need to execute the target function as-is for all possible input values just to build the input parameter-path map. At this point, we've brute forced the system and still know *nothing* about its performance characteristics. Then we still need to run the evolutionary phase, which executes the function again many times and logs the execution time.

```
1  public List<PotentialPathValue> GetAllPotentialPathValues()
2  {
3    var values = new List<PotentialPathValue>();
4
5    for (int i = 0; i < 50; i++)
6    {
7      for (int j = 0; j < 50; j++)
8      {
9        //Create temporary variables to avoid fiddling with the loop parameters.
10       int x = i;
11       int y = j;
12
13       //Log the possible x and y values.
14       var value = new PotentialPathValue();
15       values.Add(value);
16       value.Add(x);
17       value.Add(y);
18
19       //Log that we took this path.
20       value.PathDescription += "-1";
21
22       //Run the function to determine the path of those values.
23       if (x < y)
24       {
25         x += y;
26         y = x - y;
27         x -= y;
28
29         //Log that we took this path.
30         value.PathDescription += "-2";
31       }
32
33       if (y == 0)
34       {
35         //Replace return statement with path information.
36         value.Result = x;
37         value.PathDescription += "-3";
38
39         //Continue takes the place of the commented out return value.
40         //Essentially it means we're done checking this combination.
41         continue;
42       }
43
44       while (x % y != 0)
45       {
46         x += y;
47         y = x - y;
48         x -= y;
49         y %= x;
50
51         value.PathDescription += "-4";
52       }
53
54       //Replace return statement with path information.
55       value.Result = y;
56       value.PathDescription += "-5";
57     }
58   }
59
60   return values;
61 }
```

Figure 5.14.: Instrumented Non-Recursive Euclidean GCD Algorithm

6 CONCLUDING REMARKS

This thesis has offered a viable solution to automated performance testing and classification of software components. We have combined the use of evolutionary testing, genetic algorithms, constraint solvers, and instrumentation to provide two mechanisms for characterizing software performance with better-than-exhaustive execution times. Likewise, looping conditions must be handled outside of a constraint solver and a gain can only be realized if there are statements that can be removed without affecting the path. If no statements can be removed, then a brute force approach is the best option.

Based on our current work on PPTT, the following is a summary of lessons learned from the research work presented in this thesis:

- To properly classify the performance of software components based in a software system, it is necessary to employ techniques to target specific branches of the software. Genetic algorithms alone are not sufficient for classifying local minima and maxima execution times.

- Modern constraint solvers offer no mechanism for building full constraint maps for all possible iterations of a loop. In such a scenario, the modified exhaustive approach discussed in this thesis should be used instead.

- During the testing phase, several test cases produced erroneous results due to peaks in CPU usage by system processes. This caused the scheduler to devote less time to the testing process and resulted in higher execution times for tests that would otherwise run more quickly. When this happens, the fitness service becomes "confused" and continually generates new inputs based on that very high outlying test case. One option to overcome this problem is be to run each

test several times with the same inputs and use an average. At that point, outliers would become far less common.

- Each individual data type of input parameter needs to have a specialized class responsible for generating new offspring input parameters in the evolutionary testing phase. A boolean (true/false) parameter cannot be treated the same as an integer parameter that uses a range of values. Future work therefore will include supporting additional data types beyond simple primitives. Supporting more complex data types (*e.g.*, interfaces, generics, lists, and abstract data types) will provide a far more rich user experience that can be used in production systems. The current state of PPPT allows for more data types to be supported very easily by subclassing the `ApplicationVariable` class and adding a new 'Create' method method in the `ApplicationVariableFactory` class to create new instances of that variable.

LIST OF REFERENCES

LIST OF REFERENCES

[1] Walter S. Heath. *Real-Time Software Techniques*. Van Nostrand Reinhold, 1991.

[2] Hartmut Pohlheim. Tutorial, genetic and evolutionary algorithm toolbox for use with matlab. 1996.

[3] Joachim Wegener, Klaus Grimm, Matthias Grochtmann, Harmen Sthame, and Bryan Jones. Systematic testing of real-time systems. *Eurostar*, 1996.

[4] Wasif Wasif, Richard Torkar, and Robert Feldt. A Systematic Review of Search-based Testing for Non-functional System Properties. *Information and Software Technology*, 51(6):957–976, 2009.

[5] Hong Zhu, Patrick Hall, and John May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29:366–427, 1997.

[6] Wikipedia. Hill Climbing. `http://en.wikipedia.org/wiki/Hill_climbing`.

[7] Real-Coded Adaptive Range Genetic Algorithm Applied to Transonic Wing Optimization. *Parallel Problem Solving from Nature VI*.

[8] Sembiam R. Rengarajan. Genetic algorithm applications in the Juno radiometer antenna arrays. *IEEE Antennas and Propogation Society International Symposium*, pages 1–4, June 2009.

[9] Jin-Lee Kim. Integrated Genetic Algorithm and its Applications for Construction Resource Optimization. *IEEE Proceedings of the 2010 Winter Simulation Conference*, pages 3212–3219, 2010.

[10] Bogdan Korel. Automated Software Test Data Generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, August 1990.

[11] Lori A. Clarke. A Program Testing System. *Proceedings of the 1976 Annual Conference*, pages 488–491, 1976.

[12] P. Maragathavalli, S. Kanmani, J. Sam Kirubakar, and P. Sriraghavendrar. Automatic program instrumentation in generation of test data using genetic algorithm for multiple paths coverage. In *IEEE-International Conference On Advances In Engineering*, pages 349–353, 2012.

[13] The free on-line dictionary of computing. `http://dictionary.reference.com/browse/branchcoveragetesting`, Mar 2013.

[14] Yan-Quan Zhou. A study on optimizing execution time and code size in iterative compilation. *Innovations in Bio-Inspired Computing and Applications (IBICA), 2012 Third International Conference on*, pages 104–109, September 2012.

[15] Wikipedia. Constraint Solver. `https://en.wikipedia.org/wiki/Constraint_satisfaction_problem`.

[16] J. Dong, H. Ding, C. Ren, and W. Wang. Ibm smartscor - a scor based supply chain transformation platform through simulation and optimization techniques. *Proceedings of the 38th conference on Winter simulation*, pages 650–659, 2006.

[17] A. Brodsky. Service composition language to unify simulation and optimization of supply chains. *Hawaii International Conference on System Sciences, Proceedings of the 41st Annual*, page 74, January 2008.

[18] M. Bennekrouf. Optimal design of two levels reverse logistic supply chain by considering the uncertain quantity of collected multi-products. *Logistics (LOGISTIQUA), 2011 4th International Conference on*, pages 397–404, May 2011.

[19] K. Kuchcinski. Constraints-driven scheduling and resource assignment. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 8:355–383, July 2003.

[20] Alessio Bonfietti, Luca Benini, Michele Lombardi, and Michela Milano. An efficient and complete approach for throughput-maximal sdf allocation and scheduling on multi-core platforms. *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 897–902, 2010.

[21] Claude Le Pape. Implementation of resource constraints in ilog schedule: a library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3:55–66, 1994.

[22] Liliana Cucu-Grosjean. Global multiprocessor real-time scheduling as a constraint satisfaction problem. *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 42–49, September 2009.

[23] Georges Weil. Constraint programming for nurse scheduling. *Engineering in Medicine and Biology Magazine, IEEE*, 14:417–422, July 1995.

[24] Kwaiter Ghassan. A general approach to constraint solving for declarative modeling domain. *Information Visualization, 1999. Proceedings. 1999 IEEE International Conference on*, pages 424–431, 1999.

[25] P. Griebel, G. Lehrenfeld, W. Mueller, C. Tahedl, and H. Uhr. Integrating a constraint solver into a real-time animation environment. *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 12–19, September 1996.

[26] Xueliang Huang. A 3d geometric constraint solver for direct manipulation of b-rep model. *Computer-Aided Design and Computer Graphics, 2009. CAD/-Graphics '09. 11th IEEE International Conference on*, pages 351–355, August 2009.

[27] Alejandro Arbelaez. Continuous search in constraint programming. *Tools with Artificial Intelligence (ICTAI), 2010 22nd IEEE International Conference on*, 1:53–60, October 2010.

[28] Arnaud Lallouet. Consistency for partially defined constraints. *Tools with Artificial Intelligence, 2005. ICTAI 05. 17th IEEE International Conference on*, pages 8–125, November 2005.

[29] Feng-Tyan Lin. Application of mathematical constraint resolution to decision support system. *Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International*, pages 685–692, September 1989.

[30] Ricardo Mejia-Gutierrez. Knowledge modelling for supporting decision making in optimal distributed design process. *Industrial Engineering and Engineering Management, 2007 IEEE International Conference on*, pages 2076–2080, December 2007.

[31] S. Han, Y. Dang, S. Ge, D. Zhang, and T. Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 145–155, 2012.

[32] Donald Chai. A fast pseudo-boolean constraint solver. *Proceedings of the 40th annual Design Automation*, pages 830–835, 2003.

[33] Akshat Kumar. Distributed constraint optimization with structured resource constraints. *Proceedings of The 8th International Conference on Autonomous Agents and Multiagent Systems*, 2:923–930, 2009.

[34] Microsoft Solver Foundation. `http://msdn.microsoft.com/en-us/devlabs/hh145003.aspx`.

[35] Shih-Hao Hung, Shu-Jheng Huang, and Chia-Heng Tu. New tracing and performance analysis techniques for embedded applications. In *The 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 143–152, 2008.

[36] R.L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21:120–126, 1978.

[37] Thomas L. Heath. *The Thirteen Books of Euclid's Elements*. Dover Publications, 1956.