U. PORTO

FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Spectrum-based Diagnosis for Run-time Systems

Nuno Cardoso

Supervisor: Professor Rui Abreu

Co-Supervisor: Professor David Garlan

Doctoral Programme in Informatics Engineering

May 27, 2016

Engineering Faculty, University of Porto

Spectrum-based Diagnosis for Run-time Systems

Nuno Cardoso

Dissertation submitted to Faculdade de Engenharia da Universidade do Porto
to obtain the degree of

Doctor Philosophiae in Informatics Engineering

Supervisor: Professor Rui Abreu

Co-Supervisor: Professor David Garlan

May 27, 2016

D

The work developed for this thesis was co-supported by the National Science Foundation under Grant No. CNS 1116848, and by the scholarship number SFRH/BD/79368/2011 from Fundação para a Ciência e Tecnologia.

HASLab
INESCTEC

NSF

FCT
Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA

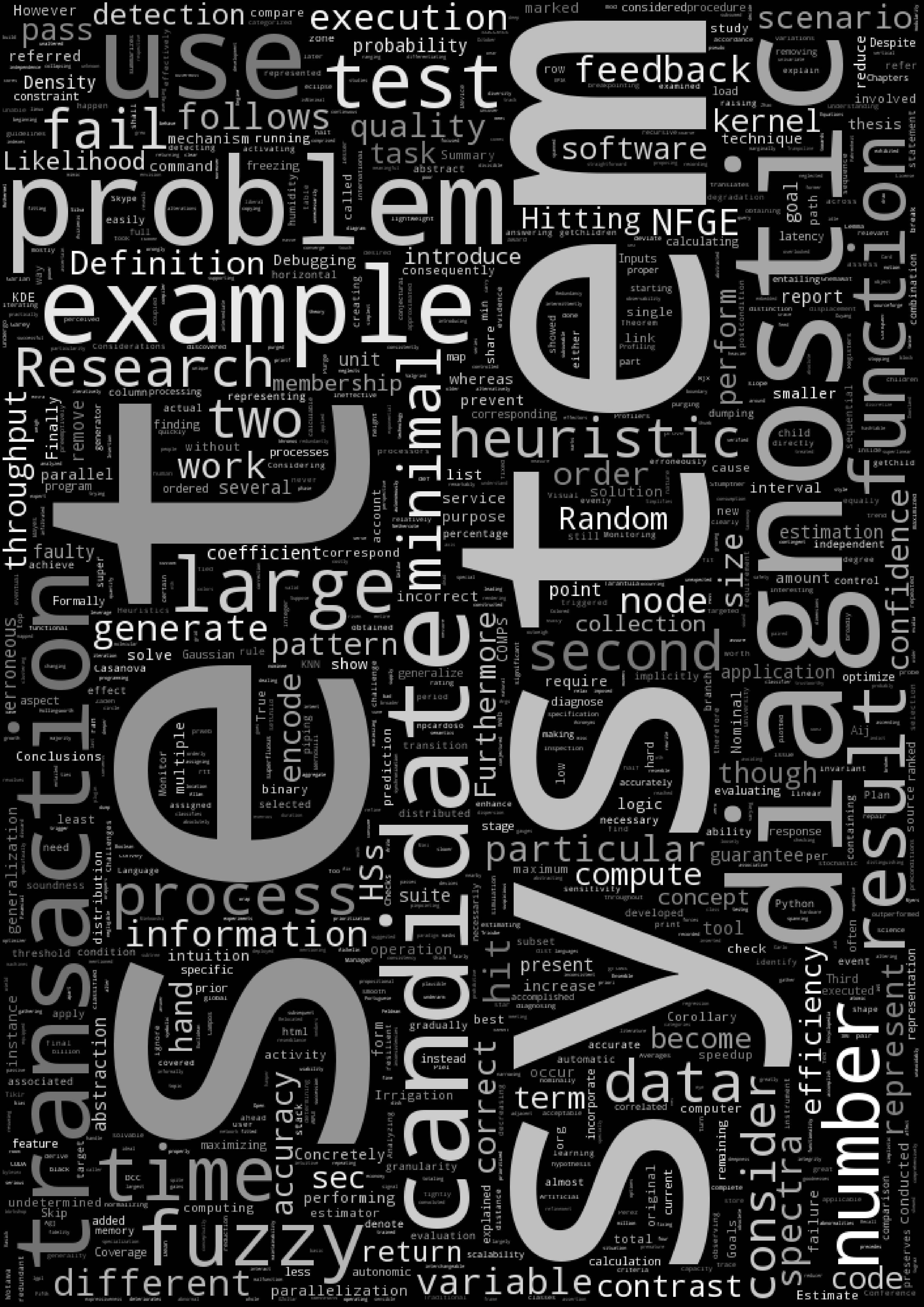
POPH

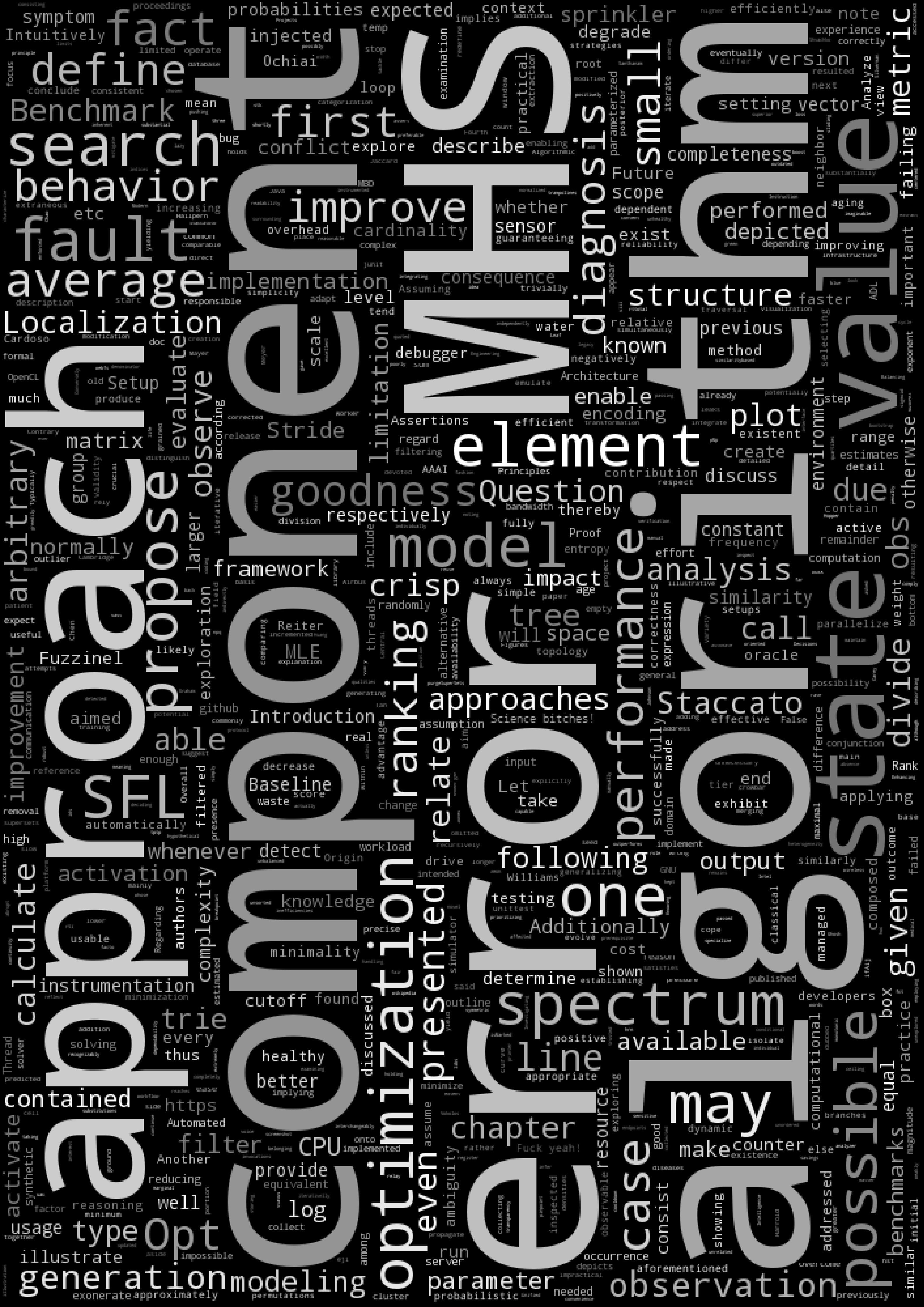
QREN
QUADRO
DE REFERÊNCIA
ESTRATÉGICO
NACIONAL

GOVERNO DE
PORTUGAL

MINISTÉRIO DA EDUCAÇÃO
E CIÊNCIA

UNIÃO EUROPEIA
Fundo Social Europeu


system
set
diagnostic
candidate
transaction
problem
use
example
test
result
number
function
heuristic
large
minimal
time
fuzzy
Research
process
information
data
consider
spectra
execution
scenario
feedback
software
kernel
detection
fail
follows
quality
task
pass
Likelihood
Definition
Hitting
NFGE
introduce
two
work
order
solution
Random
node
collection
second
particular
compute
generate
pattern
encode
HSs
hit
correct
Furthermore
term
variable
contrast
return
different
code
efficiency
confidence
represent
estimation
size
perform
throughput
erroneous
generalization
abstraction
accuracy
sec
membership
coefficient
faulty
parallel
program
several
though
guarantee
concept
become
accurate
application
Density
constraint
referred
However
compare
probability
marked
considered
procedure
study
Despite
reduce
refer
Chapters
involved
thesis
technique
latency
report
smaller
interval
amount
control
optimize
check
Third
executed
response
Plan
logic
necessary
evaluating
Irrigation
Corollary
present
increase
accomplished
tool
developed
speedup
computer
automatic
instead
activity
html
gradually
best
hand
prior
specific
intuition
condition
threshold
instance
associated
feature
undetermined
Skip
added
memory
Coverage
parallelization
performing
estimator
denote
evaluation
autonomic
calculation
total
original
current
almost
Concretely
granularity
maximizing
computing
less
Conclusions
multiple
binary
selected
distributed
service
purpose
percentage
list
point
incorrect
require
diagnose
solve
Gaussian
rule
show
generalize
Formally
KDE
command
freezing
mechanism
running
Summary
abstract
Debugging
horizontal
consequently
Inputs
proper
starting
single
link
part
map
whereas
prevent
corresponding
unit
actual
finding
without
processes
ordered
Finally
remove
achieve
aspect
effect
least
need
soundness
distribution
apply
Casanova
Language
Challenges
suite
operation
subset
occur
explained
Monitor
assigned
prediction
True
account
COMPS
stage
enhance
cause
child
directly
getChild
invariant
ranked
source
independent
degree
representation
pair
shape
event
often
Python
Goals
failure
conducted
comparison
preserves
Estimate
Conference
store
remaining
correlated
scalability
calculating
answering
getChildren
goal
path
sequential
combination
statement
break
window
represented
eclipse
study
row
load
raising
recursive
easily
full
Skype
humidity
called
mostly
way
perceived
iterating
Considerations
discovered
column
processing
representing
ineffective
Heuristics
functional
top
Considering
never
processors
Visual
still
Monitoring
triggered
implicitly
branch
Nominal
repair
ability
per
containing
stochastic
outperformed
science
Tarantula
Aij
linear
hard
low
accurately
making
inspection
sensitivity
throughout
maximum
necessarily
simulation
org
learning
hypothesis
incorporate
acceptable
ahead
user
stack
sequence
either
share
evidence
min
showed
done
Profiling
Theorem
Profilers
slope
new
equally
prove
erroneously
trend
interesting
usability
tightly
normalizing
target
derive
Tikir
redundant
Huang
generality
warning
Pietro



algorithm
approach
component
MHS
state
value
possible
element
spectrum
optimization
model
goodness
improve
search
fault
average
behavior
Localization
define
fact
Benchmark
first
diagnosis
small
structure
performance
case
ranking
presented
propose
SFL
Staccato
approaches
crisp
analysis
one
may
following
chapter
observation
parameter
modeling
generation
Opt
type
line
output
plot
metric
failing
divide
given
due
benchmarks
observe
evaluate
arbitrary
improvement
calculate
activate
contained
instrumentation
activation
complexity
framework
tree
space
call
similarity
constant
completeness
previous
method
known
enable
encoding
performed
depicted
version
setting vector
sprinkler
degrade
probabilities expected
injected
Ochiai
conflict
explore
describe
Future
scope
whether
sensor
cardinality
implementation
consequence
Question
respectively
thereby
impact
Introduction
Stride
limitation
debugger
matrix
normally
Fuzzinel
able
Baseline
MLE
Reiter
decrease
healthy
better
trie
solving
cutoff
found
knowledge
minimality
determine
cost
Additionally
testing
available
make
counter
CPU
filter
provide
log
reasoning
well
illustrate
usage
approximately
exonerate
probabilistic
needed
occurrence
aforementioned
environment
otherwise
active
remainder
frequency
computation
parallelize
range
estimates
discuss
create
existent
environment
symptom
Intuitively
Assertions
Setup
produce
much
automatically
whenever
detect
workload
Let
take
end
exhibit
tier
applying
Rank
similarly
practice
computational
managed
ambiguity
resource
consist
showing
addressed
equal
box
classical
composed
outcome
failed
Science bitches!
Fuck yeah!

Spectrum-based Fault Localization in Embedded Software

Vê e aprende!

Porto, 19 setembro 2012

[illegible]

To Lígia, my best friend and love of my life.

```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>+.>>++
.+++++++++++++++++.+.--.<+++++++++++++++++.++.>++.[>]<[[-]<]<<-<++-<--
```

Abstract

Despite the advances made in the Computer Science domain, it remains practically impossible to create faultless systems. Acknowledging this fact, the concept of *self-healing* control loops was introduced. A *self-healing* control loop is an instance of an autonomic control loop, having the purpose of both improving the dependability of some system and reducing human intervention to a minimum. In contrast to other approaches to assure dependability, *self-healing* systems achieve the aforementioned goals by employing mechanisms that, at run-time, either prevent or solve eventual errors.

Among the tasks involved in creating a *self-healing* system is the task of diagnosis. The goal of this thesis is to create an automatic diagnostic framework for run-time/*self-healing* systems. In contrast to the existent approaches, we aim at creating a diagnostic framework that is both general purpose and lightweight.

To achieve our goal while meeting the aforementioned criteria, we improve upon a development-time diagnostic approach called **Spectrum-based Fault Localization (*SFL*)**. On the one hand, *SFL* makes use of a high-level abstraction of the system under analysis (called *spectrum*), making it, in principle, usable in a large diversity of scenarios. Concretely, the only requirements to use *SFL* in a real-world scenario are that (1) the system's activity must be divisible into transactions, (2) the correctness of each transaction must be evaluable, (3) the components' activations must be observable and (4) it must be possible to associate the components' activity with the corresponding transactions. On the other hand, and when compared to classical model-based diagnosis approaches, *SFL* does not require detailed a system model, making both the modeling and diagnostic processes lightweight.

In the scope of *SFL*, the diagnostic process can be divided in two stages: diagnostic candidate generation and ranking. The first stage consists in computing sets of components that, by assuming their faultiness, would explain the observed erroneous behavior. Since many sets may meet the above criteria, in the second stage, the sets are ranked according to their likelihood of being the real explanation for the system's erratic behavior. Concretely, in this thesis, we aim at optimizing both stages. To that end, we

pointed 3 limitations of the current *SFL* approach.

The first limitation is related to the fact that the candidate generation problem is *NP-hard*, thereby being the bottleneck of the *SFL* framework. Taking this fact into account, we propose a novel algorithm, dubbed *MHS*^{2}, that is not only more efficient than state-of-the-art algorithm (*Staccato*) but is also capable of making use of multiple processing units to solve the candidate generation problem. In practice, this improvement translates into: (1) better diagnostic accuracy when setting a time-based cutoff, due to the fact that calculating more candidates increases the likelihood of finding the correct diagnostic candidate, and (2) smaller diagnostic latency when setting a solution size cutoff, due to the fact that calculating a fixed number of diagnostic candidates takes less time with *MHS*^{2} than with *Staccato*.

The second limitation is related to the fact that *SFL* abstracts the transactions' outcome in terms of correct/incorrect behavior. Even though this binary error abstraction is capable of correctly encoding functional errors, when diagnosing non-functional errors, it abstracts error symptoms (such as performance degradation), thus impairing the diagnostic accuracy. In this thesis we propose a novel approach to encode and diagnose both functional and non-functional errors by incorporating in *SFL* concepts from the fuzzy logic domain. First, we propose the replacement of the classical binary logic for fuzzy logic to detect/encode error states. The fuzzy approach to error detection encodes the error state as a continuous variable, taking values between 0 and 1 (corresponding to a pass and fail, respectively), allowing for a more accurate representation of degraded states. Second, we generalize *SFL* to take advantage of the added information. For the conducted benchmark, and when compared to the classical approach, the fuzzy *SFL* approach improved the diagnostic quality in 65% of the test cases.

The third limitation is related to how *SFL* handles fault intermittency. *SFL* accounts for the fact that faulty components may fail intermittently by considering a parameter (known as *goodness*) that quantifies the probability that faulty components may still exhibit correct behavior. The current *SFL* approach does, however, (1) assume that this goodness probability is context independent and (2) does not provide means for integrating past diagnosis experience in the calculation of the goodness parameter. In this thesis we present a novel approach, coined *NFGE*, aimed at addressing such limitations. The first limitation was addressed by generalizing both the hit spectrum abstraction and *SFL* to use information about the system's state in the diagnostic process. The second limitation was addressed by proposing a Kernel Density Estimate approach that uses feedback observations to model the components' goodnesses as a non-linear function of the system's state. We evaluated the approach with both synthetic and real data, yielding lower estimation errors, thus increasing the diagnosis performance.

```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>>++.>+
++++.++++++++++++++.++.--------.++.[>]<[[-]<]+>+->+<<++>><+-<<+++>>>>-
```

Resumo

Apesar dos avanços feitos no domínio da Ciência da Computação, continua a ser impossível, na prática, criar sistemas sem falhas. Face a este facto, o conceito de sistemas *self-healing* foi proposto. Estes sistemas tem como objetivo maximizar a sua fiabilidade e, ao mesmo tempo, reduzir a necessidade de intervenção humana. Em contraste com outras abordagens, os sistemas *self-healing* atingem os objetivos acima mencionados, empregando mecanismos que, durante a execução do sistema, previnem ou resolvem eventuais erros.

Entre as tarefas envolvidas na criação de um sistema *self-healing* está a tarefa de diagnóstico. Esta tese tem como objetivo criar uma sistema de diagnóstico automático para sistemas *self-healing*. Em contraste com as abordagens existentes, estabelecemos como requisitos necessários a criação de um sistema que seja tanto leve como de uso geral.

Para alcançar o nosso objetivo, tendo em conta os critérios mencionados, decidimos melhorar uma abordagem anteriormente utilizada no âmbito do diagnóstico na fase de desenvolvimento do sistema, chamada *SFL*. Por um lado, o *SFL* faz uso de uma abstração de alto nível do sistema sob análise (denominada de *spectrum*), tornando-se, em princípio, utilizável numa grande diversidade de cenários. Na prática, os únicos requisitos para utilizar o *SFL* num cenário real são: (1) a atividade do sistema deve ser divisível em transações, (2) a regularidade de cada transação deve ser avaliável, (3) as ativações dos componentes devem ser observáveis e (4) deve ser possível associar a atividade dos componentes com as transações correspondentes. Por outro lado, e quando comparado com as abordagens tradicionais de diagnóstico baseado em modelos, o *SFL* não requer um modelo detalhado do sistema, tornando leves tanto o processo de modelagem bem como o de diagnóstico.

No âmbito do *SFL*, o processo de diagnóstico pode ser dividido em duas fases: fase de geração e fase de ordenação dos candidatos de diagnóstico. A primeira fase consiste em computar conjuntos de componentes que, assumindo que se encontram num estado faltoso, explicariam o comportamento errático do sistema. Uma vez que muitos conjuntos podem satisfazer este critério, na segunda fase, os conjuntos são classificados de

acordo com a probabilidade de serem a verdadeira explicação para o comportamento observado. Concretamente, o nosso objetivo nesta tese prende-se com a otimização destas duas tarefas. Para tal propusemo-nos a resolver 3 limitações de forma a tornar o *SFL* mais rápido e preciso.

Em primeiro lugar, o problema de geração de candidatos pertence à categoria de complexidade *NP-hard*, sendo assim o *bottleneck* do *SFL*. Tendo em conta este facto, propusemos um novo algoritmo, denominado de *MHS*^{2}, que não só é mais eficiente do que o algoritmo existente (*Staccato*) mas também é capaz de fazer uso de múltiplos *CPUs* para resolver o problema de forma mais rápida. Na prática, esta melhoria traduz-se em: (1) melhoria da precisão do diagnóstico ao definir um *cutoff* baseado em tempo, devido ao facto de que ao calcular mais candidatos a probabilidade de encontrar o candidato diagnóstico correto aumenta e, (2) menor latência de diagnóstico ao definir *cutoff* baseado no tamanho da solução, devido ao facto de que o cálculo de um número fixo de candidatos levar menos tempo com o *MHS*^{2} do que com o *Staccato*.

Em segundo lugar, o *SFL* abstrai o resultado das transações em termos de comportamento correto/incorreto. Embora a abstração binária de erro seja capaz de codificar corretamente erros funcionais, na presença de erros não funcionais, os sintomas de erro (como por exemplo a degradação de desempenho) são completamente abstraídos, prejudicando assim a precisão do diagnóstico. Nesta tese propomos uma abordagem baseada em lógica difusa, denominada de *Fuzzinel*, para codificar e diagnosticar tanto erros funcionais como não funcionais. Em primeiro lugar, propusemos a substituição da lógica binária para lógica difusa no processo de deteção/codificação de estados de erro. Esta abordagem codifica o estado de erro como sendo uma variável contínua, assumindo valores entre 0 e 1 (correspondendo aos estados nominais e faltosos, respetivamente), permitindo assim uma representação mais precisa dos estados de desempenho não ótimo. Em segundo lugar, generalizamos o *SFL* para incorporar esta informação adicional. Nas experiências realizadas, e quando comparado com a abordagem anterior, o *Fuzzinel* melhorou a qualidade de diagnóstico em 65% dos casos de teste.

Em terceiro lugar, o *SFL* lida com o facto de os componentes defeituosos poderem falhar intermitentemente considerando um parâmetro (denominado de *goodndess*) que quantifica a probabilidade de um componente defeituoso apresentar um comportamento correto. No entanto, a abordagem atual assume que (1) essa probabilidade é independente do contexto de execução e (2) não tem em conta experiência passada diagnóstico no cálculo dessa probabilidade. Nesta tese apresentamos uma nova abordagem, denominada de *NFGE*, com o objetivo de mitigar estas limitações. A resolução para a primeira limitação passou por generalizar tanto a abstração do sistema bem como generalizar o *SFL* para usar a informação sobre o estado do sistema no processo de diagnóstico. Para resolver a segunda limitação, propusemos abordagem baseada numa técnica denominada de Kernel Density Estimates que utiliza observações de *feedback* para modelar as *goodnesses* dos componentes como uma função não linear do estado do sistema. Avaliamos a abordagem com dois casos de estudo, nos quais observamos erros de estimativa menores, aumentando consequentemente a qualidade do diagnóstico.

```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>+++.>>
+++++++++++++++.-.++++++.---------------.+++++++++.++++++.-.[>]<[[-]<]
```

Contents

```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>>----.
>+++++++++.++++++++++.+.<<<<.>>>>-----.---------.<<<<.>>++++++.>>+++.-
-.++++++++++++++.---.-------------.++++++++++++++.[>]<[[-]<]-<>+>>>+><
```

List of Figures

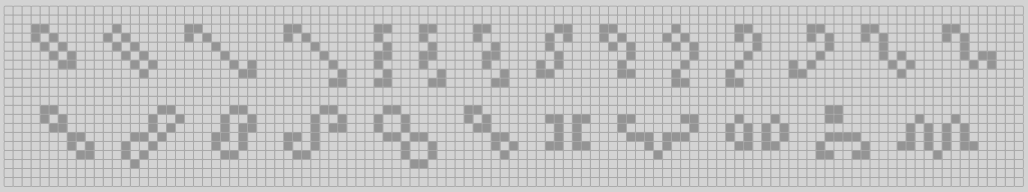
```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>>----.
>+++++++++.++++++++++.+.<<<<.>>>>-----.---------.<<<<.>>+.>>++++++.---
--.++++++++.+++.---------.+++++++++++.------------.+++++.++++++.[>]<[[
-]<]>><+>>>-<+<<-<->->+>><>>+>+<<<<>>-++<>+<++>-<>+>--+>>>>-<--<>++><>
```

List of Algorithms

```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>>----.
>+++++++++.++++++++++.+.<<<<.>>>>-----.---------.<<<<.>>+.>>---.++++++
+++++++++.---.-.+++++++++++.------------.++++++.[>]<[[-]<]<->-+-<->->+
```

List of Acronyms

ADL	Architecture Definition Language
CPU	Central Processing Unit
DCC	Dynamic Code Coverage
GUI	Graphical User Interface
HS	Hitting Set
KDE	Kernel Density Estimate
KNN	K-Nearest Neighbors
MAPE-K	Monitor, Analyze, Plan, Execute and Knowledge
MBD	Model-based Diagnosis
MHS^2	Map-Reduce Heuristic-driven Search for Minimal Hitting Sets
MHS	Minimal Hitting Set
MLE	Maximum Likelihood Estimation
NFGE	Non-linear Feedback-based Goodness Estimate
SFL	Spectrum-based Fault Localization
TMR	Triple Modular Redundancy



```
-[------->+<]>.+[--->+<]>.++++++.--.---.-----------.--[--->+<]>-.+[->+
++<]>+.-[--->+<]>--.-----------.++++++.-.-><>>++>>+<-->-<<+<>>+>-<<-++
->--++<>+-<>+><>-<><-+-<+>><>---->><->+--<<>->->><<--+<<+-<><++-<+<<-<
```

1 Introduction

Modern society is increasingly dependent on technology and, with the appearance of low-cost computing environments, this dependency has experienced an exponential growth. Within a half-century span, computers evolved from a state where they were only able to interact with nearby systems to the point where they can communicate within a matter of seconds despite their distance [Allan, 2001, Polsson, 2015]. Furthermore, with the availability of low-cost high-speed interconnections, software systems grew to global scales and gradually infiltrated most aspects of the modern life style.

The rapid growth of software systems in terms of both size and number happens mainly due to the fact that computers are an extremely versatile tool, which can be used to more efficiently solve a large variety of tasks in different domains. One side effect of the large scope of software systems is the increase in complexity [Horn, 2001]. The rise in complexity almost unavoidably leads to a growing number of bugs which, in turn, can eventually cause errors/failures [Salehie and Tahvildari, 2005].

When unexpected behavior is observed, developers need to identify the root cause(s) that made the system deviate from its intended behavior. This task (also known as software debugging, fault localization, or error diagnosis) is the most time-intensive and expensive phase of the software development cycle [Hailpern and Santhanam, 2002], and has been a concern since the beginning of computer history[1]. In fact, it was estimated that the global cost of debugging software has risen to \$312 billion annually[2]. To put this value into perspective, that is equivalent to the cost of 729 "Airbus A380" aircrafts[3].

The high cost of debugging software is related to the fact that the process of detecting, locating and fixing faults is both non-trivial and error-prone. It was estimated that a

[1] In 1946, the term bug was first used in the scope of computer science by Grace Hopper to document a problem in the Mark II computer. In fact, the source of the problem was an actual bug (more specifically a moth) that was trapped in a relay, impeding its correct functioning.

[2] University of Cambridge, "Financial content: Cambridge University study states software bugs cost economy \$312 billion per year", http://www.prweb.com/releases/2013/1/prweb10298185.htm, accessed October 06, 2015.

[3] \$428 million per unit, https://en.wikipedia.org/wiki/Airbus_A380, accessed October 06, 2015.

great share of the development resources, easily ranging from 50% to 70%, is normally assigned to software testing and diagnosis [Hailpern and Santhanam, 2002] and that even experienced developers are wrong almost 90% of the time in their initial guesses about the the faults' locations [Ko and Myers, 2008].

To ease this process several diagnostic tools have been proposed (see Chapter 6 for examples). Despite the improvement that such development-time diagnostic-related tools represent, it remains practically impossible to create faultless systems. Acknowledging this fact, the concept of self-healing system has been proposed. Self-healing systems have the goal of improving their dependability[4] at run-time while reducing human intervention by employing mechanisms that either prevent or solve eventual errors [Ghosh et al., 2007]. In [Psaier and Dustdar, 2011], the authors summarize the concept of self-healing system as follows (Figure 1.1):

> *"The reason for enhancing a system with self-healing properties is to achieve continuous availability. Compensating the dynamics of a running system, self-healing techniques momentarily are in charge of the maintenance of health. Enduring continuity includes resilience against intended, necessary adaptations and unintentional, arbitrary behavior. Self-healing implementations work by detecting disruptions, diagnosing failure root cause and deriving a remedy, and recovering with a sound strategy. Additionally, to the accuracy of the essential sensor and actuator infrastructure, the success depends on timely detection of system misbehavior. This is only possible by continuously analyzing the sensed data as well as observing the results of necessary adaptation actions. The system design leads to a control loop similar assembly. An environment dependent and preferably adaptable set of policies support remedy decisions. Possible policies include simple sets of event dependent instructions but also extended AI estimations supporting the resolution of previously unknown faults."*

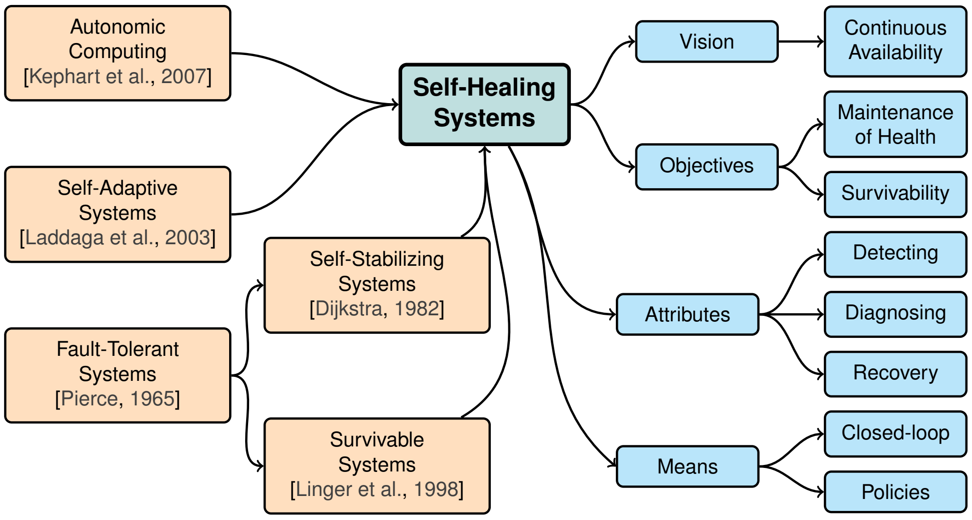

Autonomic Computing [Kephart et al., 2007]
Self-Adaptive Systems [Laddaga et al., 2003]
Fault-Tolerant Systems [Pierce, 1965]
Self-Stabilizing Systems [Dijkstra, 1982]
Survivable Systems [Linger et al., 1998]
Self-Healing Systems
Vision
Objectives
Attributes
Means
Continuous Availability
Maintenance of Health
Survivability
Detecting
Diagnosing
Recovery
Closed-loop
Policies

Figure 1.1: Relations and properties of self-healing research (adapted from [Psaier and Dustdar, 2011])

[4]According to [Avizienis et al., 2004], dependability is an integrating concept that encompasses availability (readiness for correct service), reliability (continuity of correct service), safety (absence of catastrophic consequences on the users and the environment), integrity (absence of improper systems alterations) and maintainability (ability to undergo modifications and repairs).

The self-healing concept is in fact a specialization of a broader concept, called autonomic computing [Kephart and Chess, 2003]. To implement an autonomic system, *IBM* suggests a reference model for autonomic control loops [Kephart et al., 2007], referred to as the **Monitor, Analyze, Plan, Execute and Knowledge (*MAPE-K*)** control loop and is depicted in Figure 1.2. In the *MAPE-K* autonomic loop, the managed resource represents any software or hardware resource that is given autonomic behavior by coupling it with an autonomic manager. In a practice, the managed resource can be, for instance, a web server, a database, an operating system, a cluster of machines, a hard drive, a wired or wireless network, *etc.*.

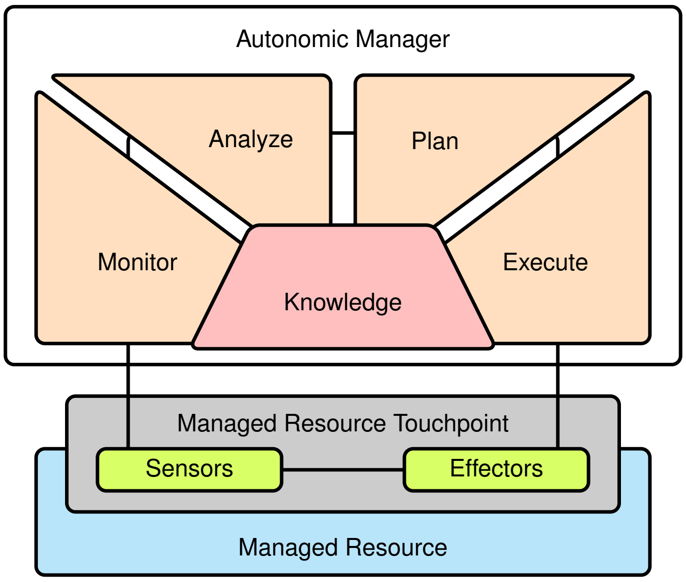


Figure 1.2: *MAPE-K* control loop

In the context of self-healing systems, the *monitor* component determines whether the system is in degraded or erroneous state, through the information collected by the managed resource's sensors (also known as probes or gauges [Garlan et al., 2001]). The *analyzer* component is responsible for pinpointing the probable source(s) of system errors. The *plan* component creates a repair plan targeted at returning the system to an operational state. The *executor* component implements the repair plan, by using a set of system effectors that carry out changes to the managed resource. The change implemented by the *effectors* can be coarse-grained (*e.g.*, adding or removing servers to a web server cluster [Garlan et al., 2004]) or fine-grained (*e.g.*, changing configuration parameters in a web server) [Bigus et al., 2002]. The *knowledge* component collects the knowledge produced by all the aforementioned components and stores it for future use.

In this thesis, our focus is on the analyzer component of the *MAPE-K* control loop. Our goal is to create an automatic diagnostic framework for run-time systems. Even though several diagnostic approaches have been proposed the large majority is either too specific to a particular application (*e.g.*, [Chao et al., 2004, Mohammadi and Hashtrudi-Zad, 2007, Kasick et al., 2010, Tan et al., 2010, Shvachko et al., 2010]) or too costly (*e.g.*, [Reiter, 1987, De Kleer and Williams, 1987, Mayer and Stumptner, 2003, Wotawa et al., 2002]). In contrast to the existent approaches, we aim at creating a diagnostic framework meeting the following criteria:

- It must be “general purpose”:
  - It must be usable in a large variety of systems.
  - It must be possible to add new components to the managed resource without altering the diagnostic framework.
  - It must handle different instrumentation granularities.
- It must be “scalable”:
  - It must be able to handle systems with large number of components.
- It must be “accurate”

To achieve our goal while meeting the aforementioned criteria, we improve over a development-time diagnostic technique called **Spectrum-based Fault Localization (*SFL*)**. *SFL* uses a high-level abstraction of the system under analysis, making it, in principle, usable in a large diversity of scenarios. The only requirements to use *SFL* in a real-world scenario are that (1) the system’s activity must be divisible into transactions, (2) the correctness of each transaction must be evaluable, (3) the components’ activations must be observable and (4) it must be possible to associate the components’ activity with the corresponding transactions.

The high-level abstraction implies that a large amount of the diagnostic complexity is traded-off for accuracy. Even though *SFL* is less accurate than some existing diagnostic approaches (*e.g.*, [Reiter, 1987, De Kleer and Williams, 1987, Mayer and Stumptner, 2003, Wotawa et al., 2002]), it is able to scale to large systems where heavier approaches are not usable. Furthermore, given enough diversity in the observations, *SFL* tends to be accurate enough for practical purposes [Santelices et al., 2009, Abreu et al., 2009b].

In the remainder of this chapter we further discuss the scope of our work. In Section 1.1 (page 4), we describe the diagnostic problem. In Section 1.2 (page 9), we present the state-of-the-art *SFL* approach for development-time diagnosis. In Section 1.3 (page 19), we discuss our research goals. In Section 1.4 (page 23), we present the origin of chapters. Finally, in Section 1.5 (page 24), we present the thesis outline.

1.1 Diagnostic Problem

In general terms, a diagnostic problem occurs whenever the behavior of a particular system (natural or artificial) deviates from the expected behavior [Reiter, 1987, De Kleer and Williams, 1992]. The challenge consists in finding the true root causes of such abnormal behavior.

In our work we use the taxonomy proposed by [Avizienis et al., 2004]:

- An **error** is an incorrect system state that may cause a failure.
- A **failure**, is the observable manifestation of an error: an error becomes a failure when it propagates to the system’s output.
- A **fault/bug** is the cause of an error in the system.

```
def f_to_c(temp_f):
    return (temp_f - 32) * 5 // 9

def is_freezing_c(temp_c):
    return temp_c <= 0

def is_freezing_f(temp_f):
    return is_freezing_c(f_to_c(temp_f))
```

Figure 1.3: Bug example

To illustrate these concepts, take for instance the three Python functions presented in Figure 1.3. The purpose of function f_to_c is to convert a temperature from *Fahrenheit* degrees to *Celsius*. Since the whole operation is performed using integer arithmetic, the implementation is faulty and may compute erroneous results. Function $is\_freezing\_f$, which is implemented by daisy chaining functions f_to_c and $is\_freezing\_c$, should return $True$ if the input value (in *Fahrenheit*) is below water's freezing point and $False$ otherwise.

	f_to_c		$is\_freezing\_f$		
Input	Expected	Observed	Expected	Observed	Outcome
$31°F$	$-0.55_5°C$	$-1°C$	$True$	$True$	Error
$32°F$	$0°C$	$0°C$	$True$	$True$	Nominal
$33°F$	$0.55_5°C$	$0°C$	$False$	$True$	Failure

Figure 1.4: $is\_freezing\_f$ function trace

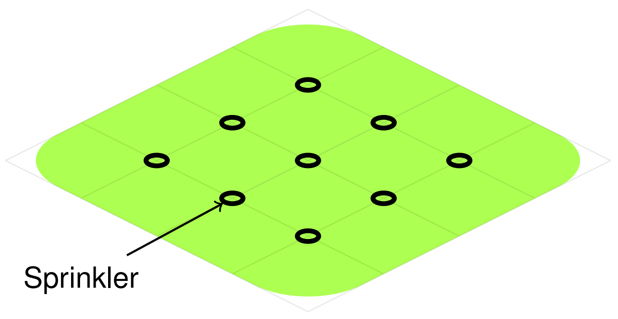
By analyzing Figure 1.4, we can see that for $32°F$ the function $is\_freezing\_f$ not only works as expected but also does not activate the fault (*i.e.*, the result of f_to_c is correct). For $31°F$, despite activating the fault (*i.e.*, the result of f_to_c is not correct), the aggregate result of both functions is correct due to the fact that function $is\_freezing\_c$ masks the error. Finally, for $33°F$ the fault is activated and the error propagates to the system output, delivering an incorrect result.

A prerequisite to diagnose a system is that the occurrence of errors/failures is detected. The process of observing the system state and deciding whether or not it satisfies the system's specification is known as the oracle problem.

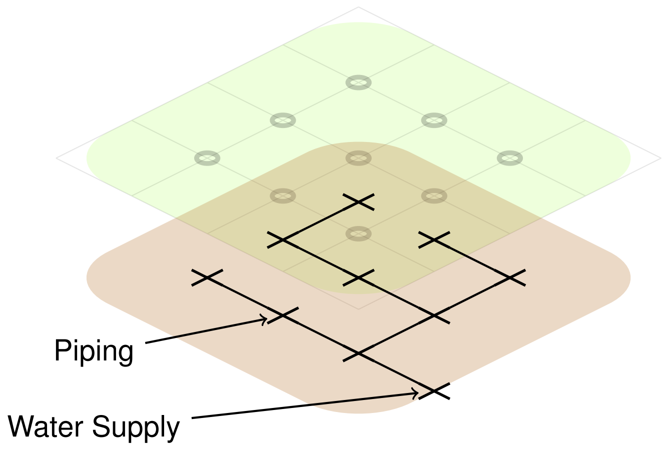
The approaches to solve the diagnostic problem can be broadly divided in two groups: heuristic and model based diagnosis (also known as diagnosis from first principles).

1.1.1 Heuristic-based Diagnosis

Heuristic-based diagnosis approaches are focused on encoding expert knowledge generated by previous diagnostic experience to more efficiently address future diagnostic problems. In such diagnostic systems, the diagnostic reasoning is greatly based upon the observed error symptoms and the possible (known) solutions to such symptoms. A consequence of this type of reasoning is that the structure of the system under analysis is only weakly represented, if present at all. While such diagnostic systems are effective in diagnosing known abnormalities, they tend to fail whenever new error symptoms emerge.



(a) Sprinklers



(b) Piping

Figure 1.5: Irrigation system example

As an illustrative example, consider an irrigation system with the additional goal of guaranteeing that irrigation problems are automatically detected and diagnosed. For the example shown in Figure 1.5a, one could use peer-similarity to accomplish the self-diagnosis goal. The usage of peer-similarity requires that all components in the system behave similarly with regard to a set of metrics and the presence of outlier metric values implies the occurrence of errors in the corresponding components [Kasick et al., 2010, Shvachko et al., 2010, Tan et al., 2010]. Assuming that, for instance, the water pressure or consumption for each sprinkler is observable, substantial variations in such metrics among the system's sprinklers would signal a sprinkler failure.

Even though this approach would most likely perform well in the described system, in a system where the sprinklers operate at different pressures or have different water consumption rates the outlier detection would be too inaccurate. Furthermore, if all the sprinklers in the system fail similarly, this particular approach may fail to detect and, consequently, to diagnose the errors.

Another problem with the aforementioned approach is related to the fact that, to meet the similarity requirement, the abstraction of the system neglects the existence of important components and may result in poor diagnostic quality. Considering the fact that the sprinklers are fed by a piping system, as depicted in Figure 1.5b, the observation of an error symptom in a sprinkler does not necessarily mean that the problem occurred there. In practice, the symptom of low water pressure may be due to water shortage, piping failure, and/or sprinkler failure.

Finally, consider a situation in which the state of the sprinklers is not directly observable but instead the system is equipped with humidity sensors, placed at arbitrary positions in the field, as shown in Figure 1.6. As the state of the system's components is not available, the peer-similarity technique is no longer usable.

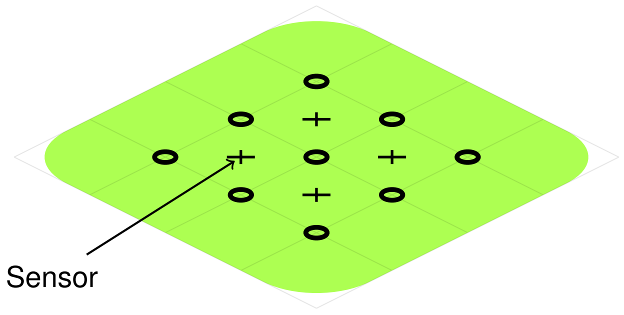


Figure 1.6: Irrigation system example – Sensors

Another example of this type of diagnostic approach is the so-called medical diagnostic guidelines. Even though such guidelines effectively enhance both diagnostic accuracy and efficiency for common diseases, they fail in the presence of unknown symptoms/diseases (*e.g.*, a patient with a green glowing eye).

Even though apparently unrelated to the computer science domain, the previous examples show the potential problems related to heuristic diagnostic techniques. Establishing a parallel with software, we see that software systems are mostly composed of heterogeneous sets of components with, almost inevitably, different degrees of state observability thus strongly limiting the scope of application of existing diagnostic heuristics over different systems [Salehie and Tahvildari, 2005].

1.1.2 Model-based Diagnosis

Model-based Diagnosis (*MBD*) approaches focus on encoding the structure and expected behavior of the system under analysis which, together with observations of the system's behavior, are used to perform the diagnostic reasoning [De Kleer and Williams, 1987, Reiter, 1987].

Definition 1 (Model-based Diagnostic System)**.** *A model-based diagnostic system DS is defined as the triple $DS = (SD, COMPS, OBS)$, where:*

- SD *is a propositional theory describing the behavior of the system*
- $COMPS = \{c_1, \ldots, c_M\}$ *is a set of components in* SD
- OBS *is a set of observable variables in* SD

The existence of such a model enables the diagnostic system to successfully cope (from a theoretical point-of-view) with new error symptoms (such as the patient with the glowing eye). Whenever the observed system behavior for a particular scenario (*i.e.*, a specific assignment over variables in OBS) conflicts with the behavior predicted by the model SD, the diagnostic reasoning revolves around finding sets of components that, by assuming their faultiness, would explain the erroneous behavior.

To guarantee generality, *MBD* algorithms reason in terms of conflicts. Informally, a conflict represents a set of components that cannot be simultaneously healthy to explain the observed erroneous behavior.

Definition 2 (h-literal)**.** *An h-literal, denotes the component's health. The positive h-literal h corresponds to a healthy component whereas, the negative h-literal $\neg h$ corresponds to an unhealthy one.*

Definition 3 (Conflict)**.** *Let $\boldsymbol{HL}^{+}(C) = \bigwedge_{j \in C} h_j$ be the conjunction of positive h-literals for a set of components $C \subseteq COMPS$ and obs an observation term over variables in OBS. C is a conflict for (DS, obs) if and only if:*

$$SD \wedge obs \wedge \boldsymbol{HL}^{+}(C) \tag{1.1}$$

is inconsistent.

In other words, a conflict is a set of components that cannot be simultaneously healthy for the observed erroneous behavior to occur.

Definition 4 (Diagnostic Candidate)**.** *Let $C \subseteq COMPS$ be a set of components. We define $d(C)$ to be the conjunction:*

$$\Big(\bigwedge_{m \in C} \neg h_m \Big) \wedge \Big(\bigwedge_{m \in COMPS \setminus C} h_m \Big) \tag{1.2}$$

Given an observation term obs over variables in OBS, a diagnostic candidate for DS is a conjunction $d(C)$ such that:

$$SD \wedge obs \wedge d(C) \tag{1.3}$$

is consistent.

In the remainder we refer to $d(C)$ simply as d, which we identify with the set C.

In this context, a diagnostic candidate is thereby a set of components that is conjectured to be unhealthy, resolving all conflicts entailed by $SD \wedge obs$.

A problem with the above definition is related to the fact that a candidate d containing all of the system's components (*i.e.*, $d = COMPS$) always resolves every conflict by $SD \wedge obs$. Intuitively, this is equivalent to saying that the whole system is faulty which, in practice, is not a very helpful conclusion. To apply the concept of a diagnostic candidate to real systems with success, one must refine the definition so that the candidate contains the minimum number of components while still solving all the conflicts.

Definition 5 (Minimal Diagnostic Candidate)**.** *A candidate d is minimal if and only if $\nexists_{d'} : d' \subset d$ such that d' is a diagnostic candidate.*

The end result of the diagnostic reasoning is the diagnostic report. A diagnostic report features a set of explanations for the erroneous behavior (*i.e.*, the diagnostic candidates) as well as a measure of how likely each explanation is. The process of calculating a diagnostic report can be broadly divided in two stages, as depicted in Figure 1.7.

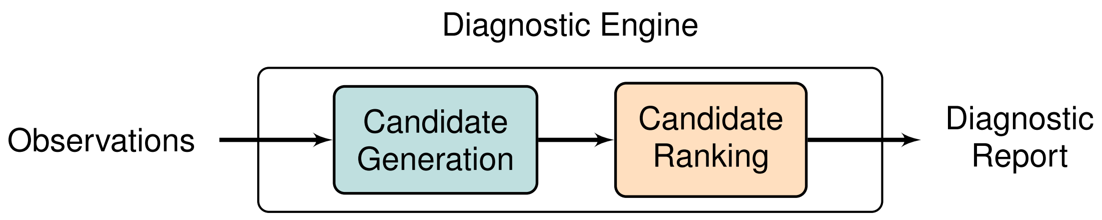


Figure 1.7: Diagnostic process

Definition 6 (Diagnostic Report)**.** *A diagnostic report* $D = (d_1, \ldots, d_k, \ldots, d_K)$ *is an ordered set of* K *diagnostic candidates, such that:*

$$\forall_{d_k \in D} : \boldsymbol{Pr}(d_k \mid obs) \geq \boldsymbol{Pr}(d_{k+1} \mid obs) \tag{1.4}$$

1.2 Spectrum-based Fault Localization

A limitation of traditional *MBD* approaches is the necessity of a detailed system model which, in practice and due to the large complexity of modern computer systems, normally entails a large modeling effort/cost, thus narrowing its range of application [Pietersma et al., 2006, Horn, 2001]. To overcome the complexity of creating precise system models, **Spectrum-based Fault Localization (*SFL*)**[5] approaches were proposed [Abreu et al., 2009a, De Kleer, 2009, Casanova et al., 2013]. Instead of relying on a fine grained model of the system (*i.e.*, $SD \in DS$) to generate conflict sets, *SFL* infer conflicts by performing a dynamic analysis of the system.

To apply *SFL* to a system, it must be abstracted in terms of two general concepts: component, and transaction. A component is an element of the system that, for diagnostic purposes, is considered to be atomic[6]. A transaction is a set of component activations that (1) share a common goal, and (2) the correctness of the provided service can be verified. The error detection mechanism, from a *SFL* perspective, is treated as a black box.

To gather all the required information to perform diagnosis, the system under analysis must be instrumented (see Section 6.3). The instrumentation's output, commonly known as spectrum [Harrold et al., 1998], is defined as the pair (A, e) (Figure 1.8), where:

- A (**activity matrix**) encodes the involvement of components in transactions.
- e (**error vector**) encodes the correctness of each individual transaction.

[5]For simplicity and unless stated otherwise, we use the acronym *SFL* to refer to a particular *SFL* approach known as Spectrum-based Reasoning for Fault Localization. In Section 6.4 (page 89) an alternative and more common *SFL* approach is presented.

[6]In a software environment, a component can be for instance a statement, a function, a class, a service, *etc.*.

$$\begin{array}{cc} \text{activity matrix} & \begin{array}{c}\text{error}\\ \text{vector}\end{array} \\ \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1M} \\ A_{21} & A_{22} & \cdots & A_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ A_{N1} & A_{N2} & \cdots & A_{NM} \end{bmatrix} & \begin{bmatrix} e_1 \\ e_2 \\ \vdots \\ e_N \end{bmatrix} \end{array}$$

Figure 1.8: Spectrum with M components and N transactions

Even though several types of spectra exist, the most commonly used is called a hit spectrum [Harrold et al., 1998, Yilmaz et al., 2008, Santelices et al., 2009].

Definition 7 (Hit Spectrum)**.** *The hit spectrum abstraction encodes the components' activity in terms of hit/not hit and the transactions' correctness in terms of pass/fail. Using the hit spectrum abstraction, A and e are defined as:*

$$A_{ij} = \begin{cases} 1, & \textit{if component } j \textit{ was involved in transaction } i \\ 0, & \textit{otherwise} \end{cases} \tag{1.5}$$

$$e_i = \begin{cases} 1, & \textit{if transaction } i \textit{ failed} \\ 0, & \textit{otherwise} \end{cases} \tag{1.6}$$

For convenience, in this thesis we may treat A_i as a set containing the indices of all components involved in transaction i. Formally, A_i can also be defined as:

$$A_i = \{j \mid \text{if component } j \text{ was involved in transaction } i\} \tag{1.7}$$

Since both forms encode the same information, we can use the two forms interchangeably. Whenever we apply set operations to A, we are implicitly using the set form. In Figure 1.9 an example hit spectrum in both forms is presented.

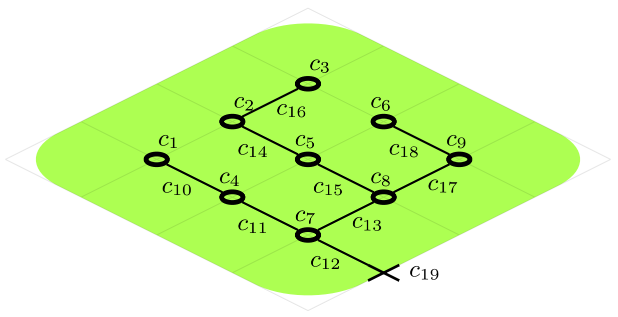
i	A			e
	c_1	c_2	c_3	
1	1	1	0	1
2	1	0	1	1
3	1	1	1	0

(a) Matrix form

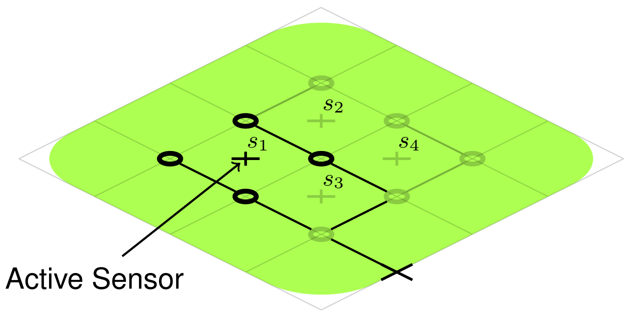
i	A	e
1	$\{1,2\}$	1
2	$\{1,3\}$	1
3	$\{1,2,3\}$	0

(b) Set form

Figure 1.9: Hit spectrum example



(a) Components



(b) Transaction

i	A																			e
	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9	c_{10}	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}	c_{16}	c_{17}	c_{18}	c_{19}	
1	●	●	.	●	●	.	.	.	.	●	●	●	●	●	●	.	.	.	●	●
2	.	●	●	.	●	●	.	.	.	.	.	●	●	●	●	●	●	●	●	.
3	.	.	.	●	●	.	●	●	.	.	●	●	●	.	●	.	.	.	●	.
4	.	.	.	.	●	●	.	●	●	.	.	●	●	.	●	.	●	●	●	●

(c) Hit spectrum (0's and 1's were replaced by "." and "●" for readability)

Figure 1.10: Irrigation system example – *SFL*

To illustrate how *SFL* can be used in an arbitrary system, consider again the example in Figure 1.6. For this system, the set of all components is composed of 19 elements (Figure 1.10a): 9 sprinklers (c_1 through c_9), 9 piping elements (c_{10} through c_{18}) and the water supply (c_{19}). There are 4 different transaction types (one for each sensor). Each of those transactions consists of the activation of all the surrounding sprinklers as well as the piping elements used to by those sprinklers, as depicted in Figure 1.10b. A possible approach to evaluate the success of each transaction would be to compare the humidity value obtained by the corresponding sensor to an arbitrary threshold interval. If the humidity on the ground is either too high or too low, the transaction fails.

Consider a scenario where, for the described system, sensors s_1 and s_4 (see Figure 1.10b) detect an incorrect humidity level whereas, the remaining sensors, detect a corrected humidity level. The spectrum corresponding to this scenario is depicted in Figure 1.10c. The spectrum contains 4 transactions (each transaction i corresponds to the sensor s_i) and both transactions 1 and 4 are marked as erroneous. Additionally, for each transaction, all the sprinklers adjacent to the corresponding sensor as well as the piping elements needed to feed such sprinklers are marked as active.

In the next sub-sections we present the relevant details of existent *SFL* algorithms to solve both the candidate generation and ranking problems.

1.2.1 Candidate Generation

In a naïve approach, the candidate generation problem can be addressed by computing the power set of all components in the system ($\mathcal{P}(COMPS)$). However, as $|\{\mathcal{P}(COMPS)\}| = 2^{\{|COMPS|\}}$, this approach becomes quickly ineffective. In practice, rather than iterating over *all* possible sets just to find that most are not minimal or even consistent with the observations, search algorithms are typically used to only consider sets that meet the minimal candidate criteria (see Definition 5).

Despite the advantage of only computing minimal candidates, the problem is still remarkably hard. The calculation of minimal candidates, conceptually referred to as hitting sets, is a problem equivalent to the Minimal Hitting Set problem [Reiter, 1987], which is known to be NP-Hard [Garey and Johnson, 1990]. The formal definition of the Minimal Hitting Set problem goes as follows:

Definition 8 (Hitting Set)**.** *Given a set U of M elements (called the universe) and a collection S of N sets, a set d is said to be a* ***Hitting Set (HS)*** *of (U, S) if and only if:*

$$\boldsymbol{HS}(U, S, d) := d \subseteq U \wedge \left(\forall_{s \in S} : d \cap s \neq \emptyset\right) \tag{1.8}$$

Corollary 1.2.1. *If $\nexists_{s \in S} : s \cap U = \emptyset$, U is a HS of S.*

Corollary 1.2.2. *If $\exists_{s \in S} : s \cap U = \emptyset$, $\boldsymbol{HS}(U, S, d)$ never holds.*

Corollary 1.2.3. *$d = \emptyset$ is a hitting for $S = \emptyset$.*

Corollary 1.2.4. *$\boldsymbol{HS}(U, S, d)$ is associative:*

$$\boldsymbol{HS}(U, S, d) \wedge \boldsymbol{HS}(U', S', d') \implies \boldsymbol{HS}(U \cup U', S \cup S', d \cup d') \tag{1.9}$$

Definition 9 (Minimal Hitting Set)**.** *A set d is a* ***Minimal Hitting Set (MHS)*** *of (U, S) if and only if:*

$$\boldsymbol{MHS}(U, S, d) := \boldsymbol{HS}(U, S, d) \wedge \left(\nexists_{d' \subset d} : \boldsymbol{HS}(U, S, d')\right) \tag{1.10}$$

i.e.*, d is a HS and no proper subset of d is a HS.*

There may be several *MHSs* d_k for (U, S), which constitute the *MHS* collection D. The *MHS* problem consists thereby in computing D for a particular pair (U, S).

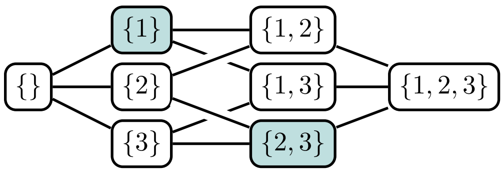
Putting this definition in terms of the candidate generation problem, the set of all components of the system ($COMPS$) is equivalent to the set U, whereas the set of all conflicts entailed by $SD \wedge obs$ is equivalent to S. A *MHS* for all the conflicts entailed by $SD \wedge obs$ is, in fact, a diagnostic candidate for $SD \wedge obs$ [Reiter, 1987].

Under the *SFL* abstraction, a conflict occurs whenever a failed transaction exists in the spectrum. The elements of the conflict set are the components activated in such transaction. Intuitively, since the transaction failed, it follows that at least one of the components must not be healthy for the erroneous behavior to occur.

As an example, consider the hit spectrum in Figure 1.11a for which all 2^M possible (but not necessarily valid) candidates (*i.e.*, $\mathcal{P}(COMPS)$) are presented in Figure 1.11b. For this particular spectrum, two minimal candidates/*MHSs* exist: $\{1\}$ and $\{2, 3\}$. Even though the set $\{1, 2, 3\}$ is also *HS*, it is not minimal as it can be subsumed either by $\{1\}$ or $\{2, 3\}$.

i	A	e
1	$\{1, 2\}$	1
2	$\{1, 3\}$	1
3	$\{1, 2, 3\}$	0

(a) Hit spectrum



(b) Hasse diagram of $\mathcal{P}(\{1, 2, 3\})$

Figure 1.11: Example

Being a NP-hard problem, the usage of exhaustive search algorithms (*e.g.*, [Reiter, 1987, Wotawa, 2001]), is prohibitive for most real-world problems. In order to solve the candidate generation problem in a reasonable amount of time, approaches that relax the strict minimality[7] constraint have been proposed [Abreu and Van Gemund, 2009, De Kleer and Williams, 1992, Feldman et al., 2008].

A simplified[8] version of the state-of-the-art *SFL* candidate generation algorithm (called *Staccato*) [Abreu and Van Gemund, 2009] is presented in Algorithm 1.1 and Figure 1.12. The algorithm works in a divide and conquer fashion by, at each stage of its execution, performing one of two different tasks (lines 2, 3, and 4 or 6–8), depending on whether the set d is a *HS*. As we shall shortly see, due to the algorithm's divide and conquer nature, d is a *HS* whenever $S = \emptyset$.

The first task, which is triggered whenever d is not a *HS* (line 1), aims at dividing the initial problem in smaller sub-problems. This goal is achieved by iteratively selecting an element $j \in U$ from a heuristically[9] ordered set (line 2) and creating a temporary collection S' containing all the sets $s \in S : j \in s$, *i.e.*, the sets hit by $\{j\}$ (line 3). Finally, the algorithm makes a recursive call to solve the sub-problem $S \setminus S'$ with set $d \cup \{j\}$ (line 4).

The second task, which occurs whenever d is a *HS* (line 5), aims at collecting *HSs* while guaranteeing that no *HS* in D has a proper subset also contained in D. The first step in this task is to check if d is minimal (line 6) with regard to the already discovered *MHS* collection D. If d is minimal, all super-sets of d in D are purged (line 7) and, finally, d is added to D (line 8).

[7]We use the term minimal in a more liberal way due to mentioned relaxation. A candidate d is said to be minimal if no other calculated candidate is contained in d.

[8]For simplicity, the cutoff conditions were omitted.

[9]The details of chosen heuristic are presented in Section 2.4.1. For now assume that the order is arbitrary and that, for every possible ordering, the algorithm computes the same result.

Algorithm 1.1 *Staccato*

Inputs: $(U, S, d = \emptyset, D = \emptyset)$

Output: Minimal hitting set collection D

```
1 if $S \neq \emptyset$ then                                   # divide task
2     for $j \in \boldsymbol{Rank}(U, S)$ do
3         $S' \leftarrow \{s \mid s \in S \wedge j \in s\}$
4         $D \leftarrow \boldsymbol{Staccato}(U \setminus \{j\}, S \setminus S', d \cup \{j\}, D)$
5 else                                                          # conquer task
6     if $\nexists_{d' \in D} : d' \subseteq d$ then
7         $D \leftarrow D \setminus \{d' \mid d' \in D \wedge d \subseteq d'\}$
8         $D \leftarrow D \cup \{d\}$
9 return $D$
```

To illustrate how *Staccato* works, consider the example in Figure 1.13[10] which represents a possible search tree for *Staccato* with $U = \{1, 2, 3\}$ and $S = \{\{1, 2\}, \{1, 3\}\}$ (Figure 1.11a). Each node in the search tree represents a call to the function (all the parameters as well as the return value are encoded as a table). Leaf nodes represent function calls for which d is a *HS* whereas intermediate nodes represent calls for which d is not a *HS*.

In the outer call to the algorithm (the leftmost node), as $S \neq \emptyset$, the algorithm performs the divide task. After exploring the sub-tree starting with $d = \{2\}$, the algorithm yields the collection $D = \{\{1, 2\}, \{2, 3\}\}$.

We can see that at this point, if the execution were to be interrupted, $\{1, 2\}$ would be erroneously considered a *MHS*. However, after exploring the sub-tree starting with $d = \{1\}$, the set $\{1, 2\}$ is removed yielding the collection $D = \{\{1\}, \{2, 3\}\}$.

The inspection of the sub-tree starting with $d = \{3\}$ does not make further changes to collection D. On the one hand, the *HS* $\{1, 3\}$ is a proper super-set of $\{1\}$. On the other hand, the *HS* $\{2, 3\}$ is already contained in D.

As expected, the result for this example would be the collection $D = \{\{1\}, \{2, 3\}\}$.

[10]Note that the order of node exploration (and consequently the shape of the tree) was selected for illustrative purposes.

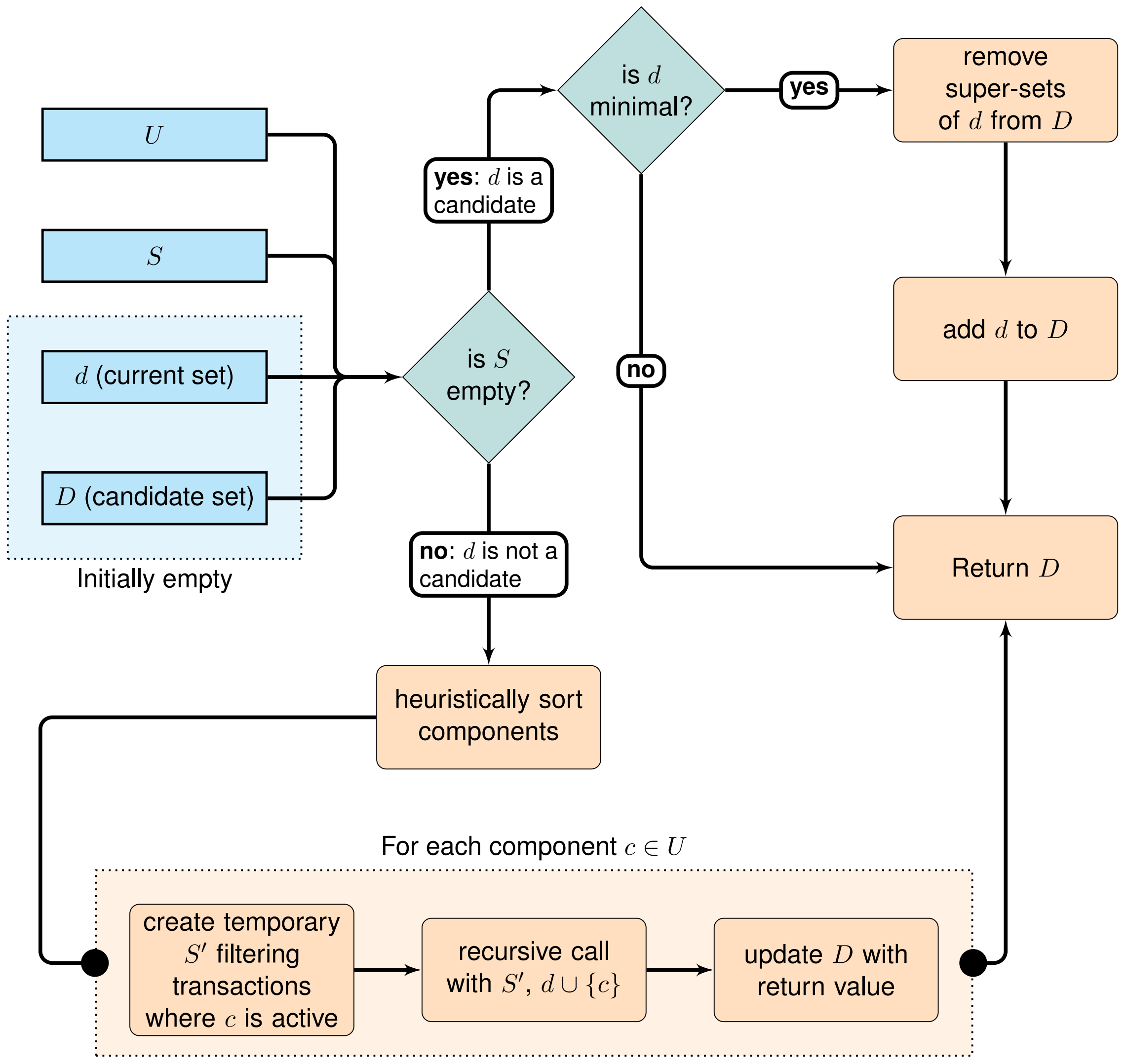

U
S
d (current set)
D (candidate set)
Initially empty
is S empty?
yes: d is a candidate
is d minimal?
yes
remove super-sets of d from D
add d to D
no
Return D
no: d is not a candidate
heuristically sort components
For each component $c \in U$
create temporary S' filtering transactions where c is active
recursive call with S', $d \cup \{c\}$
update D with return value

Figure 1.12: Candidate generation flowchart

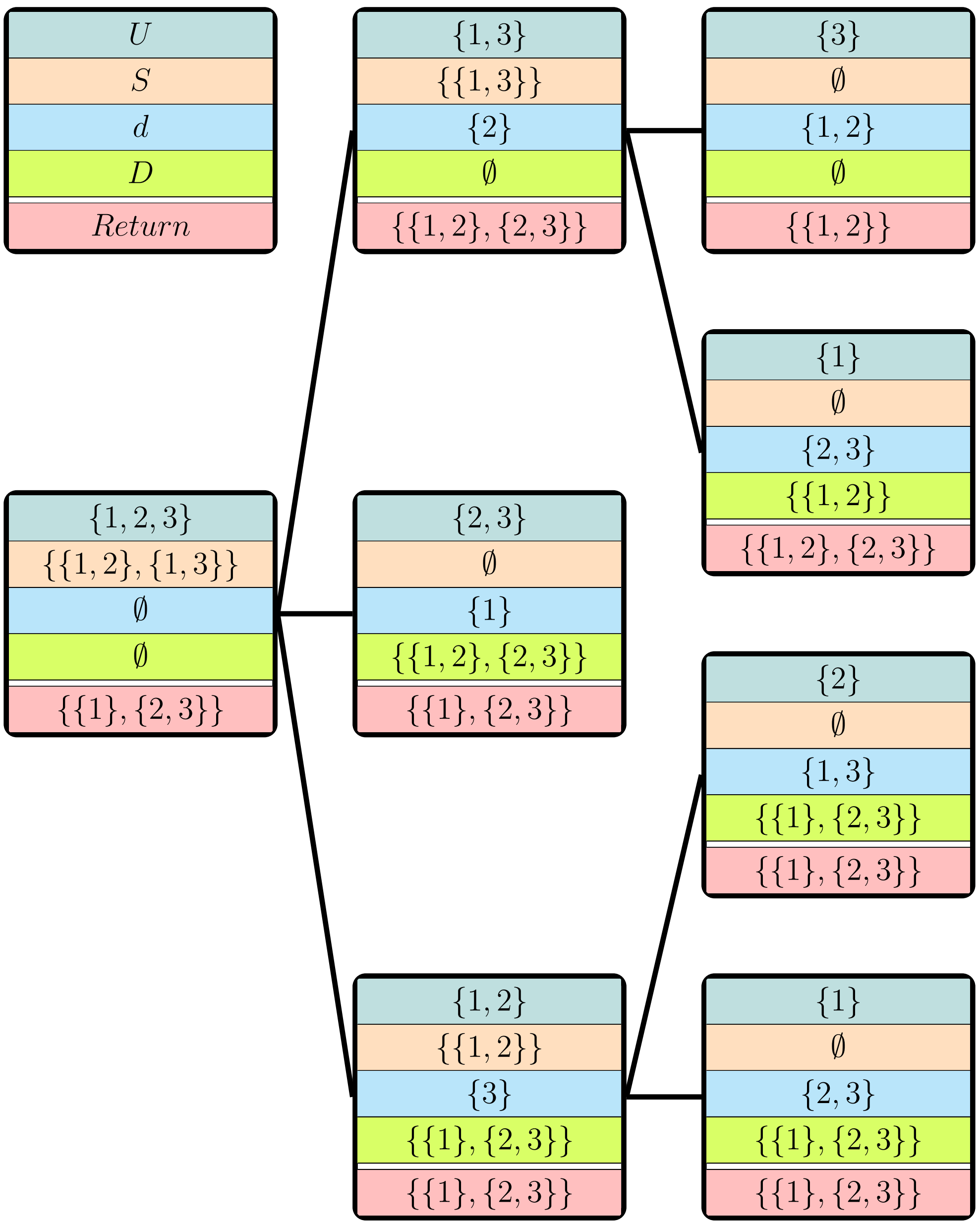


Figure 1.13: Example search tree (pre-order traversal, from top to bottom)

1.2.2 Candidate Ranking

The candidate ranking problem is normally addressed using a naïve Bayes classifier [Abreu et al., 2009a, De Kleer, 2009]. Concretely, the posterior probability of each candidate $d \in D$ given the observed run-time behavior ($\boldsymbol{Pr}(d \mid obs)$) is calculated assuming conditional independence throughout the process (Figure 1.14).

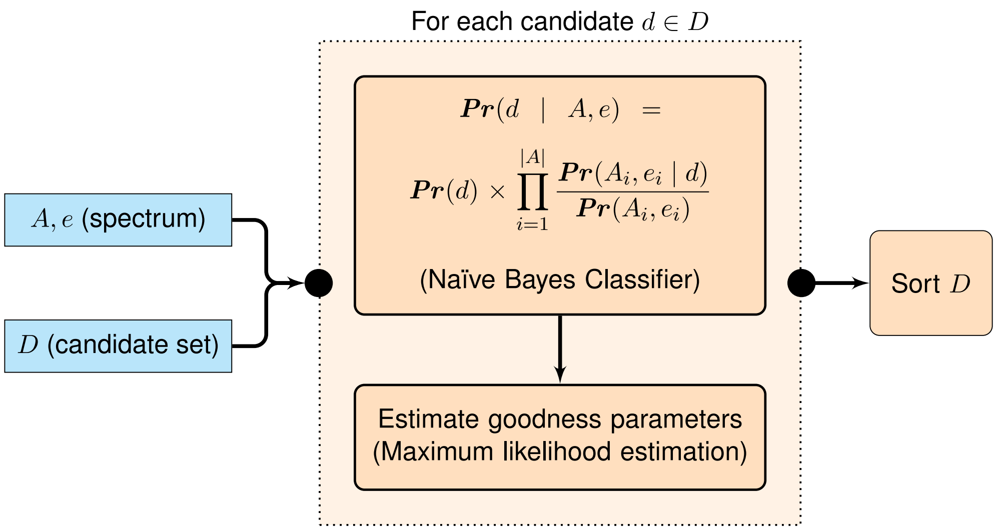

For each candidate $d \in D$
A, e (spectrum)
D (candidate set)
$\boldsymbol{Pr}(d \mid A, e) = \boldsymbol{Pr}(d) \times \prod_{i=1}^{|A|} \frac{\boldsymbol{Pr}(A_i, e_i \mid d)}{\boldsymbol{Pr}(A_i, e_i)}$
(Naïve Bayes Classifier)
Estimate goodness parameters (Maximum likelihood estimation)
Sort D

Figure 1.14: Candidate ranking flowchart

Under a set of observations, the posterior probabilities are calculated according to Bayes rule as:

$$\boldsymbol{Pr}(d \mid obs) = \boldsymbol{Pr}(d) \times \frac{\boldsymbol{Pr}(obs \mid d)}{\boldsymbol{Pr}(obs)} \tag{1.11}$$

Since for *SFL* $obs = (A, e)$, we can rewrite the previous expression as:

$$\begin{aligned} \boldsymbol{Pr}(d \mid A, e) &= \boldsymbol{Pr}(d) \times \frac{\boldsymbol{Pr}(A, e \mid d)}{\boldsymbol{Pr}(A, e)} \\ &= \boldsymbol{Pr}(d) \times \frac{\prod_{i \in 1..N} \boldsymbol{Pr}(A_i, e_i \mid d)}{\boldsymbol{Pr}(A, e)} \end{aligned} \tag{1.12}$$

The denominator $\boldsymbol{Pr}(A, e)$ is a normalizing term that is identical for all $d \in D$ and is not considered for ranking purposes.

To define $\boldsymbol{Pr}(d)$, let p_j denote the prior probability that a component c_j is at fault[11]. Assuming that components fail independently, the prior probability for a particular candidate $d \in D$ is given by:

$$\boldsymbol{Pr}(d) = \prod_{j \in d} p_j \cdot \prod_{j \in COMPS \setminus d} (1 - p_j) \tag{1.13}$$

[11]The value of p_j is application dependent. In the context of development-time fault localization it is often approximated as $p_j = 1/1000$, *i.e.*, 1 fault for each 1000 lines of code [Carey et al., 1999].

$\boldsymbol{Pr}(d)$ estimates the probability that a candidate, without further evidence, is responsible for the system's malfunction. By using equal values for all p_j it follows that the larger the candidate the smaller its prior probability will be.

In order to bias the prior probability taking run-time information into account, $\boldsymbol{Pr}(A_i, e_i \mid d)$ (referred to as likelihood) is defined as:

$$\boldsymbol{Pr}(A_i, e_i \mid d) = \begin{cases} \boldsymbol{G}(d, A_i) & \text{if } e_i = 0 \\ 1 - \boldsymbol{G}(d, A_i) & \text{otherwise} \end{cases} \tag{1.14}$$

$\boldsymbol{G}(d, A_i)$ (referred to as transaction goodness) is used to account for the fact that components may fail intermittently, estimating the probability of nominal system behavior under an activation pattern A_i and a diagnostic candidate d.

Let g_j (referred to as component goodness) denote the probability that a component c_j performs nominally. Considering that all components must perform nominally to observe a nominal system behavior, $\boldsymbol{G}(d, A_i)$ is defined as:

$$\boldsymbol{G}(d, A_i) = \prod_{j \in (d \cap A_i)} g_j \tag{1.15}$$

In scenarios where the real values for g_j are not available those values can be estimated by maximizing $\boldsymbol{Pr}(A, e \mid d)$ (**Maximum Likelihood Estimation (*MLE*)** for naïve Bayes classifier) under parameters $\{g_j \mid j \in d\}$ [Abreu et al., 2009a]. This approach implies that for a particular candidate d the optimal g_j values may differ from those for another candidate d' for the same components.

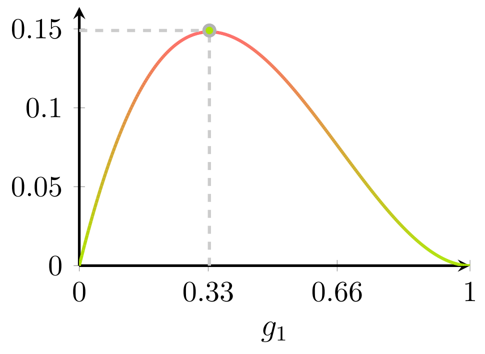
As an example, consider again the hit spectrum in Figure 1.11a. As previously explained, two minimal diagnostic candidates exist: $\{1\}$ and $\{2, 3\}$. In order to rank the candidates we calculate $\boldsymbol{Pr}(d \mid A, e)$ for both candidates. Applying the procedure described above, it follows that:

$$\boldsymbol{Pr}(\{1\} \mid A, e) = \overbrace{\frac{1}{1000} \cdot \frac{999}{1000} \cdot \frac{999}{1000}}^{\boldsymbol{Pr}(d)} \times \overbrace{\underbrace{(1 - g_1)}_{t_1} \cdot \underbrace{(1 - g_1)}_{t_2} \cdot \underbrace{g_1}_{t_3}}^{\boldsymbol{Pr}(A,e|d)} \tag{1.16}$$

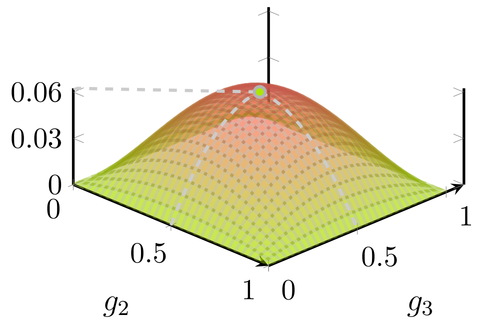
$$\boldsymbol{Pr}(\{2, 3\} \mid A, e) = \overbrace{\frac{999}{1000} \cdot \frac{1}{1000} \cdot \frac{1}{1000}}^{\boldsymbol{Pr}(d)} \times \overbrace{\underbrace{(1 - g_2)}_{t_1} \cdot \underbrace{(1 - g_3)}_{t_2} \cdot \underbrace{(g_2 \cdot g_3)}_{t_3}}^{\boldsymbol{Pr}(A,e|d)} \tag{1.17}$$

By performing a *MLE* for both functions it follows that $\boldsymbol{Pr}(\{1\} \mid A, e)$ is maximized for $g_1 = 0.3_{(3)}$ and $\boldsymbol{Pr}(\{2, 3\} \mid A, e)$ for $g_2 = g_3 = 0.5$ (see Figure 1.15).

Applying the maximizing values to both expressions, it follows that $\boldsymbol{Pr}(\{1\} \mid A, e) = 1.47 \times 10^{-04}$ and $\boldsymbol{Pr}(\{2, 3\} \mid A, e) = 6.25 \times 10^{-08}$ entailing the ranking $\langle \{1\}, \{2, 3\} \rangle$.



(a) $\boldsymbol{Pr}(A, e \mid \{1\})$



(b) $\boldsymbol{Pr}(A, e \mid \{2, 3\})$

Figure 1.15: Likelihood plots

1.3 Research Goals

The goal of our research is to improve *SFL* for run-time environments. Concretely, we improve *SFL* in two different dimensions: accuracy and latency. Accuracy is related to the number of components that are wrongly indicted in a diagnostic report while latency is related to the time needed to calculate a diagnostic report.

Even though these two metrics are also important in development-time scenarios, at run-time (or in a fully automated diagnostic setup) their importance becomes even more increased. On the one hand, a low quality diagnostic report may cause the system to halt due to a large amount of unnecessary maintenance tasks. On the other hand, and due to the fact that the system is on-line, if the errors are not corrected in a timely fashion, they may propagate to other sub-systems or even cause the system to fail.

In view of the aforementioned goals, we draw the following hypothesis:

Spectrum-based Fault Localization algorithms can be improved to better cope with the accuracy and latency requirements of run-time environments.

In the remainder of this section we discuss a set of limitations of the state-of-the-art *SFL* approach (see Section 1.2, page 9). Furthermore, we detail the research goals for this thesis.

1.3.1 Candidate Generation

In this section we discuss the limitations of the current candidate generation approach for *SFL* (see Section 1.2.1, page 12).

1.3.1.1 Algorithmic Efficiency

Staccato, the state-of-the-art algorithm for *SFL* candidate generation, was designed with diagnostic efficiency in mind. As shown in [Abreu and Van Gemund, 2009], the algorithm explores the search space in such a way that guarantees, with high likelihood, that the correct diagnostic candidate is computed. However, as seen in Figure 1.13, the search is often, from a computational point-of-view, inefficient. For the given example, the set $\{2, 3\}$ was unnecessarily evaluated twice.

By improving the algorithm's computational efficiency, it is possible to either compute the same diagnostic candidates in a smaller time frame or, alternatively, explore a larger portion of the search space in the same time frame.

? **Research Question 1**
Is it possible to optimize *Staccato* to minimize redundant/superfluous computations?

1.3.1.2 Horizontal Scalability

An important limitation of existent candidate generation algorithms is their inability to use multiple processing units to compute diagnostic candidates, also known as horizontal scalability.

A consequence of this limitation is that, given the time constraints of run-time environments, one must necessarily trade-off accuracy for performance. By having the ability to use multiple processing units it is possible to take advantage of the ever increasing number of platforms for parallel and distributed computing (*e.g.*, Map Reduce [Dean and Ghemawat, 2004]) to improve the diagnostic accuracy/latency.

? **Research Question 2**
Is it possible to parallelize *Staccato* as way of reducing the diagnostic latency and, if so, by how much?

1.3.2 Candidate Ranking

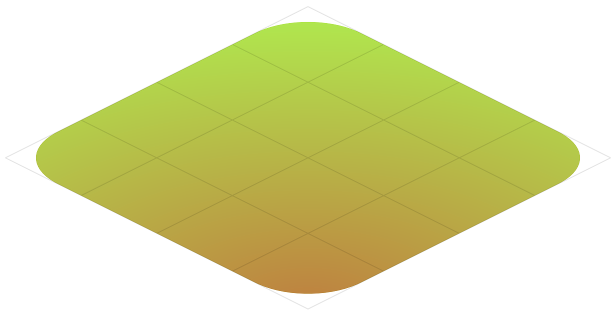
In this section we discuss the limitations of the current candidate ranking approach for *SFL* (see Section 1.2.2, page 17).

1.3.2.1 Fuzzy Errors

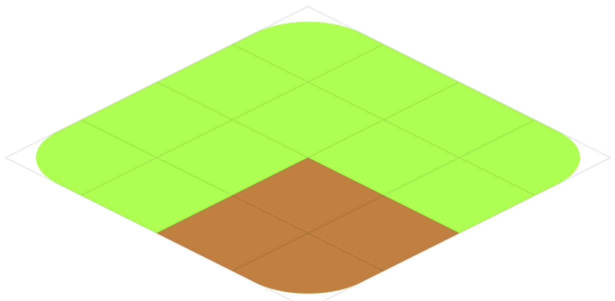
A limitation of the discussed *SFL* candidate ranking approach is related to the assumption that every transaction outcome can be categorized in terms of correct/incorrect [Abreu et al., 2009a, De Kleer, 2009, Casanova et al., 2013, Chen et al., 2013]. While a binary error abstraction works well when diagnosing functional errors (*i.e.*, the output value differs from the expected value), such an abstraction is unable to accurately represent non-functional errors/fuzzy errors (*e.g.*, performance degradation errors)[12].

The presence of non-functional errors in a system implies that the distinction between correct and incorrect states is often *fuzzy*, existing instead a gradual transition between such states. In such scenarios, it is often the case that a system does not break down recognizably but rather deteriorates over time [Ghosh et al., 2007]. Using a binary abstraction to system correctness implies that the perceived deterioration of the system (*i.e.*, the error symptoms) is completely overlooked by the diagnostic algorithm and, as a consequence, diagnostic quality is reduced.

Using our running example to illustrate this limitation, consider that the field is not properly watered. The most likely situation is that, between two arbitrary points, the soil's water level varies gradually (Figure 1.16a). The binary error abstraction limits the perceived error to two possible states (Figure 1.16b). The visual contrast between Figure 1.16a and Figure 1.16b clearly shows that information is lost in the error discretization process, thus impairing the diagnostic accuracy.



(a) Actual error state



(b) Perceived error state

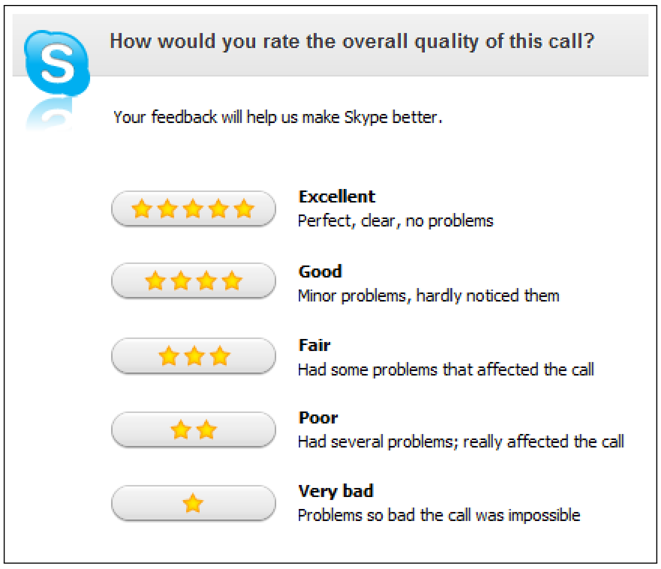
Figure 1.16: Irrigation system example – Fuzzy errors

[12]In this thesis, the terms non-functional errors and fuzzy errors are used interchangeably

As a more computer science related example, consider the following error description for a web service:

- The round-trip time (rtt) for a transaction must be less than 1s.
- Between 0.5s and 1s the performance is sub-optimal.

While a binary error coding can be easily used to represent the error state of a transaction by using the expression $e = rtt < 1$, it fails to encode the sub-optimal performance when $0.5 \leq rtt \leq 1$.



How would you rate the overall quality of this call?

Your feedback will help us make Skype better.

★★★★★	**Excellent** Perfect, clear, no problems
★★★★	**Good** Minor problems, hardly noticed them
★★★	**Fair** Had some problems that affected the call
★★	**Poor** Had several problems; really affected the call
★	**Very bad** Problems so bad the call was impossible

Figure 1.17: Skype call quality rating (screenshot)

As a real-world example of fuzzy errors, consider the case of a voice over IP service such as Skype[13]. After the call is made, the user can be asked to rate the quality of the call in terms of stars: 1 star $\rightarrow$ very bad, 5 stars $\rightarrow$ excellent (Figure 1.17). Intuitively, we can see that a binary error logic abstracts available information from the diagnostic engine as a two-valued logic cannot encode the same information as five-valued logic.

The challenge of solving this limitation is thereby twofold. First, it is necessary to define an appropriate method for both detecting and abstracting non-functional errors and the associated error symptoms. Second, it is necessary to integrate the additional knowledge in the diagnostic process.

? **Research Question 3**

How to encode fuzzy error symptoms?

? **Research Question 4**

How to improve *SFL* to make use of the fuzzy error information?

[13]`https://www.skype.com`

1.3.2.2 System State

A limitation of the *SFL* approach presented in Section 1.2.2 is related to the high level of abstraction enforced by the usage of hit spectra as it does not provide any information about the state of the system during each component's execution. Additionally, it abstracts the number of times each component was used and, consequently, the sequence in which they were used in each transaction. As a consequence, hit spectrum approaches are unable to distinguishing pairs of components for which the activity is equal.

Furthermore, the discussed *SFL* approach estimates g_j as being constant with respect to a set of observations. Consider that the effective (normally unknown) goodness for component (*e.g.*, a hard drive) was directly related to its lifetime and took the shape of a sigmoid, gradually decreasing over time. Due to the fact that, for a hard drive, the slope of the goodness curve is small and the time is monotonic, g_j can, most of the times, be successfully approximated by a constant with small errors. However, if the observations spanned over a long period or the slope of the goodness curve was larger (*e.g.*, for a floppy disk), a constant goodness function would fail to accurately model the actual goodness, entailing large average errors. Given the multiplicative nature of the goodness value usage, even small errors can have a serious impact in the diagnosis report ranking.

Finally, hit spectrum approaches are not able to incorporate existent knowledge about the system in the diagnostic process. As the state of the system is completely abstracted in the hit spectrum, it would be impossible to distinguish, for instance, a new hard drive from an old hard drive. As a consequence, even if the actual goodness curve was available for all components in the system, the algorithm would not be able to use it.

? **Research Question 5**
How to enable *SFL* to adapt to different systems based on previous diagnostic experience?

? **Research Question 6**
How to incorporate information about the system's state in *SFL*?

1.4 Origin of Chapters

Chapters 2 and 3 are based on the work from [Cardoso and Abreu, 2014a], which was published in the proceedings of *the 25th International Workshop on Principles of Diagnosis*[14] (best paper award). An early version of the paper was also published in the proceedings of both the *16th Portuguese Conference on Artificial Intelligence (EPIA)*[15] [Cardoso and Abreu, 2013b] and the *International Conference on*

[14]http://dx-2014.ist.tugraz.at/
[15]http://www.epia2013.uac.pt/

Multi-core Software Engineering, Performance, and Tools (MUSEPAT)[16] [Cardoso and Abreu, 2013a].

Chapter 4 is based on the work from [Cardoso and Abreu, 2014b], which was published in the proceedings of *the 25th International Workshop on Principles of Diagnosis* (best paper award nominee).

Chapter 5 is based on the work from [Cardoso and Abreu, 2013c], which was published in the proceedings of the *27th AAAI Conference on Artificial Intelligence (AAAI)*[17].

1.5 Thesis Outline

In Figure 1.18 we present the thesis outline. We suggest a pre-order traversal of the tree, from top to bottom.

[16] `http://eventos.fct.unl.pt/musepat2013/`
[17] `http://www.aaai.org/Conferences/AAAI/aaai13.php`

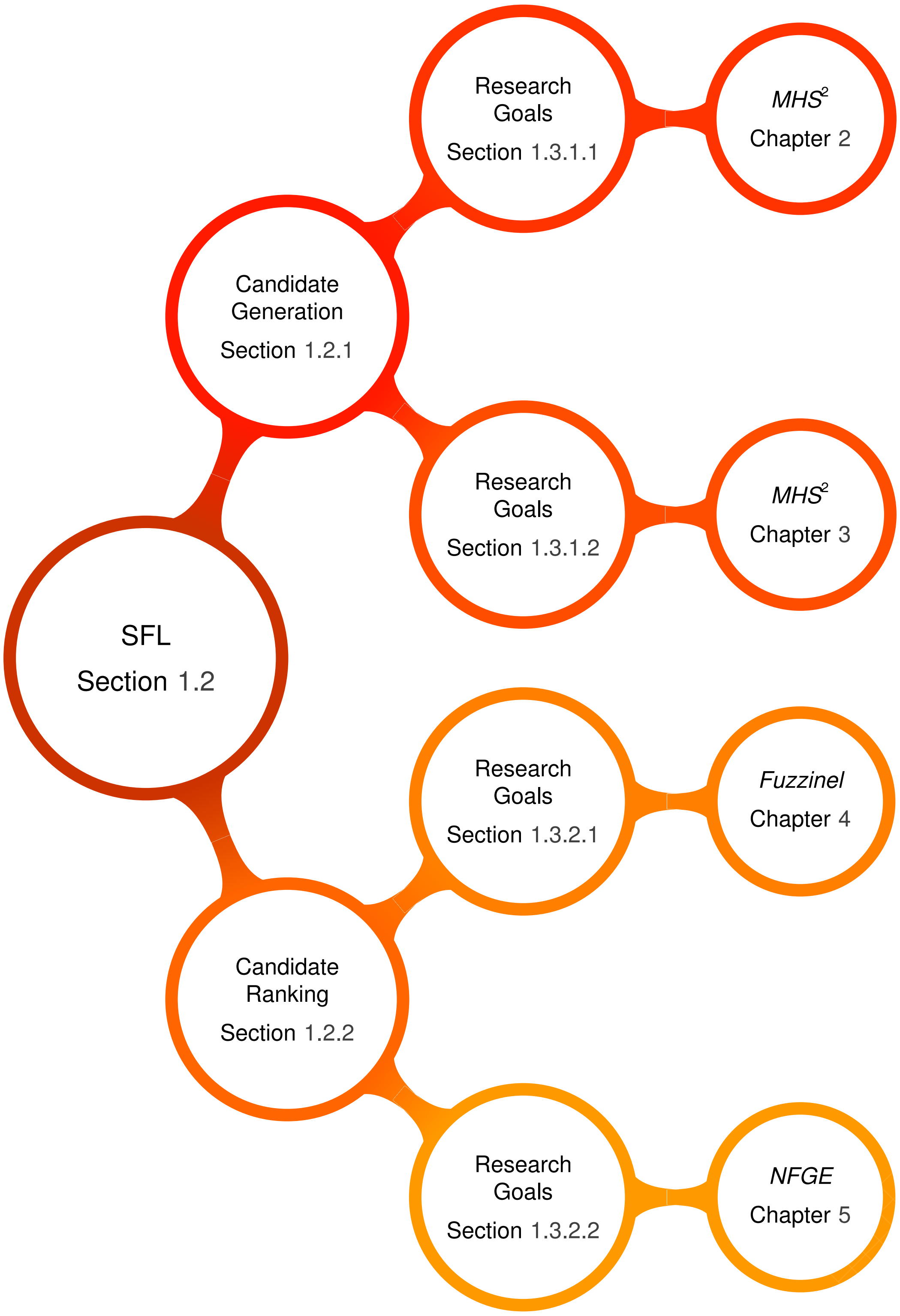


Figure 1.18: Thesis outline

```
++[---------->+<]>.-----.+++++++++++.>-[----->+<]>-.--[--->++<]>.--[--
>+++<]>.[--->++<]>++.[-->+++++<]>-.+[--->+<]>.++++.-----------.++++.--
--.-[--->+<]>++.[->+++++<]>-.--[--->+<]>-.-----------.++++++.-.+++++.-
---<->-<><+<+<+->-+<<++<-<>---<>+>><<+<>-<<+<><<+-+-<+<+><+<><>+-+-+>-
```

2 *MHS*^{2} – Optimizations

In this chapter we present the sequential version our novel *MHS* generation algorithm, dubbed **Map-Reduce Heuristic-driven Search for Minimal Hitting Sets (*MHS*^{2}).** As explained in Section 1.3.1.1 (page 20), our goal in this chapter is to improve the computational efficiency of the *Staccato* algorithm.

This chapter is divided as follows. First, we introduce our *MHS* generation algorithm. Second, we provide a formal analysis of the proposed algorithm. Third, we discuss the data-structures used in the algorithm's implementation. Fourth, we discuss some practical aspects of the algorithm's utilization. Fifth and lastly, we evaluate the performance of our algorithm.

2.1 Approach

The proposed algorithm consists of an improved version of the *Staccato* algorithm [Abreu and Van Gemund, 2009] (see Algorithm 1.1, page 14). In contrast to the original algorithm, we propose 3 optimizations that prevent redundant calculations (Figure 2.1).

The first optimization (line 7 and Figure 2.1a) prevents multiple evaluations of the same set, as it was the case of $\{2, 3\}$ in the example presented in Figure 1.13 (page 16). Removing element j from U (and, consequently, from S) after its evaluation has the practical effect of reducing the search space to a spanning tree of the original search tree.

To see the effect of this optimization, consider the example in Figure 2.1a. The example depicts the largest search trees for $U = \{1, 2, 3\}$ with and without optimization 1. Each box represents a call to algorithm and the value contained inside each box represents the value of d for that particular call.

Without the proposed optimization, a large amount of redundant work would be performed (*e.g.*, 8 evaluations of the set $\{1, 2, 3\}$). With the proposed optimization, it is guaranteed

Algorithm 2.1 *MHS²* – Optimizations

```
Inputs: $(U, S, d = \emptyset, D = \emptyset)$

Output: Minimal hitting set collection $D$

1  if $\exists_{s \in S} : s \cap U = \emptyset$ then                          # Opt 3
2      return $D$
3  if $S \neq \emptyset$ then
4      $U \leftarrow U \setminus \{j \mid j \in U \wedge (\nexists_{s \in S} : j \in s)\}$     # Opt 2
5      for $j \in \boldsymbol{Rank}(U, S)$ do
6          $S' \leftarrow \{s \mid s \in S \wedge j \in s\}$
7          $U \leftarrow U \setminus \{j\}$                                    # Opt 1
8          $D \leftarrow \boldsymbol{MHS}^2(U, S \setminus S', d \cup \{j\}, D)$
9  else
10     if $\nexists_{d' \in D} : d' \subseteq d$ then
11         $D \leftarrow D \setminus \{d' \mid d' \in D \wedge d \subseteq d'\}$
12         $D \leftarrow D \cup \{d\}$
13 return $D$
```

that every set is evaluated at most once. Concretely, the upper-bound size of the search space without optimizations can be described with the following expression:

$$\boldsymbol{a}(U) = 1 + \boldsymbol{b}(|U|) \tag{2.1}$$

$$\boldsymbol{b}(x) = \begin{cases} x + x * \boldsymbol{b}(x-1) & \text{if } x > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

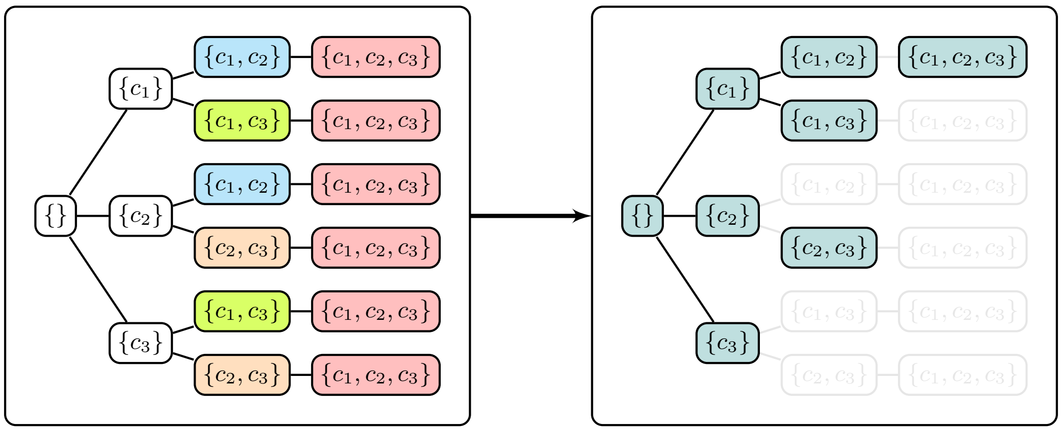
$\boldsymbol{b}(x)$ is described on the On-Line Encyclopedia of Integer Sequences as the number of permutations of nonempty subsets of a set with size x. Since the empty set may also be a solution for a given problem, it is necessary to add 1 to the result of $\boldsymbol{b}(x)$ (see `https://oeis.org/A007526` for further information).

On the other hand, the upper-bound size of the search space for the optimized algorithm is given by the following expression:

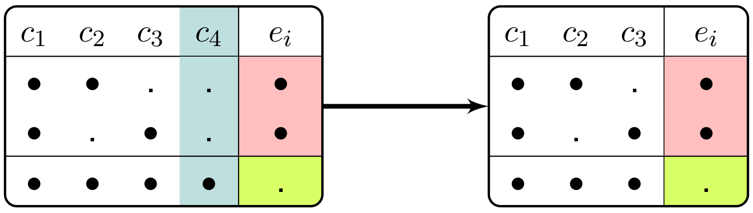
$$\boldsymbol{c}(U) = 2^{|U|} \tag{2.3}$$

To illustrate both upper-bound search space sizes, both functions are plotted in Figure 2.2. We can clearly see that $\boldsymbol{a}(U)$ grows much faster than $\boldsymbol{c}(U)$ (note the log scale).

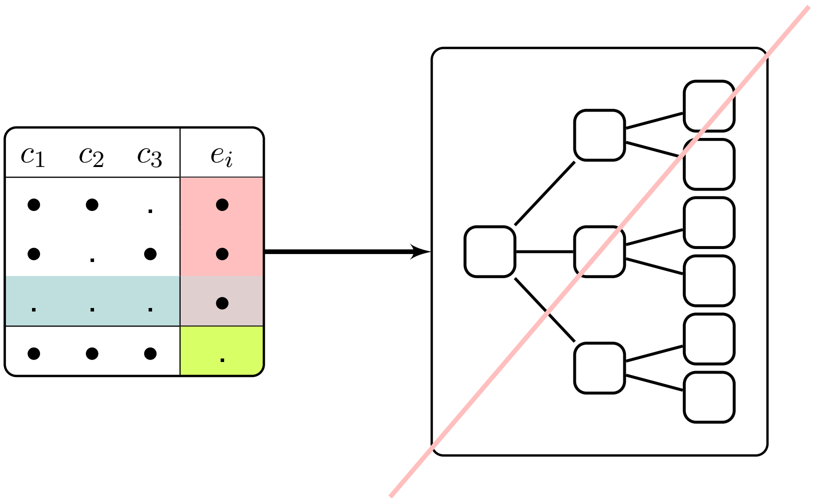
The second optimization (line 4 and Figure 2.1b) preemptively filters elements of U not contained in any set in S. As this filtering process reduces the size of U and, consequently, the size of the data structures holding (U, S), the operations performed on (U, S), such as the ranking calculation, become faster.



(a) Optimization 1



(b) Optimization 2



(c) Optimization 3

Figure 2.1: Optimizations' visual intuition

The third optimization (lines 1, 2, and Figure 2.1c) prevents the examination of branches such that $\exists_{s \in S} : s \cap U = \emptyset$, *i.e.*, there is at least one set that cannot be hit by any element in U. The existence of such a set in S guarantees, by definition, that no *HS* will be found (see Corollary 1.2.2, page 12).

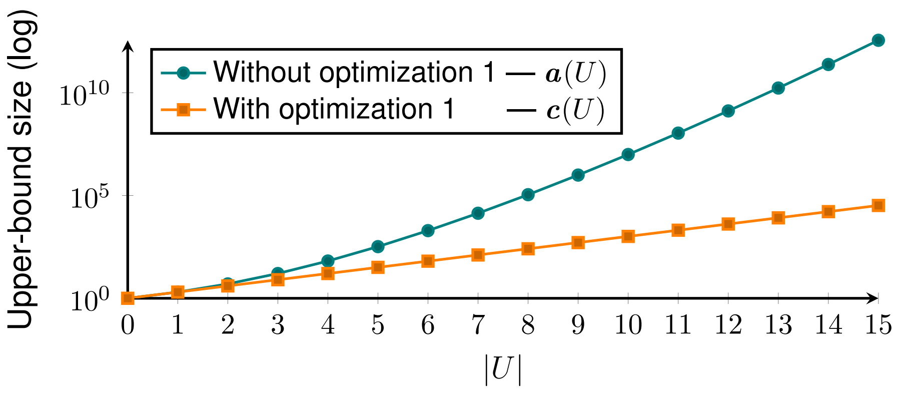


Figure 2.2: Search space size comparison

2.2 Algorithm Analysis

In this section we discuss the algorithm from a more formal point-of-view. First, we prove the algorithm's soundness and completeness.[1] Second, we provide some intuition on the complexity of the different operations performed by the *MHS²* algorithm.

2.2.1 Completeness/Soundness analysis

Lemma 2.2.1. *All sets computed by MHS² are HSs.*

Proof. Suppose *MHS²* is set to compute a single *HS*. At each step, an element $j \in U$ is selected and *MHS²* is recursively called for $S \setminus S'$ with set $d \cup \{j\}$. This procedure is repeated until $S = \emptyset$, at which point $\emptyset$ is a *HS* for S (see Corollary 1.2.3, page 12).

Let S'_n and j_n be the S' set and j element at the n^{th} recursive level. Since $\forall_{s \in S'_n} : j_n \in s$ (*i.e.*, $\{j_n\}$ is a hitting set for S'_n), it follows that:

$$\bigwedge_{1}^{n} \boldsymbol{HS}(U_n, S'_n, d_n) \tag{2.4}$$

From Corollary 1.2.4 (page 12), as $S = \bigcup_1^n S'_n$ and $d = \bigcup_1^n d_n$, it follows that:

$$\bigwedge_{1}^{n} \boldsymbol{HS}(U_n, S'_n, d_n) \implies \boldsymbol{HS}(U, S, d) \tag{2.5}$$

■

Lemma 2.2.2 (Completeness)**.** *Staccato computes all MHSs for a problem* (U, S)*.*

[1] In this context, and assuming no cutoffs are enforced, the algorithm is sound if all computed sets are *MHSs*. A complete algorithm should compute all *MHSs*. It is worth noting that a sound algorithm may miss some *MHSs* and a complete algorithm may compute some sets that are not *MHSs* (*i.e.*, soundness $\not\Leftrightarrow$ completeness).

Proof. Suppose *Staccato* does not stop the branch exploration after a *HS* is found. Under this constraint, it follows that the algorithm comprehensively explores every element in $\mathcal{P}(U)$, which contains all *HSs* for (U, S) and, consequently, all *MHSs* for (U, S).

By stopping the branch exploration after an *HS* d is found, the *HSs* contained in $\{d \cup d' \mid d' \subseteq (U \setminus d) \wedge d' \neq \emptyset\}$ are ignored. Since, by definition $\boldsymbol{HS}(U, S, d) \wedge (d' \neq \emptyset) \implies \neg\boldsymbol{MHS}(U, S, d \cup d')$, we can conclude that even though some *HSs* are ignored, all *MHSs* are nonetheless computed. ■

Theorem 2.2.1. *Optimization* 1 *preserves completeness.*

Proof. Let $\Omega = \mathcal{P}(U \setminus (d \cup \{j\}))$. After every recursive call to *Staccato*, it is guaranteed that all *MHSs* contained in $K = \{d \cup \{j\} \cup e \mid e \in \Omega\}$ are evaluated (Lemma 2.2.2).

By removing j from U for subsequent calls of *Staccato* under the same search tree branch, the algorithm prevents subsequent evaluations of the sets contained in K, only calculating the *MHSs* contained in $L = \{d \cup e \mid e \in \Omega\}$. Since $K \cup L = \{d \cup e \mid e \in \mathcal{P}(U \setminus d)\}$, the optimization preserves completeness. ■

Theorem 2.2.2. *Optimization* 2 *preserves completeness.*

Proof. Let $K = \{j \mid j \in U \wedge (\nexists_{s \in S} : j \in s)\}$. By definition, it follows that:

$$\boldsymbol{HS}(U, S, d) \wedge d \cap K \neq \emptyset \implies \neg\boldsymbol{MHS}(U, S, d) \tag{2.6}$$

Since removing elements in K from U only prevents the calculation of sets that are guaranteed not to be *MHSs*, we can conclude that the optimization preserves completeness. ■

Theorem 2.2.3. *Optimization* 3 *preserves completeness.*

Proof. From Corollary 1.2.2 (page 12), it follows that:

$$\exists_{s \in S} : s \cap U = \emptyset \implies \nexists_d : \boldsymbol{HS}(U, S, d) \tag{2.7}$$

Since halting the exploration of a branch whenever the aforementioned condition holds only prevents the examination of sets that are guaranteed not to be *HSs*, we can conclude that the optimization preserves completeness. ■

Corollary 2.2.1. *MHS^2 is complete.*

Theorem 2.2.4 (Soundness)**.** *If all MHSs are evaluated, all sets computed by MHS^2 are MHSs.*

Proof. Given the fact that a set d is only added to D if and only if $\nexists_{d' \in D} : d' \subseteq d$ and, prior to its addition, all sets in $\{d' \mid d' \in D \wedge d \subseteq d'\}$ are removed from D, it follows that, at any given point, the sets in D are relatively minimal (*i.e.*, $\nexists_{d, d' \in D} : d' \subset d$). Since, by hypothesis, all existent *MHSs* for the problem are evaluated (and therefore added to D) and all sets contained in D are always relatively minimal, no absolutely non-minimal *HS* is contained in D. ■

2.2.2 Complexity Analysis

As shown in [Garey and Johnson, 1990], the *MHS* problem is NP-Hard. In this section we provide upper-bound values for the complexity of the operations performed in each call of *MHS*².

Each time *MHS*² is called it:

1. Checks whether the current sub-problem has any solutions: $\mathcal{O}\big(M \times N\big)$, where $M = |U|$ and $N = |S|$ (line 1, optimization 3).
2. Checks whether the current sub-problem is trivially solvable: $\mathcal{O}\big(1\big)$ (line 3).

If the problem is not trivially solvable, the algorithm:

1. Removes elements from U that are guaranteed not to form *MHSs*: $\mathcal{O}\big(M \times N\big)$ (line 4, optimization 2).
2. Generates heuristic values for each element in U: at least $\mathcal{O}\big(M\big)$, usually $\mathcal{O}\big(M \times N\big)$ (line 5).
3. Ranks elements of U according to their heuristic values: $\mathcal{O}\big(M \times \log(M)\big)$ (line 5).
4. Prepares, for each element of the ranking, a sub-problem to be solved: $\mathcal{O}\big(M^2 \times N\big)$ (lines 6 – 7).
    a) Simplifies the problem: $\mathcal{O}\big(M \times N\big)$ (line 6).
    b) Removes the current element from U: $\mathcal{O}\big(1\big)$ (line 7, optimization 1).

If the problem is trivially solvable, the algorithm:

1. Checks if the current set d is minimal with respect to D: $\mathcal{O}\big(I \times K \times L\big)$, where $I = |d|, K = |D|$, and $L = \sum_{d' \in D} \frac{|d'|}{K}$ (line 10).
2. Removes non-minimal *HSs* from D: $\mathcal{O}\big(I \times K \times L\big)$ (line 11).
3. Adds d to D: $\mathcal{O}\big(I\big)$ (line 12).

2.3 Data structures

To have an efficient implementation of *MHS*² the following operations must be performed efficiently:

1. Remove elements from S (line 6);

2. Check set for relative minimality (line 10);

3. Purge proper super sets of a set from collection D (line 11);

In the remainder of this section, we analyze the data structures used in our reference implementation. The implementation is available at `https://github.com/npcardoso/MHS2`.

2.3.1 Encoding (U, S)

To efficiently encode and manipulate (U, S), we make use of a $N \times M$ binary matrix (referred to as A) where each of the matrix's rows encodes the membership of each element $j \in U$ in a particular set S_i (*i.e.*, $A_{ij} = [j \in S_i]$, where $[.]$ denotes Iverson's operator: $[True] = 1, [False] = 0$).

To avoid making multiple copies of the matrix when a modification to (U, S) is required, we make use of a data structure we refer to as a *filter*. In practice, a filter is a vector storing which columns/rows should be ignored at each stage of the algorithm's execution. Consequently, the resultant data structure is a 3-tuple (A, fc, fr) representing the original (U, S) matrix, and the filtered columns/rows respectively.

To illustrate how this works in practice, consider the example in Figure 2.3 showing the matrix representation of (U, S). Similarly to Figure 1.13 (page 16), each node in the search tree represents a call to the function (all the parameters as well as the return value are encoded as a table). Colored matrix cells represent the filtered elements of the original matrix. Each circle marks the element that triggered the filtering of corresponding row.

A naïve implementation of such filters could be done with Boolean vectors where each cell encodes the state of each column/row. Even though this approach is more efficient than copying the matrix multiple times, it is possible to improve this structure to be more efficient when iterating over a filtered matrix.

To that end, we use an integer vector where each cell contains the index of the next unfiltered element. Such filter vectors have $n + 1$ elements, where n is the maximum number of elements to be filtered. The filter vector indexes are zero-based so it is possible to filter the first row/column of the matrix, which are one-based. The initial state for n element filter is formally defined as $f_i = (i + 1) \mod (n + 1)$.

To iterate using one of such filters, a counter starting at 0 must be kept and, after each iteration, it is updated as $i = f_i$. The loop ends when the counter returns its value to 0.

To filter an element i, one must iterate back from $j = i - 1$ until $f_j \neq i$, setting each f_j to be equal to f_i. In contrast to the naïve approach, when all elements are filtered, the loop is never executed, improving the efficiency.

Figure 2.4 depicts the states of a filter for a maximum of four elements after filtering elements 3, 1, 2, and, finally, 4.

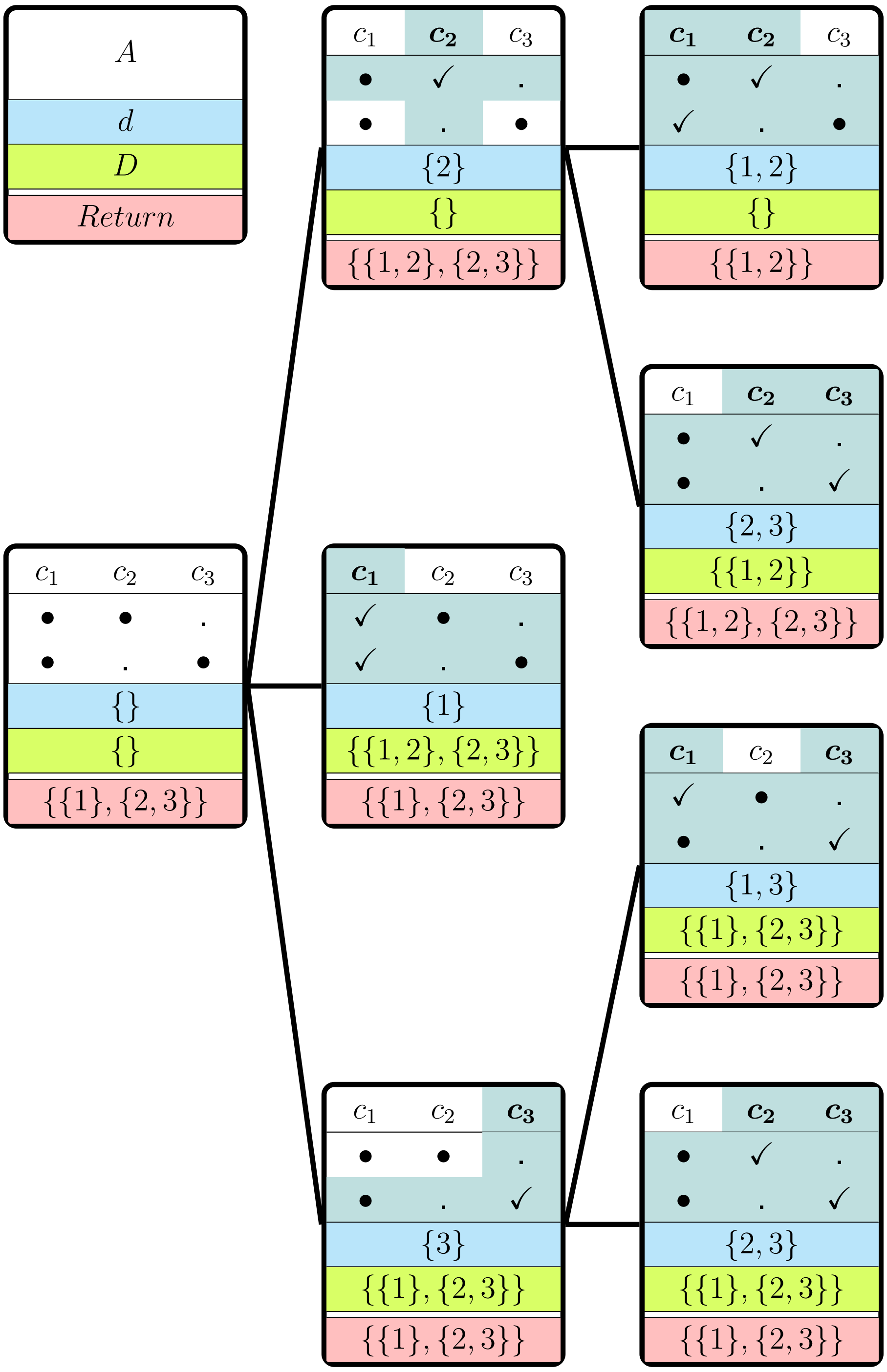


Figure 2.3: Example search tree showing binary matrix for (U, S) (no optimizations)

2.3.2 Encoding D

To efficiently store *HSs*, check their relative minimality and purge non-*MHSs*, we use the trie data structure. A trie is a tree-like data structure where the set of node values in the path from the root to a marked node corresponds to an element (in this case a *HS*) stored in the trie.

To operate a trie, we assume the existence of the following basic operations:

i	0	1	2	3	4
Initial state	1	2	3	4	0
After filter $i = 2$	1	3	$\textcircled{3}$	4	0
After filter $i = 4$	1	3	3	0	$\textcircled{0}$
After filter $i = 1$	3	$\textcircled{3}$	3	0	0
After filter $i = 3$	0	0	0	$\textcircled{0}$	0

Figure 2.4: Filter with 4 elements

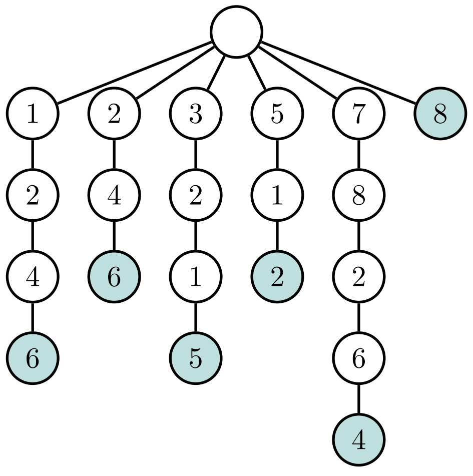
- $\boldsymbol{isMarked}(n)$: Checks whether node n is marked. If node n is an invalid node (denoted as $\times$) the function returns $False$.

- $\boldsymbol{getChild}(n, e)$: Returns the child node of n with value e or $\times$ if no child of node n has value e.

- $\boldsymbol{getChildren}(n)$: Returns the node values of the children of node n.

- $\boldsymbol{setChild}(n, e, n')$: Assigns node n' as the child of n with value e. If $n' = \times$, the function removes the child node with value e instead.

An important prerequisite to efficiently implement the required (complex) trie operations is that the elements in all *HSs* are ordered (in the following we assume ascending order). On the one hand, the ordering assumption guarantees that any *HS* has a unique representation (*e.g.*, {1,2,3} *vs.* {2,3,1} or {3,2,1}). On the other hand, it guarantees that the trie itself has an ordered structure (*i.e.*, $\nexists_{n,n',e,e'} : n' = \boldsymbol{getChild}(n, e) \wedge e' \in \boldsymbol{getChildren}(n') \wedge e' \leq e$), enabling a reduction in the amount of nodes to be processed when checking for minimality and purging super sets.


(a) Using unordered sets

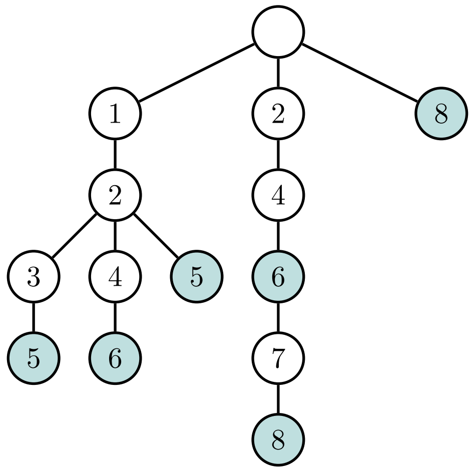

(b) Using ordered sets

Figure 2.5: A trie encoding 6 *HSs*

As an example consider two possible tries storing *HSs* $\{1, 2, 3, 5\}$, $\{1, 2, 4, 6\}$, $\{1, 2, 5\}$, $\{2, 4, 6\}$, $\{2, 4, 6, 7, 8\}$, and $\{8\}$, which are presented in Figure 2.5.

Algorithm 2.2 $\mathit{isMinimal}$

Inputs: (n, d)

Output: Boolean

```
if $\boldsymbol{isMarked}(n)$ then
    return $False$
else if $n \neq \times$ then
    while $d \neq \emptyset$ do
        $e \leftarrow \boldsymbol{min}(d)$
        $d \leftarrow d \setminus \{e\}$
        $n' \leftarrow \boldsymbol{getChild}(n, e)$
        if $\neg\boldsymbol{isMinimal}(n', d)$ then
            return $False$

return $True$
```

The algorithm for checking the minimality of an *HS* in view of the *HSs* encoded in the trie (line 10 in Algorithm 2.1) is presented in Algorithm 2.2. The algorithm attempts to find a path from the root node to a marked node composed exclusively of elements from set d. If no such path is found, set d has no subsets in the trie and, consequently, is minimal. In practice, the algorithm works by iteratively selecting an element e from d and recursively exploring the child node with value e using the set $d \setminus \{e\}$. Since the trie has an ordered structure, the elements of d are orderly selected and removed from d, avoiding the need to explore different permutations of the same sets.

The algorithm for purging all supersets of a particular *HS* from the trie (line 11 in Algorithm 2.1) is presented in Algorithm 2.3. The algorithm attempts to find all the paths from the root node to some node in the trie containing all elements from set d and removes them. In practice, the algorithm explores the tree recursively and, whenever the value (e) of the node being inspected is contained in d, e is removed from d for subsequent recursive calls in that particular branch. Whenever d becomes empty (signaling that all elements in the original set d are contained in the path to node n), that particular subtree is removed from the trie.

While removing the subtree when d is empty effectively removes all supersets from the trie, it may also leave extraneous nodes in the trie (*i.e.*, nodes that are not marked and do not have marked nodes in their sub-trees). To remove such extraneous nodes, after exploring the sub-trees of a particular node, the algorithm checks if it is marked and, if not, if it has no children. If the node is not marked and has no children, it is also removed from the trie.

Algorithm 2.3 $\mathit{purgeSuperSets}$

```
Inputs: $(n, d)$
Output: Node
if $d = \emptyset$ then
    return $\times$
else
    $e \leftarrow \boldsymbol{min}(d)$
    for $e' : e' \in \boldsymbol{getChildren}(n) \wedge e' \leq e$ do
        $d' \leftarrow d$
        if $e' = e$ then
            $d' \leftarrow d' \setminus \{e\}$
        $n' \leftarrow \boldsymbol{purgeSuperSets}(\boldsymbol{getChild}(n, e'), d')$
        $n \leftarrow \boldsymbol{setChild}(n, e', n')$
if $\neg \boldsymbol{marked}(n) \wedge \boldsymbol{getChildren}(n) = \emptyset$ then
    return $\times$
return $n$
```

2.4 Practical Considerations

In this section we discuss domain-specific aspects of the proposed algorithm. First, we discuss the selection of the heuristic as a way of improving the performance of *MHS²* in a particular domain. Second, we present two pre-processing operations aimed at decreasing the problem's complexity while guaranteeing that the solution remains unaltered.

2.4.1 Heuristics

To fine tune the algorithm for a domain specific goal, the algorithm uses the concept of heuristic to drive the search by determining the shape and order of exploration of the search tree. An example of heuristic would be to rank the elements of U according to the number of sets they hit. Concretely, such a heuristic would be defined as follows:

$$\boldsymbol{\mathcal{H}}(j) = \left|\{s \mid s \in S \wedge j \in s\}\right| \tag{2.8}$$

Intuitively, this heuristic attempts to greedily minimize the size of the computed *HSs*.

In the context of *SFL*, as faults may not be consistently triggered, failing transactions convey information on the possible causes of a system malfunction whereas successful transactions help the diagnosis system to exonerate components that would otherwise be inaccurately judged as faulty. As the goal of diagnosis is to find the best explanation for an error, using a heuristic that minimizes the size of the computed *HSs* may not be ideal.

To convey this intuition to *Staccato*, the *Ochiai* similarity coefficient [e Silva MeyerDada al., 2004] was used as a heuristic [Abreu and Van Gemund, 2009]. The selection of this particular similarity coefficient was based on the results of previous work on similarity-based *SFL* showing evidence that, for real-world diagnostic scenarios, *Ochiai* provides good heuristic results [Abreu et al., 2007]. Concretely, the heuristic is defined as follows:

$$\boldsymbol{\mathcal{H}}(j) = \frac{n_{11}(j)}{\sqrt{\big(n_{11}(j) + n_{01}(j)\big) * \big(n_{11}(j) + n_{10}(j)\big)}} \tag{2.9}$$

where

- n_{11}: Number of failing transactions where j was active.
- n_{10}: Number of failing transactions where j was not active.
- n_{01}: Number of passing transactions where j was active.

To illustrate how the heuristic works in practice, consider the search tree in Figure 2.6 in which the heuristic values are presented. Similarly to Figure 2.3 (page 34), each node in the search tree represents a call to the function. Unlike Figure 2.3, instead of only representing the conflicts, the full hit spectrum is presented[2]. Also, Figure 2.6 depicts a search tree for the fully optimized *MHS* algorithm.

We can see that the ranking for the outermost call the component evaluation order is $\langle c_2, c_1, c_4, c_3 \rangle$. Intuitively, this is due to the fact that c_1 and c_2 share the same activation pattern in failing transactions but c_1 was also activated in a passing transaction whereas c_2 was not. The same logic can be applied between $c3$ and c_4. Since c_1 and c_2 were activated in more failing transactions than c_3 and c_4, they are evaluated first.

It is important to note that, due to the fact that the failing transactions get filtered during the algorithm's execution, the value of the heuristic for a particular element may vary for different nodes in the search tree.

2.4.2 Ambiguity Group Removal

Ambiguity groups [González-Sanchez et al., 2011a] occur when two or more elements of U are members of the same sets in S (*i.e.*, for two elements j_1, j_2, $\forall_{s \in S} : j_1 \in s \Leftrightarrow j_2 \in s$). If such groups exist, if follows that:

$$\boldsymbol{MHS}(U, S, d \cup \{j_1\}) \Leftrightarrow \boldsymbol{MHS}(U, S, d \cup \{j_2\}) \tag{2.10}$$

Intuitively, the above formula states that the elements from an ambiguity group are interchangeable since they hit the same sets. It is possible to take advantage of this fact by collapsing the different ambiguity groups into single elements (and making sure such information is available), thus reducing the search space.

[2]Recall that every node having all rows with $e_i = 1$ (*i.e.*, the conflicts) filtered represents a (potentially non-minimal) diagnostic candidate.

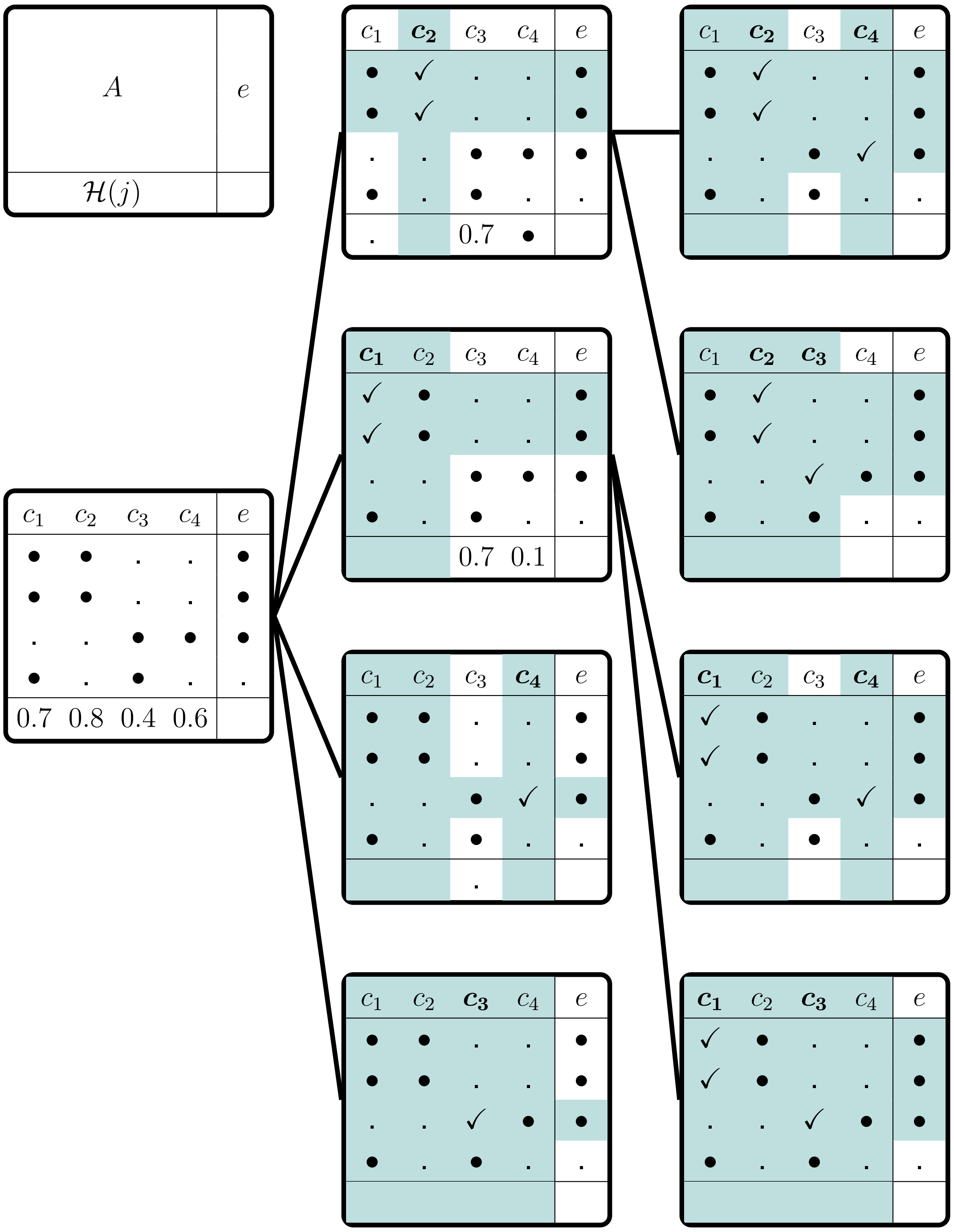


Figure 2.6: Example search tree with heuristic values (with optimizations)

As an example consider the following *MHS* problem:

$$\begin{aligned} U &= \{1,2,3,4\} \\ S &= \{\{1,2\},\{3,4\}\} \\ D &= \{\{1,3\},\{1,4\},\{2,3\},\{2,4\}\} \end{aligned} \tag{2.11}$$

For this example elements 1 and 2 form an ambiguity group. Also, 3 and 4 form another ambiguity group. Running the same algorithm with ambiguity group removal, only 1 *HS* is generated ($\{1,3\}$). However, since we know that 1 and 2 as well as 3 and 4

are interchangeable, the remaining 3 *MHSs* can be generated by applying all possible substitutions.

2.4.3 Problem Minimization

A non-minimal (U, S) problem occurs when:

$$\exists_{s,s' \in S} : s \subset s' \tag{2.12}$$

Whenever such condition holds, solving (U, S) is equivalent to solving (U, S'), where $S' = S \setminus \{s'\}$. This is due to the fact that, on the one hand, by hitting s it is guaranteed by definition that all sets s' are also necessarily hit. On the other hand, by hitting a set s' with an element x (*i.e.*, $x \in s'$), it does not guarantee that s is also hit. In the case of x not hitting s, 2 possible outcomes can take place:

1. There is a set $\epsilon \in S'$ such that $x \in (s' \cap \epsilon)$. In this scenario, if $d \cup \{x\}$ is a *MHS* for (U, S), it also is a *MHS* for (U, S').
2. There is not a set $\epsilon \in S'$ such that $x \in (s' \cap \epsilon)$. In this case, if $d \cup \{x\}$ is a *HS* for (U, S), d also is a *HS* for (U, S) and, therefore, $d \cup \{x\}$ is not a *MHS* for (U, S).

Given this, the removal of all sets s' from S does not alter its result. Consequently, solving the problem (U, S) or its minimal version is equivalent. However, since the size of the data-structures is reduced, the algorithm performs faster. To implement this functionality, one may use the trie data structure described in Section 2.3.2 to filter the non-minimal sets.

As an example consider the following problem:

$$\begin{aligned} U &= \{1, 2, 3, 4\} \\ S &= \{\{1, 2\}, \{1, 3\}, \{1, 2, 4\}, \{1, 2, 3, 4\}\} \\ D &= \{\{1\}, \{2, 3\}\} \end{aligned} \tag{2.13}$$

For this problem we can see that sets $\{1, 2, 4\}$ and $\{1, 2, 3, 4\}$ are not minimal as both are supersets of $\{1, 2\}$. In fact, only sets $\{1, 2\}$ and $\{1, 3\}$ are minimal, which produce the following minimal problem:

$$\begin{aligned} U' &= \{1, 2, 3\} \\ S' &= \{\{1, 2\}, \{1, 3\}\} \\ D' &= \{\{1\}, \{2, 3\}\} \end{aligned} \tag{2.14}$$

As expected, the solution to both problems is equal.

2.5 Benchmark

To assess the performance of our algorithm we conducted a set of benchmarks aimed at evaluating the impact of each of the proposed optimizations(all the benchmarks were conducted in a single computer with $2\times$ *Intel Xeon* **Central Processing Unit (*CPU*)** *X5570* @ 2.93GHz, 4 cores each).

For our benchmarks, we generated several *MHS* problems by means of a Bernoulli process, with parameters $M = |U|$, $N = |S|$, and $R = \boldsymbol{Pr}(j \in s)$. The presented results represent the average over 100 problems for each combination (M, N, R). Furthermore, when applicable, the *Ochiai* heuristic was used, and no pre-processing was performed on (U, S).

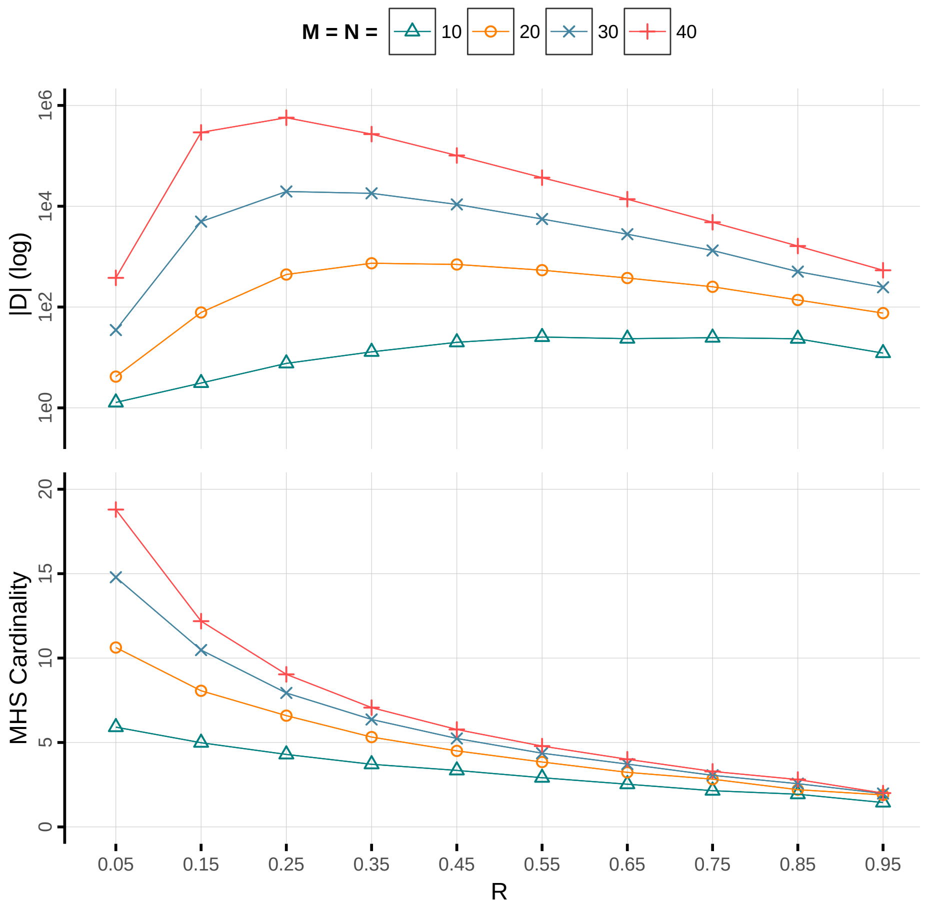


Figure 2.7: (M, N, R) parameters' impact

To better understand how the parameters affect the problem's solution, consider the following observations (Figure 2.7):

1. For the same (M, N) parameter values:

   a) The average *MHS* cardinality for problems generated with smaller R values is larger than the average *MHS* cardinality for problems generated with larger R

values (*i.e.*, the *MHS* cardinality is negatively correlated with the R value).

   b) The average solution size (*i.e.*, $|D|$) was minimal for problems generated with $R = 0.05$.

   c) $|D|$ was **not** maximal for problems generated with $R = 0.95$.

2. Problems generated with larger (M, N) values have the maximal $|D|$ value for smaller R values than problems generated with smaller (M, N) values (*i.e.*, the value of R for maximal $|D|$ is negatively correlated with (M, N)).

2.5.1 Small problems

The first benchmark in this section is aimed at evaluating the impact of each optimization for small problems, for which all *MHSs* can be calculated ($M = N \in \{10, 20, 30, 40\}$, and $R \in \{0.05, 0.15, ..., 0.95\}$).

In Figure 2.8, we observe the throughputs[3] for 5 different implementations of the algorithm with an increasing number of features. At one end of the scale, *Baseline* represents an implementation with no optimizations as well as no heuristic (*i.e.*, random ranking). At the other end, *Opts $1 - 3$* represents an implementation with both the *Ochiai* heuristic and all the proposed optimizations (the ids are presented in Algorithm 2.1). *Heuristic* represents the performance of *Staccato*. For readability, data points for which the throughput was less than 0.1 *MHS*/sec were omitted from the plots.

The analysis of the results shows that the heuristic by itself introduces a significant performance improvement over *Baseline* for $M = N = 10$ ($5\times$ faster on average, note the log scale). Such result presents a strong evidence that using a heuristic to drive the search can not only improve the quality of the computed *MHSs* (as shown in [Abreu and Van Gemund, 2009]) but also improve the computational efficiency of the algorithm. Additionally, and in comparison to the *Heuristic* (*i.e.*, *Staccato*) performance, all the optimizations showed an improved performance of at least $1.5\times$ (*Opt 1* for $R = 0.95$), at most $170000\times$ (*Opts $1 - 3$* for $R = 0.05$), and on average $34000\times$, $8000\times$, and $1700\times$ for *Opts $1 - 3$*, *Opts $1 - 2$*, and *Opt 1* respectively.

Analyzing the remaining tests cases ($M = N \in \{20, 30, 40\}$), we can see that, with the increase of the problem size and for small R values, the relative contribution of each optimization becomes more significant (for $M = N \in \{10, 20, 30\}, R = 0.05$, *Opts $1 - 3$* performs $30\times$, $7000\times$, and $1000000\times$ faster than *Opt 1*, respectively). We also observe that, on average and for all combinations of (M, N, R), *Opts $1 - 3$* was the algorithm with the best performance, implying that the computational savings introduced by optimization 3 should, on average, outweigh its overhead.

It also worth understanding how each optimized implementation improves over *Heuristic*. A closer inspection of the results reveals the following patterns:

[3]The throughput metric is calculated as the number of generated *MHSs* divided by the total run-time and is measured in *MHSs*/sec. When calculating this metric, we took care to discard all non-minimal *HSs*.

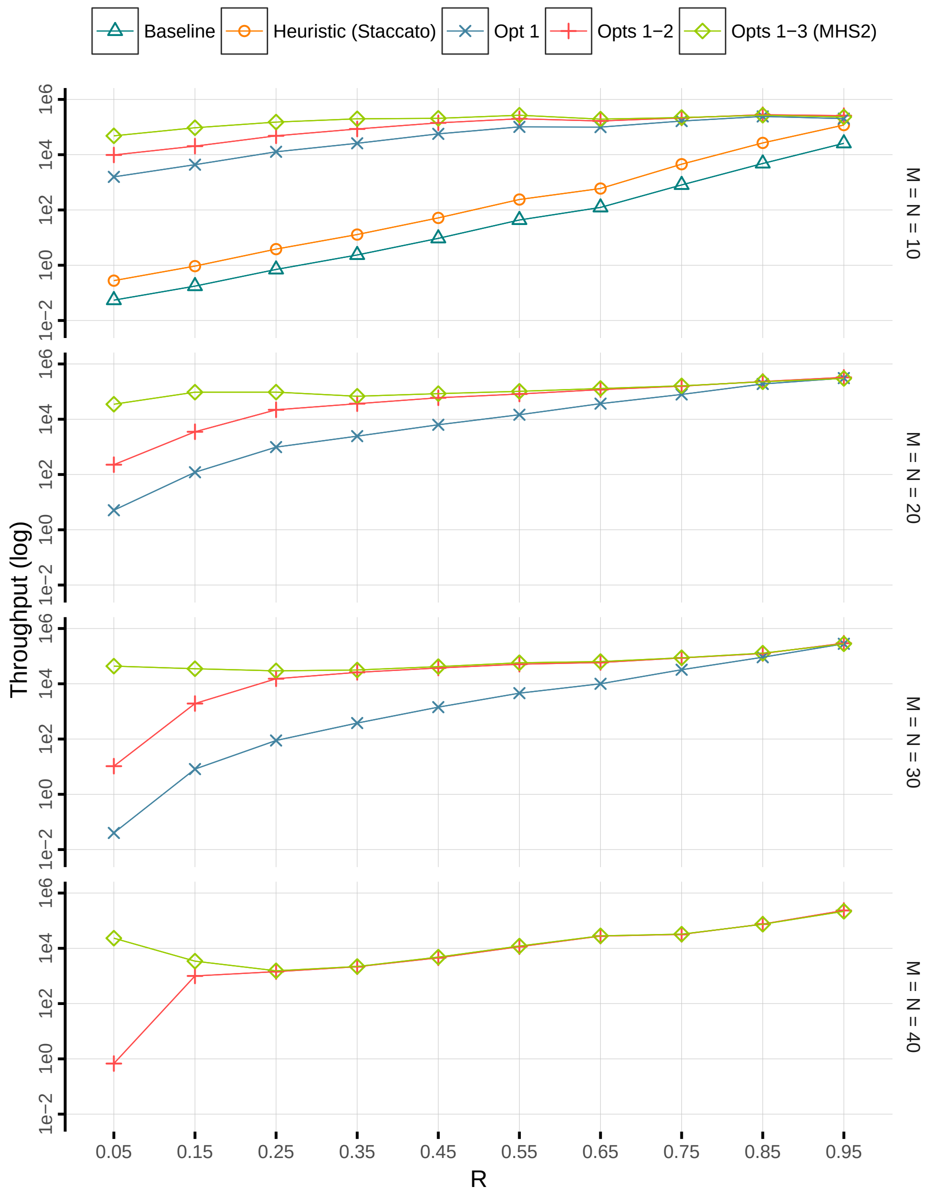


Figure 2.8: Small problems' results

1. All algorithms have similar performances for large R values.
2. All optimizations are more effective for smaller R values.
3. Optimizations 1 and 2 have a considerable effect for all R values whereas optimization 3 is only effective for small R values.

Pattern 1 can be explained by noting that the average *MHS* cardinality is negatively correlated with the R value. It follows then that, problems with large R values have shallow search trees and, consequently, the optimizations, which focus on improving the search tree exploration, have a lesser performance impact.

Conversely, pattern 2 can be explained using the complimentary argument. For small R values, as the search trees become deeper, the non-optimized algorithms perform a large amount of unnecessary divide tasks, thus leaving (exponentially) more room for improvements.

To explain pattern 3, we shall look at the optimizations individually:

- Optimization 1 prevents the exploration of paths composed of the same elements although in different orders. Such inefficiencies occur whenever the *MHSs* are composed of more than 1 element.
- Optimization 2 reduces the overhead associated with the heuristic calculation. The overhead reduction also occurs whenever the *MHSs* are composed of more than 1 element.
- Optimization 3 performs a look-ahead verification to assess whether the current sub-tree is a dead-end (*i.e.*, no *MHSs* will be generated in such sub-tree) and terminate the exploration if a dead end is reached. The impact of this optimization is contingent on how far ahead it detects the dead-end which, in turn, is dependent on the deepness of the search tree. As the deepness of the search tree is negatively correlated with R, this optimization is more effective for small R values.

2.5.2 Large problems

The second benchmark is aimed at evaluating the impact of each optimization for large problems where it is impractical to calculate all *MHSs* ($M = N = 10^3$, and $R \in \{0.05, \cdots, 0.95\}$). In all the following test cases a time based cutoff of 30 seconds was enforced.

The first two plots in Figure 2.9 present both the percentage of *HSs* that were minimal (*MHS*%) and the average *HS* cardinality for each of the 5 implementations. For large problems, we observe that the heuristic plays an important role in assuring that the computed *HSs* are in fact minimal (98% *vs.* 0% minimality for *Heuristic* and *Baseline*, on average). Even though the *MHS*% of *Opts* $1-2$ and *Opts* $1-3$ is lower when compared to *Opt* 1, we can see that the average *HS* cardinality of the former implementations is comparable to cardinality of the later, implying that the number of extraneous elements in the non-minimal *HSs* is small (specially when compared to *Baseline*).

The remainder of Figure 2.9 presents the number of *HSs* as well as the throughput for all implementations. While *Baseline*, *Heuristic*, and *Opt* 1 compute 500 *HSs* on average, the remaining implementations compute 93000 *HSs* on average ($186\times$ better). Taking into account the number of computed *HSs*, despite the lower *MHS*%, the absolute number of *MHSs* for optimizations 2 and 3 is effectively larger than the number of *MHSs* calculated by the remainder algorithms.

Finally, it is interesting to note that even though we increased the problem size by $25\times$ factor, the throughput of *MHS*^{2} is comparable to the throughput observed in Figure 2.8. We can conclude that the fully optimized *MHS*^{2} scales to large problems more efficiently than *Staccato*.

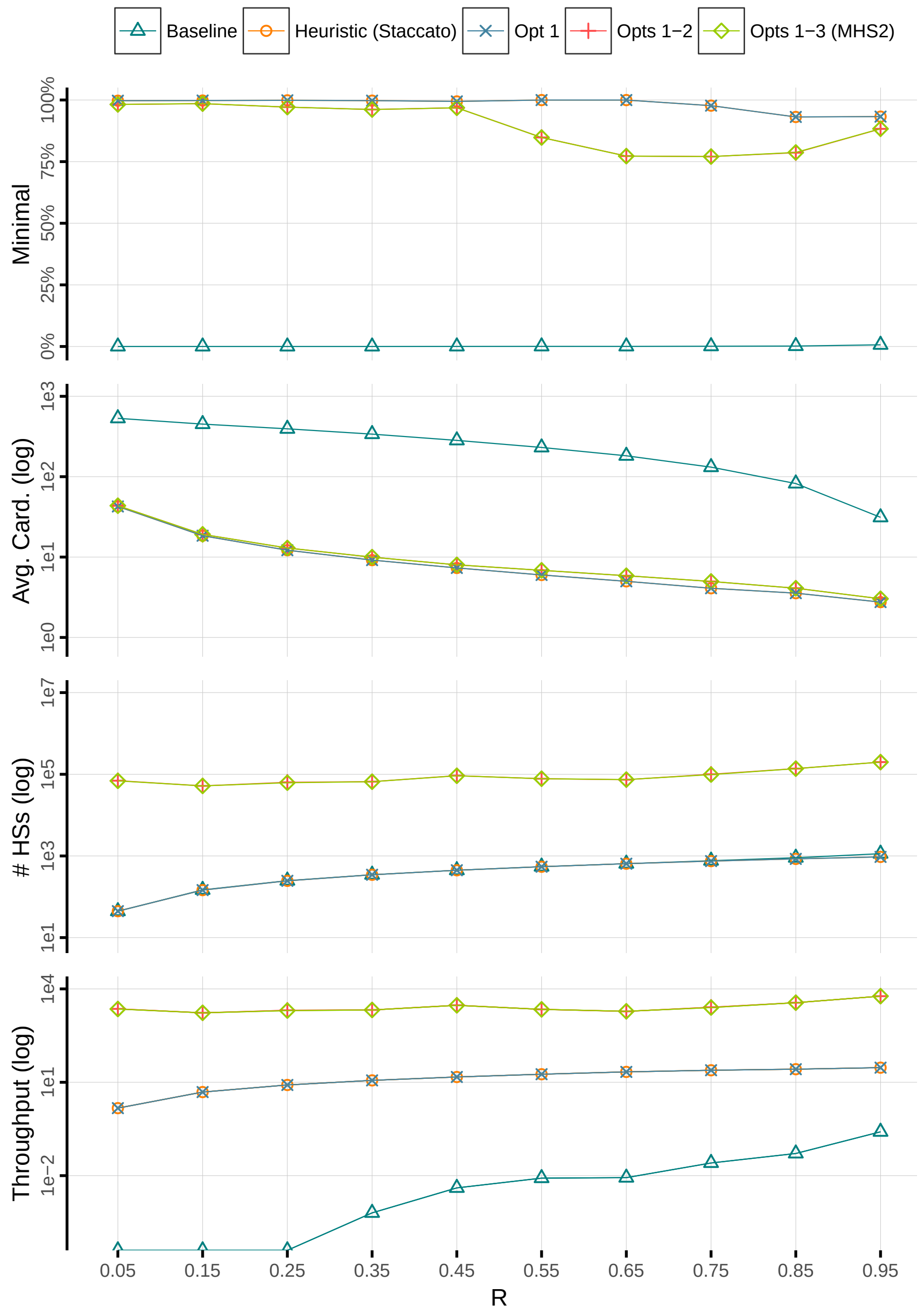


Figure 2.9: Large problems' results

2.6 Summary

In this chapter we successfully addressed the limitations presented in Section 1.3.1.1 (page 20) thereby positively answering Research Question 1 (page 20). Concretely, in this chapter:

- We proposed 3 optimizations to *Staccato* (Section 2.1, page 27):
  - The first optimization prevents multiple examinations of the same set.

    - The second optimization preemptively filters elements of U not contained in any set in S.
    - The third optimization prevents the examination of branches such that there is at least one set that cannot be hit by any element in U.
- We provided a formal analysis of the proposed algorithm:
    - We formally proved that the optimized algorithm is both sound and complete (Section 2.2.1, page 30).
    - We gave some intuition about the complexity of the algorithm's different operations (Section 2.2.2, page 32).
- We analyzed the implementation details of our reference implementation, which is available at `https://github.com/npcardoso/MHS2` (Section 2.3, page 32).
- We discussed some practical aspects of our *MHS* generation algorithm:
    - We discussed how the heuristic influences the algorithm's performance (Section 2.4.1, page 37).
    - We discussed how to improve the algorithm's performance in the presence of ambiguity groups (Section 2.4.2, page 38).
    - We discussed how to improve the algorithm's performance when solving non-minimal problems (Section 2.4.3, page 40).
- We presented the conducted benchmarks showing that our algorithm:
    1. Performs $34000\times$ faster than *Staccato* for small problems (Section 2.5.1, page 42).
    2. Performs $186\times$ faster than *Staccato* for large problems (Section 2.5.2, page 44).
    3. Exhibits a similar throughput in both small and large problems.

The faster algorithm enables the exploration of a larger number of *HS*, increasing the likelihood of actually finding the "best" *MHS* for a particular instance of the problem. In the particular case of *SFL*, this improvement translates into:

- Better diagnostic accuracy when setting a time-based cutoff, due to the fact that calculating more candidates increases the likelihood of finding the correct diagnostic candidate.
- Smaller diagnostic latency when setting a solution size cutoff, due to the fact that calculating a fixed number of diagnostic candidates takes less time with *MHS*^{2} than with *Staccato*.

```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>>---.-
----.+++++++++++.<<++.<.>-----.<.>>>---.>+.+++++++++++++++++.<++++++++
+++++++++.>------..<++++.>.---.+++++++++++++++++.<----.>------.<++++++
++.>-----.-.[>]<[[-]<]+->->->-<>-->>>+>>-+<>->+++->>-+->>><><<<>+<<<><
```

3 *MHS*^{2} – Parallelization

In this section we present the parallelized version of *MHS*^{2}. As explained in Section 1.3.1.2 (page 20), our goal in this chapter is to enable the computation of *MHSs* in parallel or even distributed environments.

This chapter is divided as follows. First, we introduce our *MHS* generation algorithm. Second, we evaluate the performance of our algorithm.

3.1 Approach

The parallel algorithm can be seen as a Map-Reduce algorithm [Dean and Ghemawat, 2004] (Figure 3.1). The map task (Algorithm 3.1) is a modified version of Algorithm 2.1 (page 28).

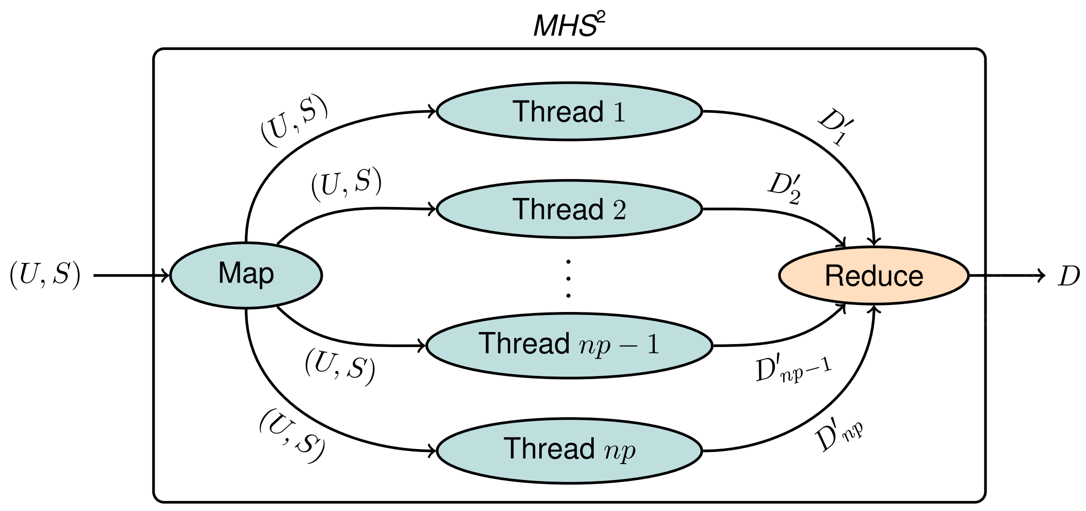


Figure 3.1: *MHS*^{2} Map-Reduce workflow

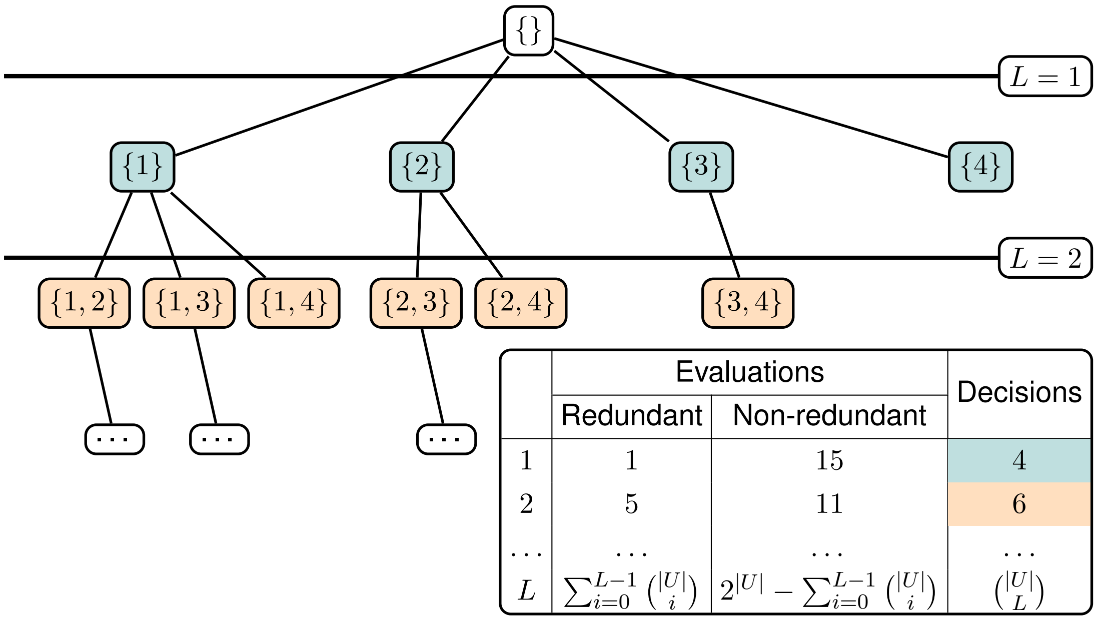
Algorithm 3.1 *MHS²* – Map task

```
Inputs: $(U, S, d = \emptyset, D = \emptyset)$
Parameters: $(L, \boldsymbol{Skip}, k, np)$
Output: Minimal hitting set collection $D'_k$
1  if $\exists_{s \in S} : s \cap U = \emptyset$ then
2      return $D$
3  if $S \neq \emptyset$ then
4      $U \leftarrow U \setminus \{j \mid j \in U \wedge (\nexists_{s \in S} : j \in s)\}$
5      for $j \in \boldsymbol{Rank}(U, S)$ do
6          if $|d| + 1 = L \wedge \boldsymbol{Skip}(k, np)$ then
7              $U \leftarrow U \setminus \{j\}$                                # Opt 1
8              continue
9          $S' \leftarrow \{s \mid s \in S \wedge j \in s\}$
10         $U \leftarrow U \setminus \{j\}$
11         $D \leftarrow \boldsymbol{MHS}^2(U \setminus \{j\}, S \setminus S', d \cup \{j\}, D)$
12 else
13     if $\nexists_{d' \in D} : d' \subseteq d$ then
14         $D \leftarrow D \setminus \{d' \mid d' \in D \wedge d \subseteq d'\}$
15         $D \leftarrow D \cup \{d\}$
16 return $D$
```

In contrast to the sequential algorithm, we added a parameter L that sets the *fork-level*, *i.e.*, the number of calls in the stack (or equivalently, $|d| + 1$), at which the computation is divided among the processes/threads. When a process of the distributed algorithm reaches the target level L, it uses a load division function ($\boldsymbol{Skip}$) to decide which elements of the ranking to skip or to analyze. The value of L implicitly controls the granularity of decision of the load division function at the cost of performing more redundant calculations (Figure 3.2). Implicitly, by setting a value $L > 1$, all processes redundantly compute all *HS* such that $|d| < L$.

In parallel processing environments, it is desirable to minimize the amount of time processes are idle. Idleness periods may arise when a process completes its work before other threads on which subsequent processing stages depend on. Concretely, for Map-Reduce algorithms, it is desirable that all map complete their jobs simultaneously, thus minimizing idle periods and improving resources usage. Taking this fact into account, a correct selection of the load distribution function (*i.e.*,$\boldsymbol{Skip}$) is critical for obtaining good performance.

With regard to the load division, we propose two different approaches. The first, referred to as *Stride*, consists in assigning elements of the ranking to processes cyclically (Figure 3.3a). Formally, a process $p_{k \in \{1..np\}}$ is assigned to a branch b_l of the search tree



	Evaluations		Decisions
	Redundant	Non-redundant	
1	1	15	4
2	5	11	6
$\dots$	$\dots$	$\dots$	$\dots$
L	$\sum_{i=0}^{L-1}\binom{\lvert U\rvert}{i}$	$2^{\lvert U\rvert} - \sum_{i=0}^{L-1}\binom{\lvert U\rvert}{i}$	$\binom{\lvert U\rvert}{L}$

Figure 3.2: L parameter intuition

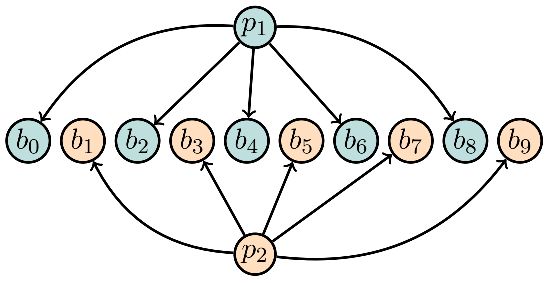
if:

$$l \mod np = k - 1 \tag{3.1}$$

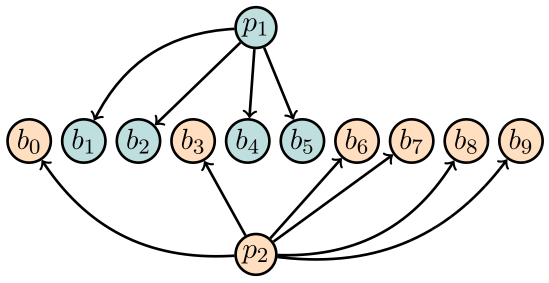
The second approach, referred to as *Random*, uses a pseudo-random generator to divide the computation (Figure 3.3b). This random generator is fed into a uniform distribution generator that assures that, over time, all p_k get assigned a similar number of elements although in random order. This method is aimed at obtaining a more even distribution of the problem across processes than *Stride* (Figure 3.4). Formally, a process $p_{k\in\{1..np\}}$ is assigned to an element of the ranking if:

$$\boldsymbol{rand}() \mod np = k - 1 \tag{3.2}$$

A particularity of this approach is that the seed of the pseudo random generator must be shared across every process to assure that no further communication is needed.



(a) *Stride* function



(b) *Random* function

Figure 3.3: $\boldsymbol{Skip}$ functions intuition

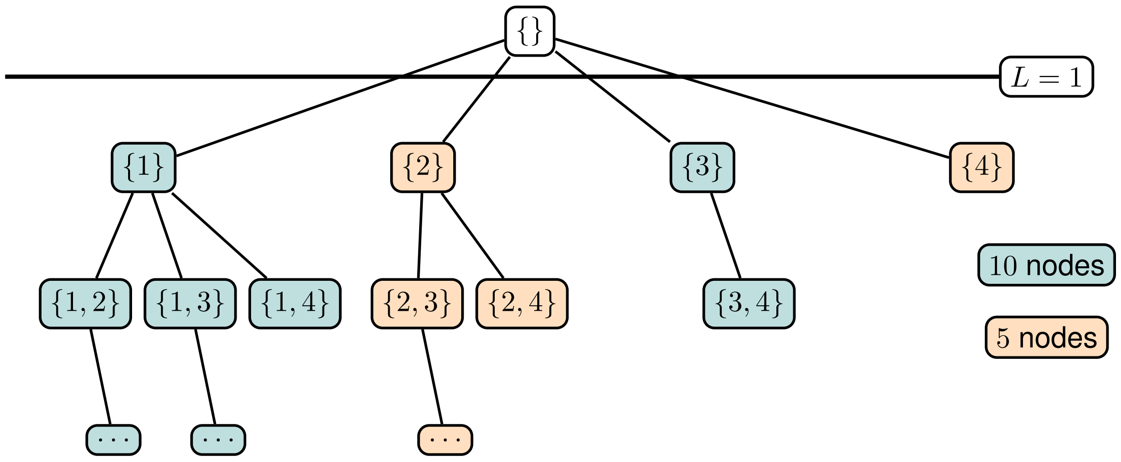


Figure 3.4: Balancing problem when using the *Stride* function

Algorithm 3.2 *MHS²* – Reduce task

Inputs: $(D'_1, ..., D'_{np})$

Output: Minimal hitting set collection D

```
1 $D \leftarrow \emptyset$
2 $D' \leftarrow \textbf{Sort}(\bigcup_{k=1}^{np} D'_k)$
3 for $d \in D'$ do
4     if $\nexists_{d' \in D} : d' \subseteq d$ then
5         $D \leftarrow D \cup \{d\}$

6 return $D$
```

Finally, the reduce task (Algorithm 3.2), responsible for merging all partial *MHS* collections $D'_{k\in\{1..np\}}$ originating from the map task. The reducer works by merging all *HSs* in a list, ordered by cardinality. The ordered list is then iterated, adding all *MHSs* to D. As the *HSs* are inserted in an increasing cardinality order, it is not necessary to look for subsumable *HSs* (line 14 in Algorithm 3.1) in D, thus improving the algorithm's performance.

3.2 Benchmark

To assess the performance of our algorithm we conducted a set of benchmarks in a setup similar to the one presented in Section 2.5 (page 41). To evaluate the gains introduced by the parallelization of the candidate generation algorithm we use the *speedup* metric. In parallel computing, speedup refers to how much a parallel algorithm optimizes an arbitrary metric in relation to the corresponding sequential algorithm. Concretely, the speedup metric is defined as:

$$S_p = \frac{M_1}{M_p} \tag{3.3}$$

where:

- p is the number of processors;
- M_1 is the value of an arbitrary metric for the sequential algorithm;
- M_p is the value of the same arbitrary metric for the parallel algorithm with p processors.

In small problems where *all* minimal candidates are calculable, the observed metric is the execution time, evaluating how much faster is the parallel version of the algorithm when compared to its sequential counterpart. In large problems where only a subset of the solution is calculable, we evaluate the algorithm's throughput improvement by registering the number of minimal candidates calculated by the parallel algorithm with different number of processors and constrained by the same calculation time.

A normalized version of the speedup is called *efficiency* and is defined as:

$$E_p = \frac{S_p}{p} = \frac{M_1}{p \times M_p} \tag{3.4}$$

The efficiency value typically ranges between zero and one and estimates how well-utilized the processors are in solving the problem, compared to how much effort is wasted in communication and synchronization.

3.2.1 Small Problems

The first parallelization benchmark, is aimed at evaluating the behavior of *MHS*² for small problems ($M = N = 40$, $R \in \{0.25, 0.5, 0.75\}$).

In Figure 3.5, we plot the speedup as well as the efficiency for both the *Stride* and *Random* load distribution functions. Also, Figure 3.5 shows the run-times[1] as well as the throughput[2] for both load distribution functions.

The analysis of Figure 3.5 shows different speedup/efficiency patterns for different R values, which are due to the large variation in the run times: ≈ 200, 5.7 and 0.1 seconds for $R = \{0.25, 0.5, 0.75\}$, respectively. On the one hand, when the run time is small (*i.e.*, for large R values), the parallelization overhead has a higher relative impact in the algorithm's performance. On the other hand, when the run time becomes larger due to the increase of both the cardinality and the amount of *MHSs* (*i.e.*, for small R values), the relative parallelization overhead becomes almost insignificant.

[1] In this plot, each cluster is composed of 1 data point per test case (100 data points for each load distribution function). The horizontal displacement inside each cluster was only added to improve the visualization of the results.

[2] In this section the throughput represents the average throughput per thread. Concretely it is calculated as $\frac{|D|}{np \times t}$, where t represents the cut-off time.

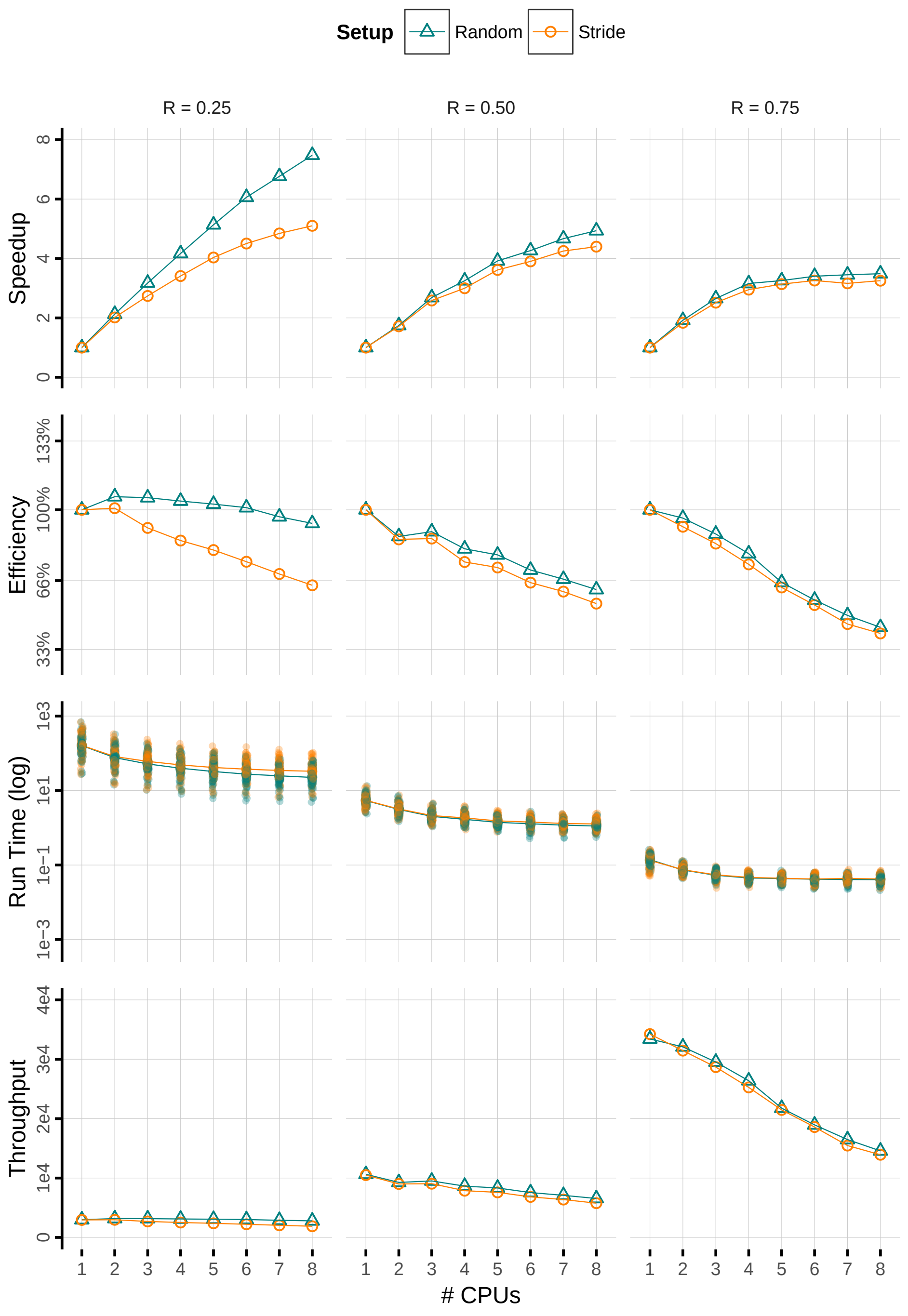


Figure 3.5: Small problems' results

For $R = 0.25$ and in contrast to $R \in \{0.5, 0.75\}$, the speedup/efficiency of *Random* is superior to the performance of *Stride*. The reason for such a difference in efficiency is that the *cool-down period* (*i.e.*, the period in which at least one process is idle and another is active) of *Stride* was longer than the one observed for *Random*. The smaller *cool-down* period of *Random* shows that the usage of a stochastic approach was successful at evenly dividing an unbalanced search tree among threads, leading to a better load division.

Finally, we observe that *Random* experienced *super-linear* speedup (i.e, efficiency above 100%) for some of the test cases. This pattern emerges due to the fact that the complexity of the operations performed on D (lines 13 – 15 in Algorithm 3.1), is not linear in the number of elements stored in it. As a consequence, by reducing size of D to $\frac{1}{np}$ effectively reduces the cost of the operations by a factor greater than np.

3.2.2 Large Problems

The second benchmark is aimed at evaluating the behavior of *MHS*^{2} for large problems ($M = N = 10^3$, and $R \in \{0.25, 0.5, 0.75\}$).

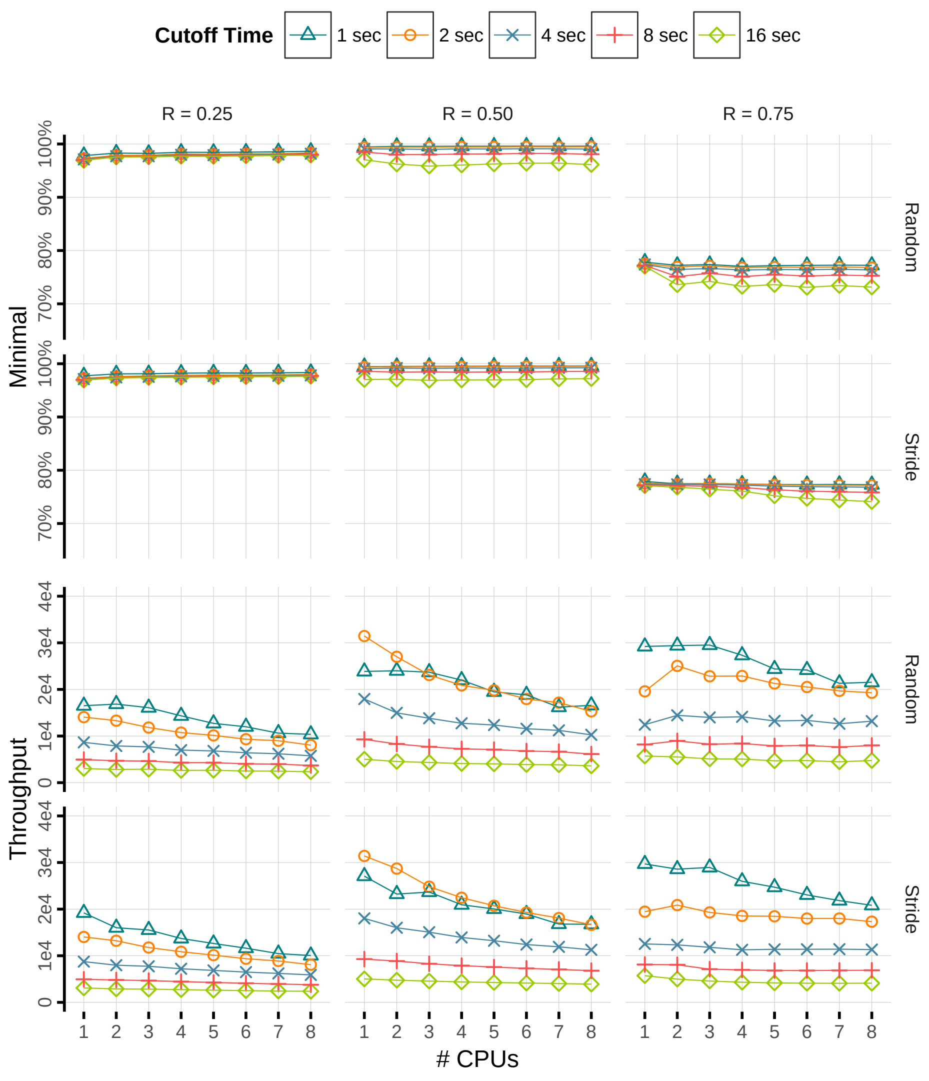


Figure 3.6: Large problems' results

Figure 3.6 shows the minimality percentage and throughput for the large problems when using time-based cutoffs of $1, 2, 4, 8,$ and 16 seconds per process with a varying number of processes.

It appears that, for large problems, there is no significant difference in terms of the amount of generated *MHSs* between *Random* and *Stride*. By performing a two-tailed T-test we determined, with a 99% confidence interval, that both approaches should have, on average, equal throughputs.

Regarding the minimality percentage of both approaches, we observed similar results to those presented in Figure 2.9 (page 45): 97% for $R = \{0.25, 0.5\}$, while for $R = 0.75$ the percentage decreases to around 75%. Also, the throughput is consistent with all previous benchmarks.

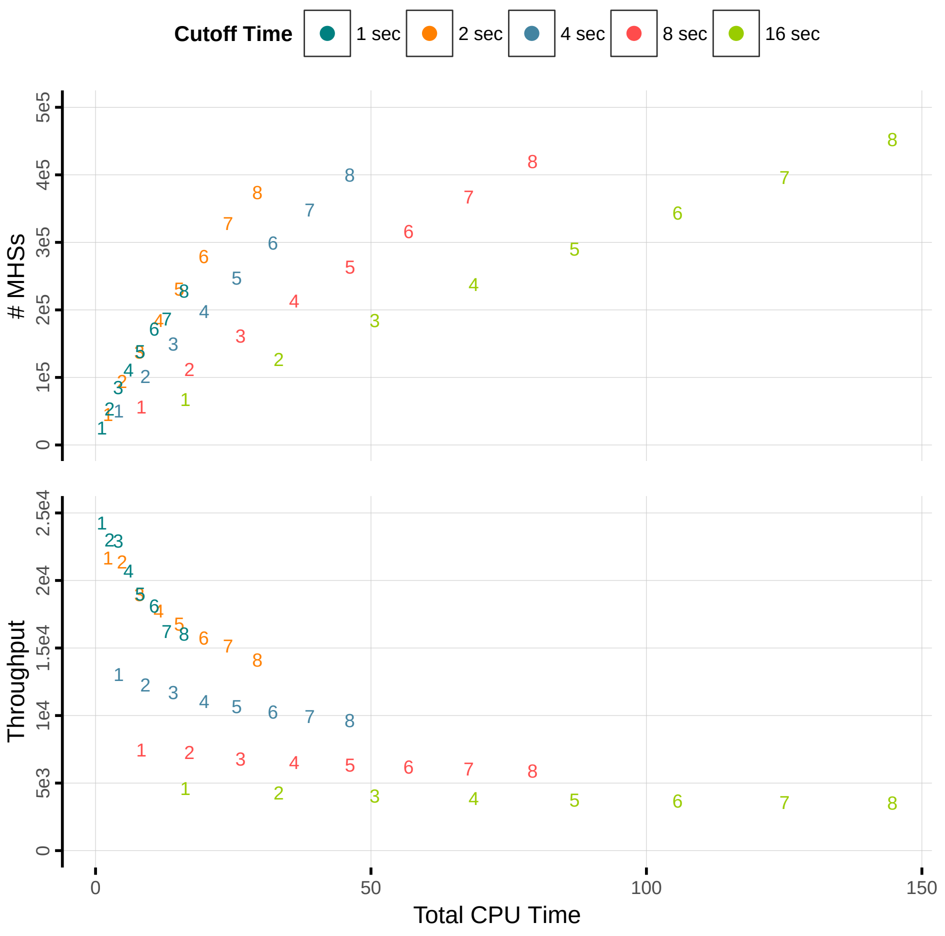


Figure 3.7: Large problems' results with x-axis transformation

Figure 3.7 shows the number of *MHSs* and throughput after applying a data transformation. In spite of using the number of processes as the x-axis, we use the total *CPU* time for all processes (rt_{cpu}), which is calculated as:

$$rt_{cpu} = rt_{wall} \times np \tag{3.5}$$

where rt_{wall} is the perceived run time for the algorithm's execution when using a "wall" clock[3]. Each data point is represented by a number encoding the number of used *CPUs*[4].

[3]Since rt_{wall} accounts for both the map and reduce phases whereas the cutoff is only applied to the map phase, it is always larger than the cutoff value.

[4]The points' y-axis values are the averages over the different R values for the data points in Figure 3.6.

We can see that, for instance, running the algorithm for 8 seconds on 8 *CPUs* accounts approximately for the same total run-time as running the algorithm for 16 seconds on 4 *CPUs*.

Using this plot we can easily see that, for a given total calculation time, it is, on average, preferable to divide the task among the maximum possible number of *CPUs* (eventually, given enough *CPUs*, this trend should hit a ceiling). As an example, we can see that 8 threads running for 2 seconds produce more *MHSs* than 2 threads running for 16 despite using approximately the same total *CPU* time. This result is in accordance with the super-linear speedup observed in Figure 3.5.

3.3 Summary

In this chapter we successfully addressed the limitations presented in Section 1.3.1.2 (page 20) thereby positively answering Research Question 2 (page 20). Concretely, in this chapter:

- We proposed a parallelization approach for *MHS*^{2} (Section 3.1, page 47):
  - We proposed an approach based on the Map-Reduce paradigm, thereby being applicable in both parallel and distributed environments.
  - We proposed two load distribution strategies: *Stride* and *Random*.
- We presented the conducted benchmarks showing that:
  1. When computing full solutions for complex enough problems, the algorithm performs at approximately 100% efficiency even when using 8 worker threads (Section 3.2.1, page 51).
  2. When solving simple problems for which the algorithm's run-time is small, the efficiency quickly drops with the increase of worker threads (see Section 3.2.1, page 51).
  3. On average, the algorithm was more efficient when the problem was divided across the maximum number of threads (see Section 3.2.2, page 53).
  4. The algorithm exhibits a similar throughput per thread in both small/large problems and sequential/parallel setups (Figures 2.8, 2.9, 3.5 and 3.6, pages 43, 45, 52 and 53, respectively).

The usage of parallel processing enables the exploration of a larger number of *HS*, increasing the likelihood of actually finding the "best" *MHS* for a particular instance of the problem. In the particular case of *SFL*, and as stated in Section 2.6 (page 45), this improvement translates into:

- Better diagnostic accuracy when setting a time-based cutoff, due to the fact that calculating more candidates increases the likelihood of finding the correct candidate.

- Smaller diagnostic latency when setting a solution size cutoff, due to the fact that calculating a fixed amount of diagnostic candidates takes less time with *MHS*^{2} than with *Staccato*.

```
-[------->+<]>---.[++>-------<]>.+++++..--[->+++<]>+.+++++.---------.+
++++++.-><--<><><---+++>--<+>><+>><-<>+><-+-++>-+---><<++-+<><<+<>+-+>
```

4 *Fuzzinel*

In this chapter we focus on addressing the *SFL* limitations explained in Section 1.3.2.1 (page 21). Concretely, we propose a generalization to the *SFL* diagnostic framework, dubbed *Fuzzinel*[1], capable of diagnosing fuzzy errors with better accuracy. This chapter is divided as follows. First, we introduce our enhanced *SFL* approach. Second, we evaluate the performance of our approach.

4.1 Approach

The problem of handling fuzzy errors can be divided in two sub-problems: detection and diagnostic problems. In this section we discuss how to solve both problems in the context of *SFL*.

4.1.1 Fuzzy Error Detection

Existent approaches to error detection (*e.g.*, [Casanova et al., 2013]) make use of first-order logic descriptions of the correct behavior of the system (weak-fault models) to assign transactions to one of two possible sets: the pass set and the fail set (P and F respectively, where $F = \overline{P}$). A consequence of such fault models is the *crisp* distinction between correct and incorrect system states. While this classical logic description enables an accurate representation of functional errors, it is unable to accurately represent a large variety of non-functional errors.

[1] *Fuzzinel* is a combination of 50% "Fuzzy" and 50% "Barinel", which is the name of the original *SFL* algorithm.

As an example, consider a type of non-functional error that, informally, is described by the statement "The system is slow". Even though we can easily relate the slowness of the system to an appropriate metric (*e.g.*, response time), it is not easy to define a crisp boundary in this same metric to distinguish acceptable and slow transactions. By setting a crisp boundary at, for instance, 1 second, a response time of 0.9999 seconds would be considered to be correct whereas a marginally superior response time would be considered incorrect. Also, a response time of 0.9999 seconds would result in the same type of error information (pass) as a smaller response time even though the larger response time may represent an error symptom.

To overcome the expressiveness limitation of the classical logic error detection mechanisms, we propose their generalization using fuzzy logic [Zadeh, 1965]. Fuzzy logic extends the notion of binary set membership by introducing the concept of membership functions, denoted $\boldsymbol{\mu_A}$ (membership function for set A), mapping a set of problem-specific variables onto the continuous interval $[0,1]$, where the endpoints of 0 and 1 conform to no membership and full membership, respectively. In the context of error detection, the concept of fuzzy membership enables the representation of 3 types of system states: correct ($\boldsymbol{\mu_{\widetilde{F}}}(x) = 0$), incorrect ($\boldsymbol{\mu_{\widetilde{F}}}(x) = 1$) and degraded ($0 < \boldsymbol{\mu_{\widetilde{F}}}(x) < 1$). Intuitively, since $F = \overline{P}$, in the new fuzzy error model a degraded transaction exhibits both correct and incorrect behaviors simultaneously, however with different degrees.

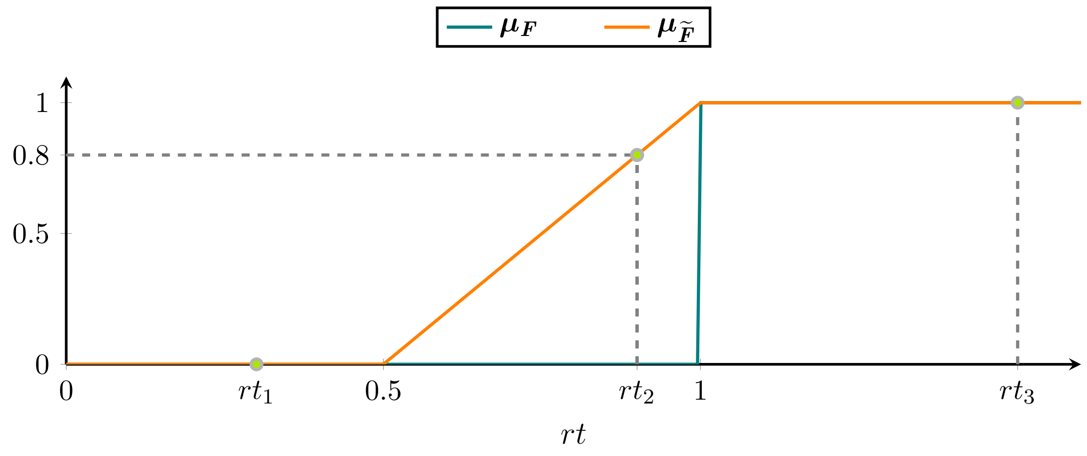


Figure 4.1: Crisp *vs.* fuzzy sets

As an example, consider the crisp fail set containing all response times (rt) above 1 second. This same set could be represented in terms of a membership function as (Figure 4.1):

$$\boldsymbol{\mu_F}(rt) = \begin{cases} 0 & , rt \leq 1 \\ 1 & , rt > 1 \end{cases} \tag{4.1}$$

To achieve the goal of representing the degraded state, consider that all response times below 0.5 seconds are considered correct and all times above 1 second incorrect. Furthermore, consider that the amount of degradation follows a linear pattern between those two thresholds. The fuzzy fail set membership function representing this particular type of error could be defined as:

$$\mu_{\widetilde{F}}(rt) = \begin{cases} 0 & , rt < 0.5 \\ 2 \cdot rt - 1 & , 0.5 \leq rt \leq 1 \\ 1 & , rt > 1 \end{cases} \tag{4.2}$$

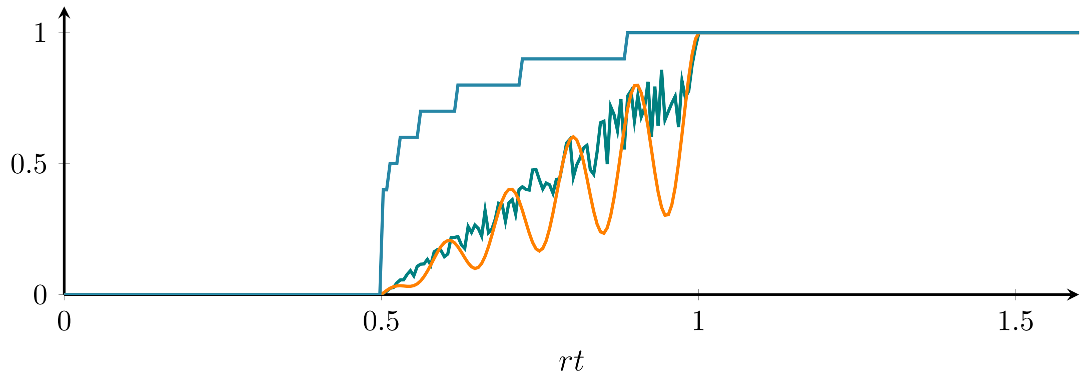


Figure 4.2: Arbitrary membership functions

It is important to note that the assumption of linear degradation introduced in Equation (4.2) was only made for simplicity. In real-world scenarios, the membership functions are application dependent and can exhibit arbitrary patterns. From our approach's point-of-view, the membership functions are treated as black-boxes. Figure 4.2 shows a set of alternative membership functions for the error previously described. Despite their odd shapes they are acceptable membership functions, provided that they correctly describe the error state of the transaction.

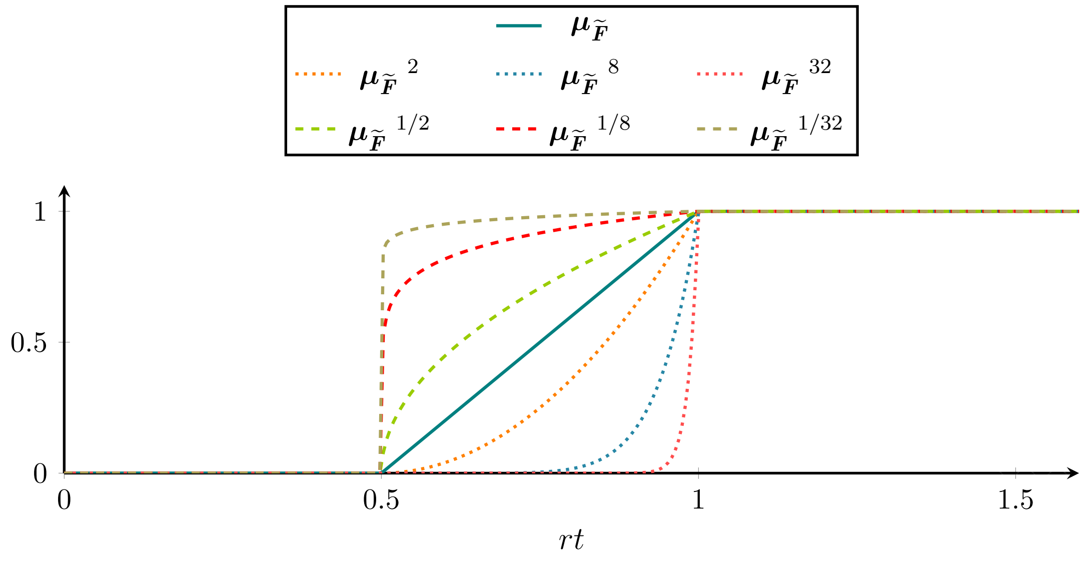


Figure 4.3: Error detection sensitivity intuition

Furthermore, it is possible to change the error detection sensitivity by raising $\mu_{\widetilde{F}}$ to an exponent, as depicted in Figure 4.3. We can see that by raising $\mu_{\widetilde{F}}$ to an exponent in the interval $]1, +\infty[$, the error detection becomes less sensitive to errors in the fuzzy zone (*i.e.*, the error value in the fuzzy zone becomes smaller than the original). In contrast, by raising $\boldsymbol{\mu}_{\widetilde{\boldsymbol{F}}}$ to an exponent in the interval $]0, 1[$, the error detection becomes more sensitive to errors in the fuzzy zone. In fact, when the exponent tends to either $+\infty$ or 0, the fuzzy membership becomes a binary membership and errors in the fuzzy zone become passes and fails, respectively.

i	rt	A		e	
		c_1	c_2	$\boldsymbol{\mu}_{\boldsymbol{F}}(rt_i)$	$\boldsymbol{\mu}_{\widetilde{\boldsymbol{F}}}(rt_i)$
1	0.3	.	●	0	0
2	0.9	●	.	0	0.8
3	1.5	●	●	1	1

Figure 4.4: Fuzzy error hit spectrum example

To conclude the illustration of the fuzzy error detection process, consider the spectrum presented in Figure 4.4, which also contains the run-times for each transaction (marked in Figure 4.1). From this spectrum we can see that, in particular for t_2, the crisp error vector neglected an error symptom whereas the fuzzy error vector categorized that same transaction as being 80% degraded.

Finally, it is worth mentioning that, even though our simplistic example only used one variable to determine the error value of the transactions, the error value can be a function of an arbitrary number of variables. Furthermore, a system may have several membership function for different types of transactions.

4.1.2 Fuzzy Error Diagnosis

Using fuzzy logic to detect errors, it is possible to assert that a particular transaction is 80% degraded (*i.e.*, $\boldsymbol{\mu}_{\widetilde{\boldsymbol{F}}} = 0.8$ and consequently $\boldsymbol{\mu}_{\widetilde{\boldsymbol{P}}} = 0.2$). The remaining challenge consists in integrating this additional knowledge in the diagnostic process.

As an example consider again the spectrum depicted in Figure 4.4. Using the approach explained in Section 1.2.2 (*i.e.*, using $e = \boldsymbol{\mu}_{\boldsymbol{F}}$), it follows that the candidates[2] $d_1 = \{c_1\}$ and $d_2 = \{c_2\}$ are ranked equally. However, intuitively, we would expect d_1 to be ranked ahead of d_2 since transaction t_2, in which component c_1 was involved, shows error symptoms whereas t_1 does not.

To solve this limitation we make use of the concept of probability of a fuzzy event [Zadeh, 1968]. The probability of a fuzzy event is defined as:

$$\boldsymbol{Pr}(\alpha) = \sum_{x \in \Omega} \boldsymbol{\mu}_{\boldsymbol{x}}(\alpha) \cdot \boldsymbol{Pr}(x) \tag{4.3}$$

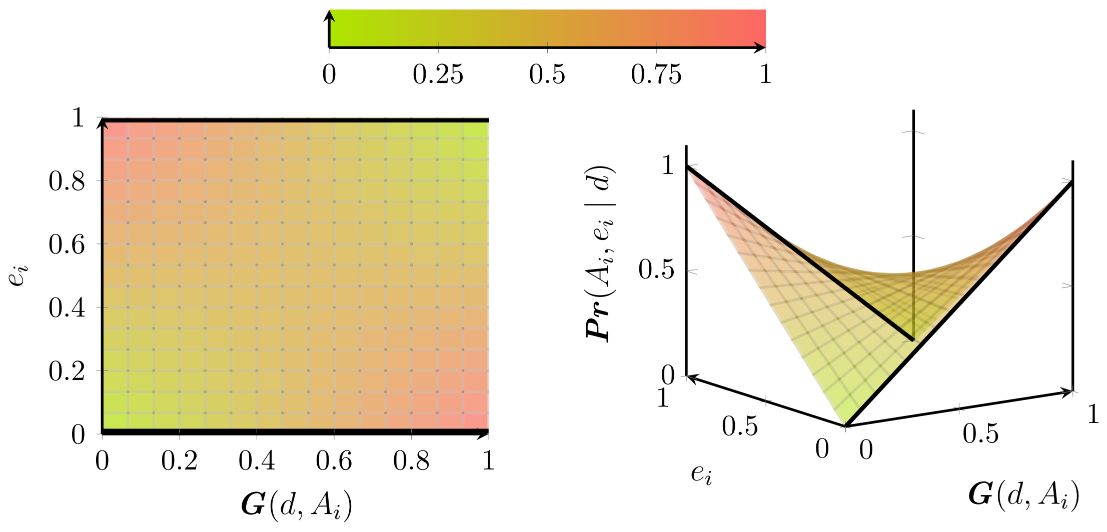
where α is an arbitrary event, and Ω is a set representing all the possible outcomes of α.

[2]The candidates for the fuzzy approach are calculated by setting a threshold for $\boldsymbol{\mu}_{\widetilde{\boldsymbol{F}}}$ to discretize transactions in terms of pass/fail. In this example we use the threshold $\boldsymbol{\mu}_{\widetilde{\boldsymbol{F}}} = 1$.

Mapping this definition to the problem at hand, we generalize Equation (1.14) as:

$$\boldsymbol{Pr}(A_i, e_i \mid d) = \underbrace{e_i \cdot \big(1 - \boldsymbol{G}(d, A_i)\big)}_{x=F} + \underbrace{(1 - e_i) \cdot \boldsymbol{G}(d, A_i)}_{x=P} \tag{4.4}$$

where the first part of the equation ($x = F$) accounts for the incorrect behavior whereas the second part ($x = P$) accounts for the correct behavior. In contrast to Equation (1.14), this generalization is valid for fuzzy error values (*i.e.*, $e \in]0, 1[$). Figure 4.5 shows the plot of the Equation (4.4) with respect to e_i and $\boldsymbol{G}(d, A_i)$. For comparison, we also plot Equation (1.14) with thick black lines.



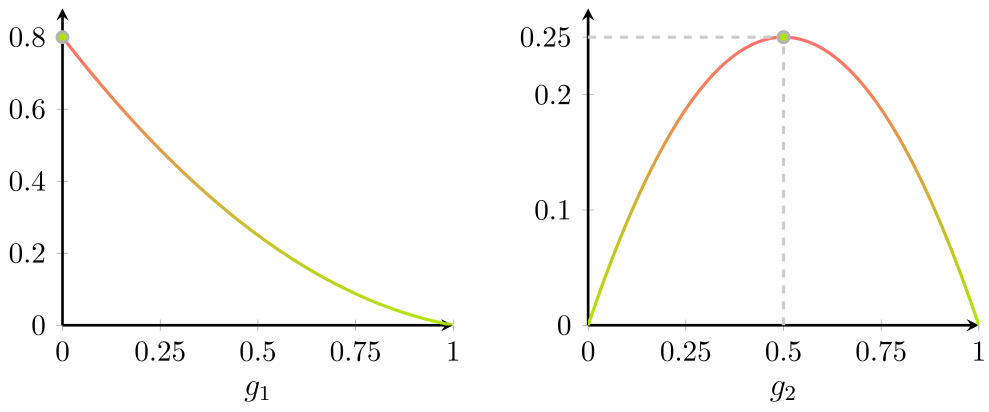
(a) 2D view **(b)** 3D view

Figure 4.5: Likelihood function plot

Using the above generalization, the probabilities of the two candidates are calculated as follows:

$$\begin{aligned}\boldsymbol{Pr}(A, e \mid d_1) &= \underbrace{(0.8 \cdot (1 - g_1) + (1 - 0.8) \cdot g_1)}_{t_2} \\ &\times \underbrace{(1 \cdot (1 - g_1) + (1 - 1) \cdot g_1)}_{t_3} \\ &= \underbrace{(0.8 \cdot (1 - g_1) + 0.2 \cdot g_1)}_{t_2} \times \underbrace{(1 - g_1)}_{t_3}\end{aligned} \tag{4.5}$$

$$\begin{aligned}\boldsymbol{Pr}(A,e \mid d_2) &= \underbrace{(0\cdot(1-g_2)+(1-0)\cdot g_2)}_{t_1}\\ &\times \underbrace{(1\cdot(1-g_2)+(1-1)\cdot g_2)}_{t_3}\\ &= \underbrace{g_2}_{t_1}\times\underbrace{(1-g_2)}_{t_3}\end{aligned} \tag{4.6}$$



(a) $\boldsymbol{Pr}(A,e \mid d_1)$ **(b)** $\boldsymbol{Pr}(A,e \mid d_2)$

Figure 4.6: Likelihood plots

By performing an *MLE* for both functions (Figure 4.6) it follows that $\boldsymbol{Pr}(A,e \mid d_1)$ is maximized for $g_1 = 0$ and $\boldsymbol{Pr}(A,e \mid d_2)$ for $g_2 = 0.5$. Applying the maximizing values to both expressions, it follows that $\boldsymbol{Pr}(d_1 \mid A,e) \approx 8\times10^{-4}$ and $\boldsymbol{Pr}(d_2 \mid A,e) \approx 2.5\times10^{-4}$. Such probabilities entail the ranking $\langle d_1, d_2\rangle$, which breaks the ambiguity between d_1 and d_2, thus improving the diagnostic accuracy.

4.2 Benchmark

In this section we describe our benchmark approach and discuss results.

4.2.1 Setup

To evaluate our approach we make use of a simulator as proposed in [Chen et al., 2013][3]. The simulator provides functions to describe and execute a probabilistic model of an arbitrary system, thereby gathering the required spectra. The authors showed that, in the scope of *SFL*, the benchmark results for both real and synthetic data are comparable. This is mainly due to the fact that, since system is highly abstracted, the spectra generated by real and simulated systems is similar.

[3]`https://github.com/SERG-Delft/sfl-simulator`

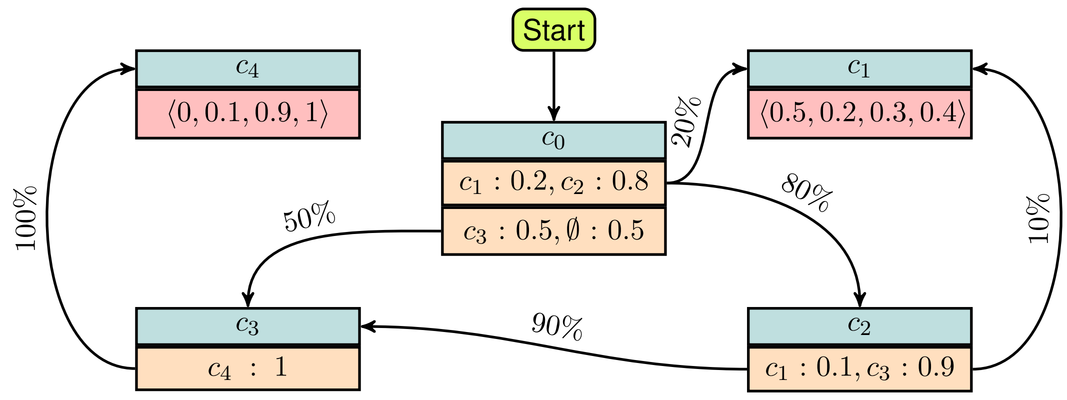


Figure 4.7: Probabilistic topology model

Our simulator consists of a stack automaton which takes as input a probabilistic model of the system (such as, for instance, the one depicted in Figure 4.7) and, through a Monte Carlo process, generates spectra. The probabilistic model describes the system's topology, the interaction between components, and the system's faults. To build the topology portion of the model, two primitives exist: components and links (depicted in blue and orange colors, respectively). Concretely, a component (identified by its numeric ID) contains a list of links. A link consists of a list of component IDs with associated transition probabilities ($\emptyset$ corresponds to no transition).

Whenever a component is activated, its links are sequentially evaluated. The evaluation of a link consists of pushing the component's next link onto the stack (if it exists) and randomly selecting a component (based on the transition probabilities defined in the activated link) to continue the execution. When all the links belonging to a component have been evaluated, a link is popped from the call stack, returning the control to the caller component. As soon as all the links belonging to a component have been evaluated and the stack is empty, the execution halts.

Considering the description of the simulator, a transaction can be generated by pushing the first link of a component marked as an entry point onto the stack (c_0 in Figure 4.7) and running the simulator from that state until its execution comes to a stop. The spectrum is generated by repeating this process an arbitrary number of times while recording which components have been activated in each transaction.

To emulate the error behavior, components may be injected with faults (depicted in red), which are parameterized over 4 variables (p_c, p_d, p_i, and p_f). p_c, p_d, and p_i correspond to the probabilities of correct, degraded, and incorrect behavior. During the simulation, whenever a faulty component is activated, the outcome of such activation (in terms of correct, degraded, or incorrect) is randomly determined using such probabilities. In the event of a component performing erroneously, it has an associated probability p_f of failure which, whenever it occurs, it results in an premature end of the transaction (in Figure 4.7, an error in component c_4 always results in a failure whereas an error in component c_1 only has a 40% chance of resulting in a failure). To determine the transaction's fuzzy error value, we apply the following rules:

$$e = \begin{cases} 1, & \text{if at least one component performed erroneously} \\ 0, & \text{if all components performed correctly} \\ \boldsymbol{rand}(0,1), & \text{otherwise}^{4} \end{cases} \tag{4.7}$$

To generate the spectra required for our benchmark we undergo a two-step process. In the first stage we randomly generate a set of system models, while in the second, we use such models to generate the required spectra.

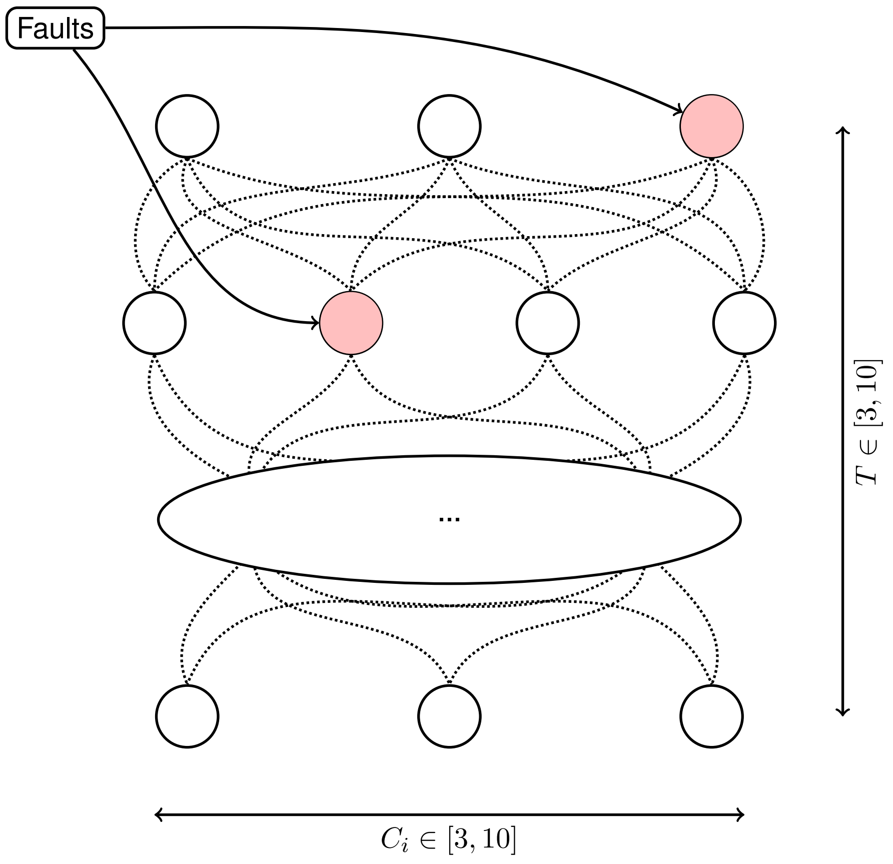


Figure 4.8: N-tier service architecture

We generated system models that comply with a N-tier service architecture (Figure 4.8). Systems were created by randomly selecting the number of tiers ($T \in [3,10]$) as well as the number of components in each tier ($C_i \in [3,10], 1 \leq i \leq T$). Every component is connected to all the components of the next tier with random transition probabilities. To exhibit erroneous behavior, a number of faults (F) was randomly injected (in terms of position) in the systems.

In our setup, we generated 100 systems for each value $F \in [2,4]$, totaling 300 systems. The injected faults had 90% and 10% probabilities of degraded, and erroneous behavior, respectively.

The spectra generation is parameterized over a single variable E, representing the number of errors at the end of which the simulation stops. For each generated system, we ran 10 simulations for each value $E \in [1, 9]$, totaling 90 spectra per system. Overall, our benchmark is composed of $300 \times 90 = 27000$ test cases.

4.2.2 Evaluation Metric

The wasted effort metric evaluates, for a particular diagnostic report, how many healthy components need to be inspected before all faulty components are found [Steimann et al., 2013]. To calculate this metric one must undergo an iterative process. Starting with the first candidate, all the candidate's components are inspected to determine whether that particular component was responsible for the erroneous behavior. Depending on the result of such inspection two outcomes may occur. On the one hand, if the component is found to be faulty, that particular component is removed from all other candidates in the ranking. On the other hand, if the component is found to be healthy, all candidates in the ranking containing that particular component are removed. This process is repeated until all faulty components are found. In the case of the last inspected candidate being tied with other candidates, it is assumed that, on average, half of the healthy components are examined.

During this iterative process, we keep track of two counters: inspected components (I) and faulty components (C). Using these two counters, the wasted effort metric is calculated as

$$W = I - C \tag{4.8}$$

	d	Rank	I	C
1	$\{c_1, c_4\}$	1	2	1
2	~~$\{c_2, c_3, c_4\}$~~	2		
3	~~$\{c_3, c_4, c_5\}$~~	3		
4	$\{$~~c_1~~$, c_2\}$	4	4	2
5	$\{c_3, c_5\}$	4		

Figure 4.9: Example diagnostic report

As an example consider the diagnostic report presented in Figure 4.9 for which the correct diagnostic candidate is $d = \{c_1, c_2\}$. In order to calculate the wasted effort, we start by examining c_1 and c_4 finding that c_1 is faulty while c_4 is healthy. Due to c_4 being healthy, candidates d_2 and d_3 are discarded. Examining d_4 we observe that the only unexplored component (c_2) is faulty. Additionally, we see that both system's faults were discovered. However, as d_5 is tied with d_4, we must inspect half of the healthy components. The wasted effort of this diagnosis is therefore $W = 4 - 2 = 2$, meaning that 2 healthy components (c_4 and c_3/c_5) were examined in the process of finding the root cause of the system's errors.

A normalized version of the wasted effort is called *diagnostic quality* and is defined as:

$$Q = 1 - \frac{W}{M - C} \tag{4.9}$$

where M is the number of system components. The diagnostic quality value is between zero and one and estimates the fraction of system's healthy components that need to be examined before all faulty components are found.

We refine the diagnostic quality metric to take into account the fact that, for a specific spectrum, not all components of the systems can be at fault. As an example consider a system with 1000 components with a spectrum consisting of a single failing transaction activating 2 components. Assuming the diagnostic algorithm only proposes plausible[5] candidates, the quality is contained in the interval between 1 and $\frac{999}{1000}$. Instead of calculating the diagnostic quality using the M components of the system, we use M_s, the number of "suspicious" components to calculate the new metric, which shall be referred to as "fair quality" (Q_f). A component is said to be suspicious if it was activated in a failing transaction. A consequence of using Q_f is that the diagnostic qualities of all possible permutations of the ranking always have a lower bound quality of 0.

4.2.3 Results

In this section, we compare the performance of the crisp diagnostic approach (see Section 1.2.2, page 17) with our fuzzy approach for the generated spectra.

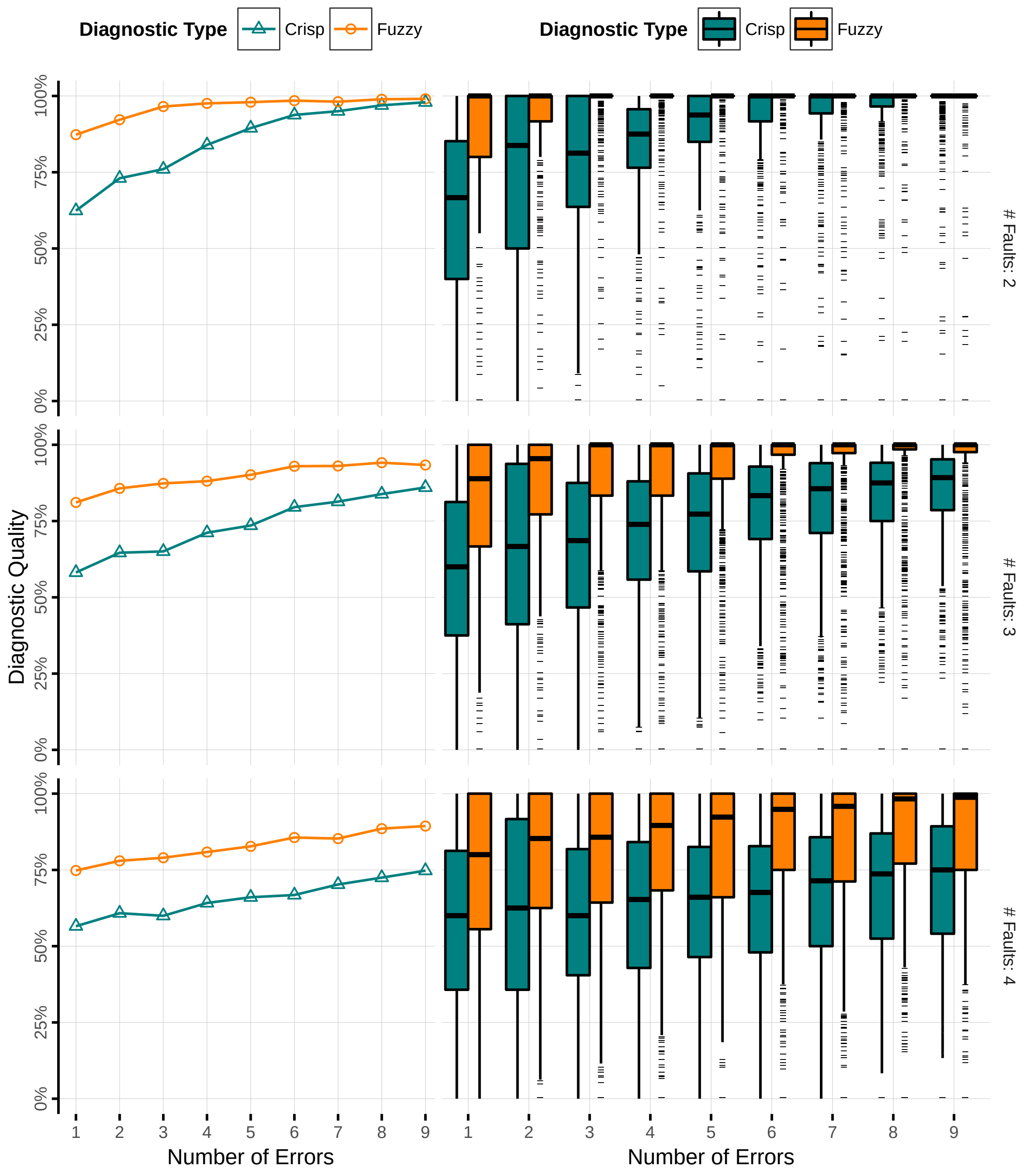
In Figure 4.10a, we compare the average Q_f for each test scenario. From the analysis of the plot we can see that the crisp approach was, on average, outperformed by the fuzzy approach. This is due to the fact that the fuzzy approach is able to successfully take advantage of the extra fuzzy error information to break the ties in the ranking (as shown in the example from Figure 4.4) that occur when dealing with small numbers of erroneous transactions. Furthermore, with the increase of erroneous transactions, it appears that the average crisp approach's Q_f seems to converge towards to the same average Q_f as the fuzzy approach. This happens due to the fact that the information introduced by the occurrence of errors eventually compensates for the limitations imposed by the crisp error abstraction.

In Figure 4.10b, we present a set of box plots[6] comparing the diagnostic quality distributions of both approaches for each test scenario. From the analysis of the plots we can see that the fuzzy approach not only has a better performance than the crisp approach, but also that the fuzzy approach distribution is more skewed towards better quality results than the crisp approach. Additionally, we can see that the fuzzy approach exhibits a higher consistency (*i.e.*, smaller inter-quartile range) than the crisp approach. This tendency can be further observed in Figure 4.11, where we plot a more detailed version of Figure 4.10b, in which we include every test case result.[7]

[5]By plausible we mean that all the candidate's components were at least activated once in an erroneous transaction.

[6]For each test scenario, the box corresponds to 2^{nd} and 3^{rd} quartiles (*i.e.*, 50% of the cases), the vertical lines correspond to the 1^{st} and 4^{th} quartiles, and the small dashes correspond to test cases categorized as outliers. A test case is considered to be an outlier if its distance from the box is greater that $1.5 * IQR$ (inter-quartile range, *i.e.*, the height of the box).

[7]To improve the visualization, we added transparency and random horizontal displacement to each data point.



(a) Averages **(b)** Box plots

Figure 4.10: Benchmark results

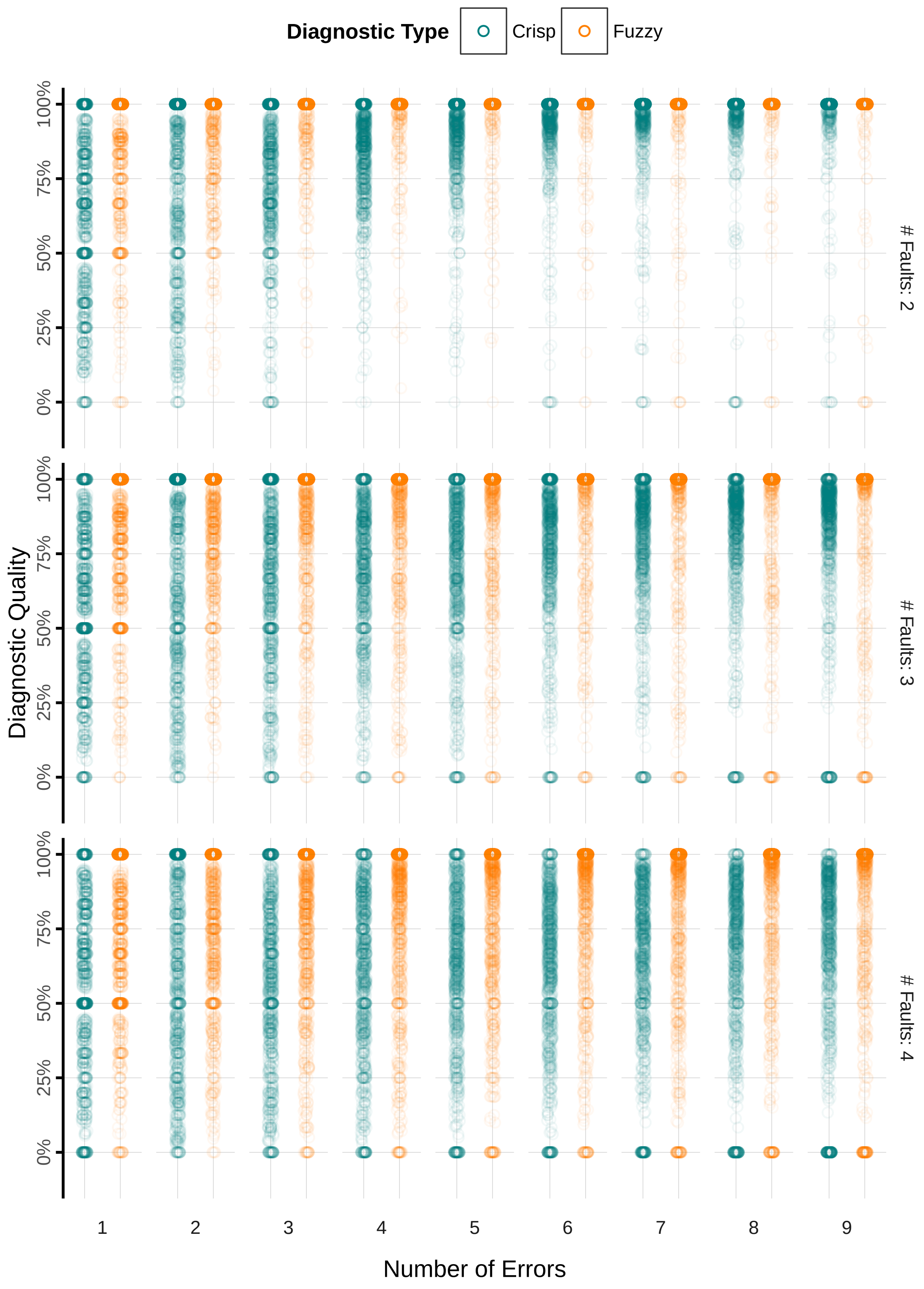


Figure 4.11: Benchmark results (detailed view)

A pairwise analysis of the data (Figure 4.12) shows that our approach outperformed the crisp approach in 65% of the test cases. Moreover, in 94% of the cases our approach was at least as accurate as the classical approach. In the remaining 6% of the test cases the accuracy loss was due to (1) lack of observations, and (2) marginal variations in the posterior probability but large enough to make the relative ranking change. The overall average improvement of quality introduced by our algorithm was of $\Delta Q_f = 0.153$, representing a relative improvement of 21%. By performing a paired one-tailed T-test, we can ascertain that our approach introduced a relative improvement of 20%, with a 99% confidence interval.

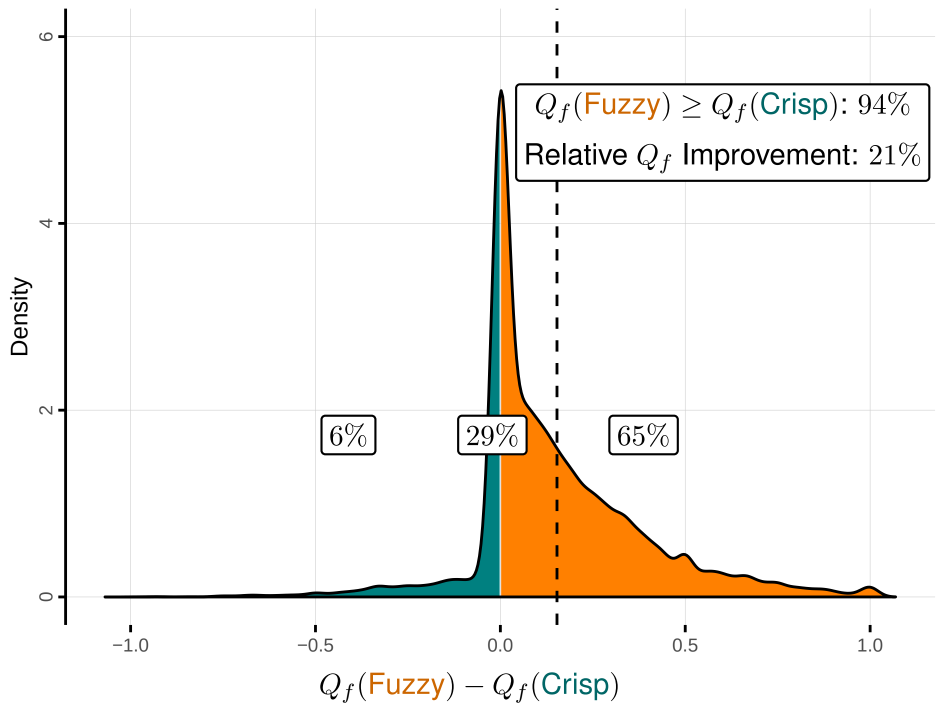


Figure 4.12: Quality improvement density plot

4.3 Summary

In this chapter we successfully addressed the limitations presented in Section 1.3.2.1 (page 21) thereby answering Research Questions 3 and 4 (page 22). Concretely, in this chapter:

- We addressed Research Question 3 by using fuzzy logic instead of the classical binary logic to detect/encode error states (Section 4.1.1, page 57).
- We addressed Research Question 4 by generalizing the state-of-the-art *SFL* using the concept of probability of a fuzzy event (Section 4.1.2, page 60).
- We presented the conducted benchmarks showing that our fuzzy error approach (Section 4.2.3, page 66):
  - Improved the diagnostic quality in 65% of the test cases.
  - Performed at least as good as the classical approach in 94% of the test cases.

- The average relative improvement introduced by our approach was of 20%, with a 99% confidence interval.

```
-[--->+<]>-------.--------.+.--.<>-++--<+<>-><->+-->>+->-+>++++++><-+<
```

5 NFGE

In this chapter we focus on addressing the *SFL* limitations explained in Section 1.3.2.2 (page 23). Concretely, we present our approach, dubbed **Non-linear Feedback-based Goodness Estimate (*NFGE*)**, aimed at modeling the components' goodness as a non-linear function (referred to as $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$) of a set of observable state variables. Additionally, we discuss how the *SFL* diagnostic framework can be generalized to make use of state-based goodness models, such as the one we propose in this chapter.

5.1 Approach

Our approach can be divided into two independent stages: the modeling and diagnostic stages.

5.1.1 Modeling $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$

Let an abstract data type, henceforward referred to as feedback spectrum, be the input of our modeling approach.

Definition 10 (Feedback Spectrum)**.** *The feedback spectrum encodes the result of a set of diagnoses. Concretely, let M denote the cardinality of $COMPS$. Fb_{ej} consists of a $2 \times M$ matrix defined as:*

$$Fb_{ej} = \langle st_1, ..., st_k, ..., st_K \rangle \tag{5.1}$$

Fb_{0j} and Fb_{1j} are the pass and fail feedback observation lists for component c_j, respectively. Each element of Fb_{ej} encodes the value of a set of state variables for component c_j.

In the following we assume that a mechanism for collecting feedback spectra exists.

Our approach to model $\breve{g}_j(st)$ consists of a probabilistic model that is derived from the feedback spectrum. Concretely, we estimate the pass and fail probability density functions[1], from which we trivially calculate $\breve{g}_j(st)$.

The estimation of the pass/fail probability densities consist of a **Kernel Density Estimate (*KDE*)**, $\hat{f}_{ej}(st)$, defined as:

$$\hat{f}_{ej}(st) = \frac{1}{bw} \sum_{st' \in Fb_{ej}} \boldsymbol{K}\left(\frac{st - st'}{bw}\right) \tag{5.2}$$

where $bw > 0$ is the bandwidth, a smoothing parameter, and $\boldsymbol{K}(\cdot)$ is a kernel function[2] [Rosenblatt, 1956, Parzen, 1962]. A key aspect of *KDE* is the selection of the bandwidth parameter bw. In our approach, we estimate bw by using the Silverman's "rule of thumb" [Silverman, 1986], defined as:

$$bw = 0.9 \times \boldsymbol{min}\left(\sigma, \frac{R}{1.34}\right) \times |Fb_{ej}|^{(-0.2)} \tag{5.3}$$

where R is the inter-quartile range of Fb_{ej}. Regarding $\boldsymbol{K}(\cdot)$, even though several options exist, in our approach, we use the Gaussian kernel. Additionally, and without loss of generality, we assume the *KDEs* are a function of a single variable.

As an example, consider the modeling process of $\breve{g}_j(st)$ for a component c_j given 5 pass and 3 failed feedback observations with values $Fb_{0j} = \{5, 7, 15, 20, 40\}$ and $Fb_{1j} = \{40, 44, 60\}$, respectively.

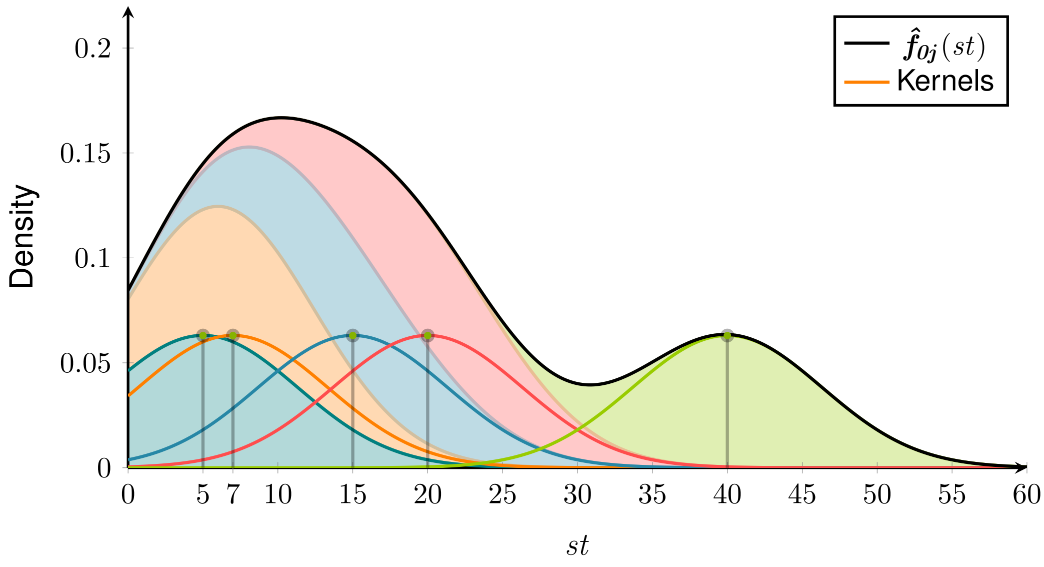


Figure 5.1: Density estimation and underlying kernels

[1]The variables are arbitrary and must be selected on component-to-component basis. Currently, the set of variables must be manually selected.
[2]A kernel is a symmetric but not necessarily positive function that integrates to one.

The first step in modeling $\breve{g}_j(st)$ is the estimation of the probability density function (Equation (5.2)) for the nominal executions with parameters $Fb_{0j} = \{5, 7, 15, 20, 40\}$ and $bw = 6.328$ (Figure 5.1). Note that for each value in the horizontal axis, the *KDE* value corresponds to the summation of all underlying kernels at the same point. From Equation (5.2) we can see that $st' \in Fb_{ej}$ determines each kernel's offset and bw the dispersion of the density. In particular, when using the Gaussian kernel, st' corresponds to its mean and bw to its standard deviation.

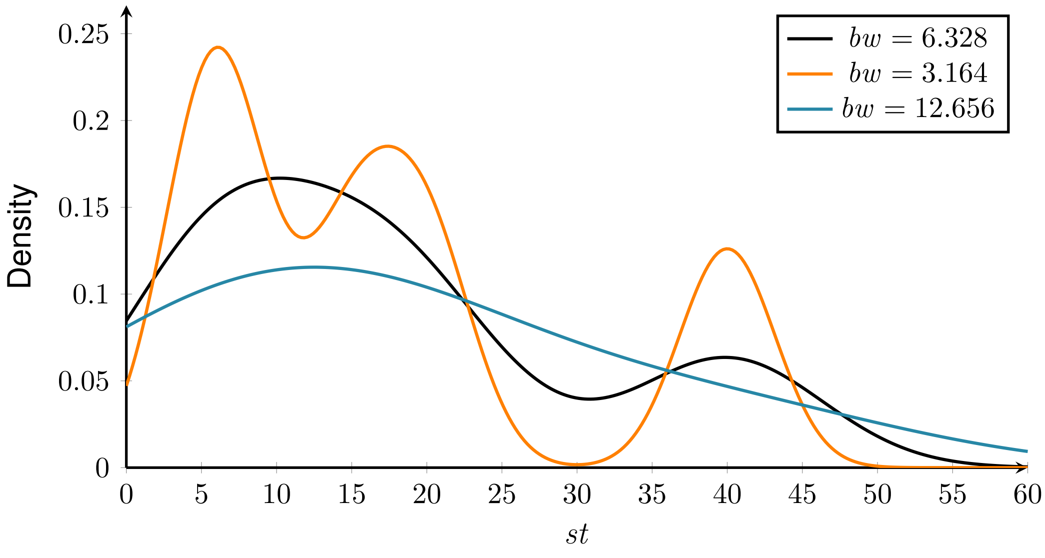


Figure 5.2: Impact of bw value

Figure 5.2 provides a visual intuition on the effect of the parameter bw in the estimate. A sensible selection of bw is crucial in order to yield good results as using a small bw value will reflect sampling artifacts whereas a large bw value will smooth some behavioral trends.

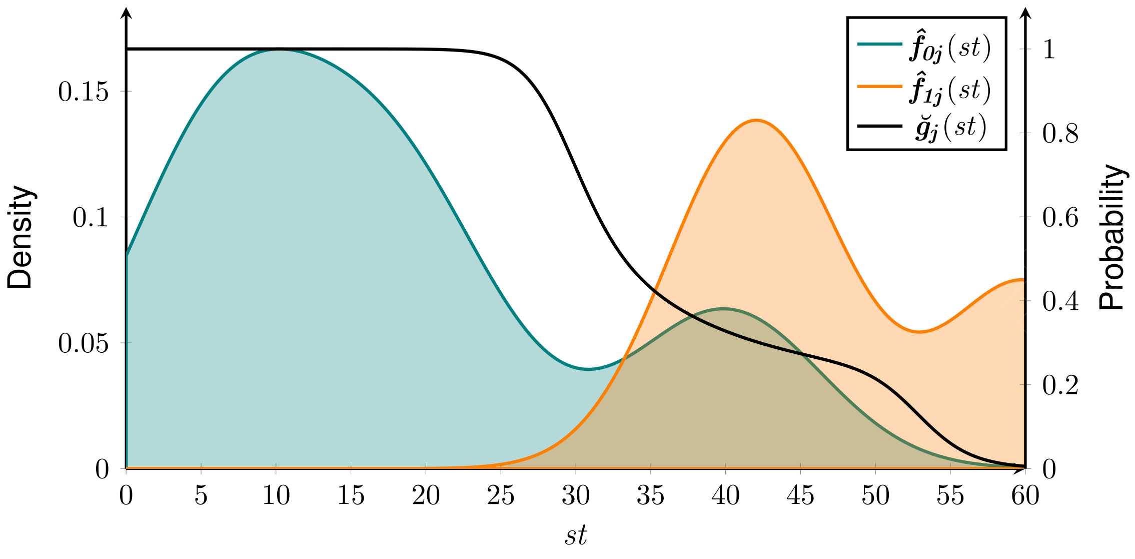


Figure 5.3: Goodness *vs.* pass/fail *KDE*s

The second and final step is the derivation of $\breve{g}_j(st)$ from the pass/fail *KDE*s. We will assume that the previous step was repeated for the fail executions yielding the densities depicted in Figure 5.3. $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$, is defined as:

$$\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) = \frac{\hat{\boldsymbol{f}}_{\boldsymbol{0j}}(st)}{\hat{\boldsymbol{f}}_{\boldsymbol{0j}}(st) + \hat{\boldsymbol{f}}_{\boldsymbol{1j}}(st)} \tag{5.4}$$

5.1.2 Ranking using $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$

To use the $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ models in the *SFL* framework we make use of a data-structure which we shall refer to as state spectrum.

Definition 11 (State spectrum)**.** *State spectrum is a generalization of the hit spectrum data structure (see Definition 7, page 10) that is able to encode the state of a set of variables for each execution of $c_j \in COMPS$. Formally, we redefine A as:*

$$A_{ij} = \begin{cases} \clubsuit, & \textit{if } c_j \textit{ does not have a } \breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) \textit{ model and was involved in transaction } i \\ \langle st_1, \cdots, st_k \rangle, & \textit{if } c_j \textit{ has a } \breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) \textit{ model and was involved in transaction } i \\ \emptyset, & \textit{otherwise} \end{cases} \tag{5.5}$$

Each element st_k, encodes the state of the observed variables for the k^{th} activation of component j in transaction i.

From the definition, it follows that if no component has a $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ model, only $\clubsuit$ and $\emptyset$ will appear in the activity matrix. In this scenario, $\clubsuit$ corresponds to a 1 and $\emptyset$ to a 0 in the terms of Definition 7.

Furthermore, since the state spectrum may encode an arbitrary set of state variables, it generalizes over a range of alternative types of spectra described in the literature (*e.g.*, count spectrum, time spectrum, path spectrum, *etc.*).

To apply $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ to the *SFL* framework we generalize $\boldsymbol{G}(d, A_i)$ as:

$$\boldsymbol{G}(d, A_i) = \prod_{j \in (d \cap A_i)} \begin{cases} g_j, & \text{if} A_{ij} = \clubsuit \\ \breve{\boldsymbol{G}}(j, A_{ij}), & \text{otherwise} \end{cases} \tag{5.6}$$

$$\breve{\boldsymbol{G}}(j, S) = \prod_{st \in S} \breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) \tag{5.7}$$

In contrast to the former version of $\boldsymbol{G}(d, A_i)$, our generalization does take into account all the states in which c_j was active.

In the case of existing components with no $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ models, the *MLE* procedure still needs to be executed. To perform the *MLE*, all $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ must be evaluated such that $\boldsymbol{G}(d, A_i)$ is reduced to the form:

$$\boldsymbol{G}(d, A_i) = P \times \prod_{j \in A_i'} g_j \tag{5.8}$$

$$A'_i = \{j \mid j \in d \wedge A_{ij} = \clubsuit\} \tag{5.9}$$

where P is the result of the product of all $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ for transaction i.

The *MLE* as the effect of maximizing the $\boldsymbol{Pr}(d \mid A, e)$ function by fitting a set of parameters (in this case g_j). In other words, by using the *MLE*, the constant g_j models are fitted to the input data (*i.e.*, (A, e)) and thus the diagnostic is more resilient to novel errors than when using $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ models. To address this issue, we further improve $\boldsymbol{G}(d, A_i)$ as:

$$\boldsymbol{G}(d, A_i) = \prod_{j \in (d \cap A_i)} \begin{cases} g_j, & \text{if} A_{ij} = \clubsuit \\ (1 - \alpha_j) \cdot g_j + \alpha_j \cdot \breve{\boldsymbol{G}}(j, A_{ij}), & \text{otherwise} \end{cases} \tag{5.10}$$

where $\alpha_j \in [0, 1]$ is the estimator confidence factor for component j. In scenarios in which the $\breve{\boldsymbol{G}}(j, S)$ is fully trusted, α_j should be set to 1. Alternatively, it is possible to fall back to the original *SFL* algorithm by setting every α_j to 0.

As an example, consider that a $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ model is known to become gradually obsolete over time. In this scenario, it should be possible to fit, for instance, a sigmoid function to the obsolescence pattern, as shown in Figure 5.4. We can see that, as time passes, the impact of $\breve{\boldsymbol{G}}(j, S)$ in the diagnosis gradually decreases while the impact of g_j increases.

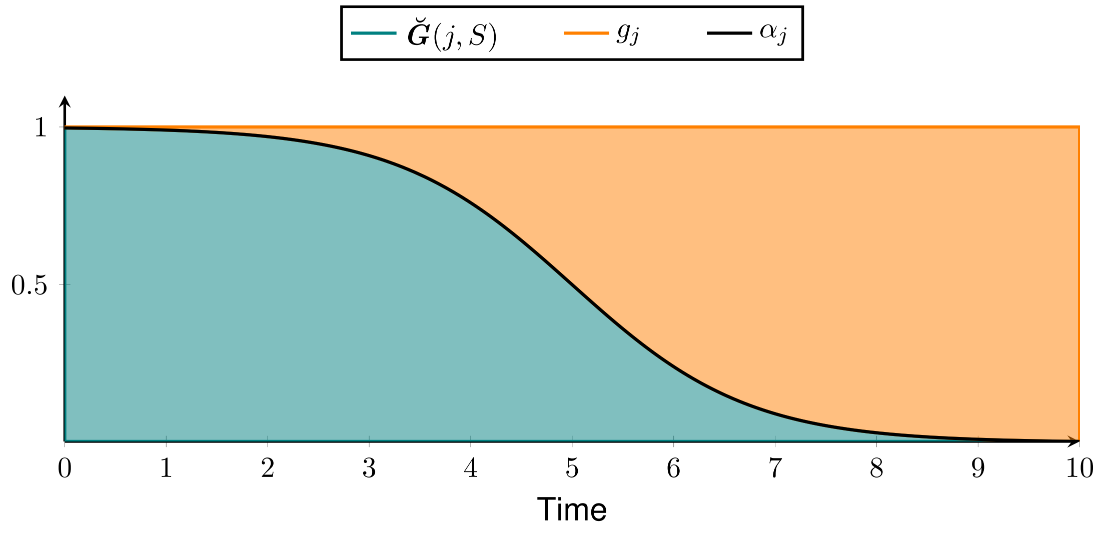


Figure 5.4: Estimation model confidence *vs.* time

To compare *NFGE* with the approach presented in Section 1.2.2 (page 17), consider the state spectrum presented in Figure 5.5a, for which the only minimal candidates are $d_1 = \{1\}$ and $d_2 = \{2\}$. Furthermore, consider that the $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ model for both c_1 and c_2 was defined as shown in Figure 5.5b.

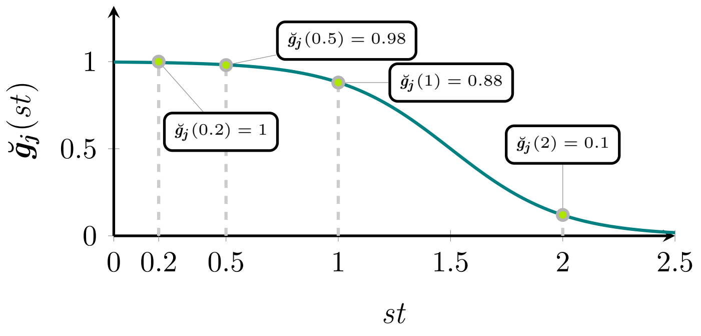
By applying *NFGE* with $\alpha_1 = \alpha_2 = 0$ (*i.e.*, the $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ models are not used in the diagnostic process), it follows that:

$$\boldsymbol{Pr}(A, e \mid d_1) = \underbrace{(1 - g_1)}_{t_1} \times \underbrace{g_1}_{t_2} \tag{5.11}$$

$$\boldsymbol{Pr}(A, e \mid d_2) = \underbrace{(1 - g_2)}_{t_1} \times \underbrace{g_2}_{t_3} \tag{5.12}$$

i	A		e
	c_1	c_2	
1	$\langle 2 \rangle$	$\langle 0.5 \rangle$	1
2	$\langle 0.2 \rangle$	$\emptyset$	0
3	$\emptyset$	$\langle 1 \rangle$	0

(a) State spectrum



(b) Goodness Function

Figure 5.5: *NFGE* example

We can see that, after applying the *MLE* procedure, d_1 and d_2 are ranked equally.

In contrast by applying *NFGE* with $\alpha_1 = \alpha_2 = 1$ (*i.e.*, the *MLE* procedure is not used in the diagnostic process), it follows that:

$$\begin{aligned} \boldsymbol{Pr}(A, e \mid d_1) &= \underbrace{\big(1 - \boldsymbol{\breve{g}_1}(2)\big)}_{t_1} \times \underbrace{\boldsymbol{\breve{g}_1}(0.2)}_{t_2} \\ &= \underbrace{(1 - 0.1)}_{t_1} \times \underbrace{1}_{t_2} \\ &= \underbrace{0.9}_{t_1} \times \underbrace{1}_{t_2} \\ &= 0.9 \end{aligned} \tag{5.13}$$

$$\begin{aligned} \boldsymbol{Pr}(A, e \mid d_2) &= \underbrace{\big(1 - \boldsymbol{\breve{g}_2}(0.5)\big)}_{t_1} \times \underbrace{\boldsymbol{\breve{g}_2}(1)}_{t_3} \\ &= \underbrace{(1 - 0.98)}_{t_1} \times \underbrace{0.88}_{t_3} \\ &= \underbrace{0.02}_{t_1} \times \underbrace{0.88}_{t_3} \\ &= 0.0176 \end{aligned} \tag{5.14}$$

We can see that, for this particular example, *NFGE* succeeds at differentiating d_1 from d_2.

5.2 Benchmark

To assess the performance of *NFGE* we conducted two studies. The first study is aimed at evaluating the prediction error of the modeling approach. At this stage, we use synthetic goodness models in order to be able to test a wider set of goodness patterns. After establishing the prediction error of *NFGE*, the second study aims at exploring, in a real application, the cases where the classical approach tends to fail.

5.2.1 Prediction Error Study

To assess the prediction error of our approach we generated a set of 20000 random synthetic goodness models (M). With the purpose of having different learning and observation generation processes, we modified Equation (5.2) such that each underlying kernel has an individual bw_i. Formally, the synthetic goodness models have the underlying pass and fail distributions defined as:

$$\hat{\boldsymbol{f}}_{\boldsymbol{ej}}(st) = \sum_{i=1}^{|Fb_{ej}|} \frac{\boldsymbol{K}(\frac{st - Fb_{eji}}{bw_i})}{bw_i} \tag{5.15}$$

Additionally, in our synthetic data setup, two types of models can be distinguished. The first set of models use the Gaussian kernel as their building block. These models are intended to mimic the behavior of components that exhibit smooth transitions between any two points in the feature space (*i.e.*, the domain of the observable variables). This can be the case of component aging in which the goodness normally decreases gradually over time (*e.g.*, Figure 5.5b).

The second type of model uses the Box kernel, *i.e.*, a rectangular-shaped kernel centered at Fb_{eji} with bw_i width and $\frac{1}{bw_i}$ height. This set of models is intended to emulate components that exhibit abrupt transitions in their goodness functions. Also, as the original kernel differs from the learning kernel, the process of generation and learning becomes substantially different, allowing us to get to more significant conclusions.

Finally, the generated models range from simple patterns, such as for instance the one depicted in Figure 5.5b, to more complex patterns with up to 20 supporting kernels for both the pass and fail densities (*i.e.*, $|Fb_{ej}| \leq 20$).

To generate the feedback spectra, for each model we randomly selected a set of 200 values, F, in its feature space. For each of them we determine whether the component performed nominally in a Bernoulli trial process parameterized parametrized with $\boldsymbol{M_l}(F_k)$. For each $\boldsymbol{M_l} \in M$, we trained several estimators with different training data sizes.

For each trained model, the prediction error is obtained by comparing the predicted goodness and the original goodness at 1000 evenly distributed samples of the feature space and averaging the differences. Figure 5.6 summarizes the results our benchmarks. The first two setups represent the average prediction error for *NFGE* in both the Gaussian and Box cases, respectively. The last setup represents the average prediction error for a constant estimator defined as:

$$\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) = \frac{|Fb_{0j}|}{|Fb_{0j}| + |Fb_{1j}|} \tag{5.16}$$

As discussed, the approach presented in Section 1.2.2 (page 17) is not able to incorporate feedback information, rendering a direct comparison with *NFGE* impossible. Despite, we provide the results for the constant estimator with the goal of establishing a ground of comparison with the hypothetical performance of the traditional *SFL* approach if it was able to incorporate such information.

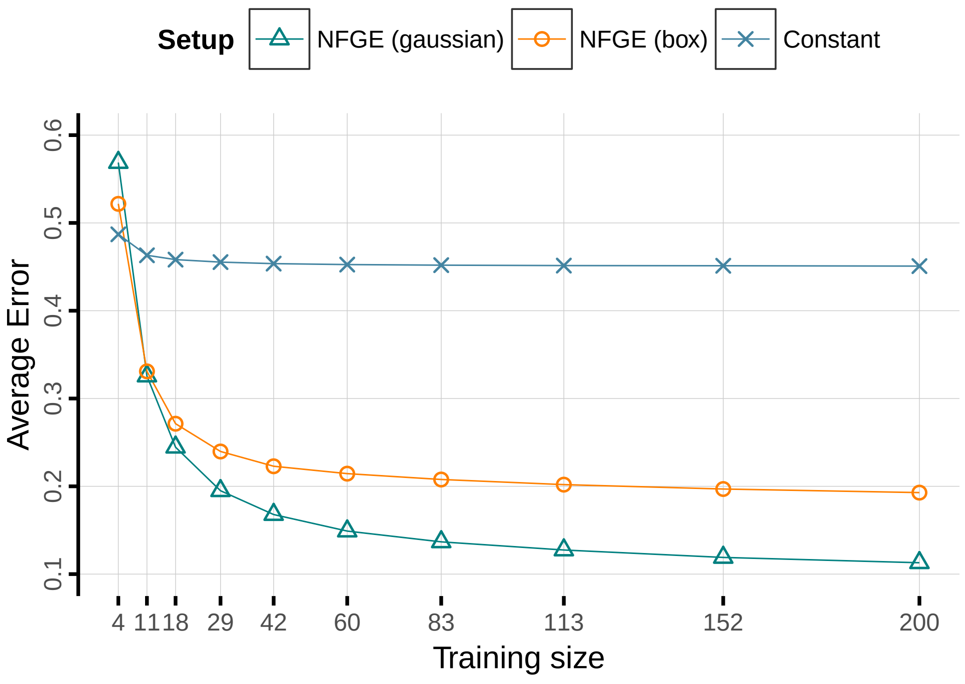


Figure 5.6: Prediction errors

From the analysis of Figure 5.6, we can see a clear improvement introduced by *NFGE* over the baseline. As expected, the constant estimation presents a large average error of 44%. Moreover, the constant estimator does not scale with the amount of available feedback data (*i.e.*, the average error does not significantly improve with the increase of the available data).

In contrast, *NFGE*, in its best case scenario, was able to perform at 10% of average error. Globally, *NFGE* presented an average of 21% and 26% of prediction error for the smooth and non-smooth cases, respectively. The reason for the 5% difference relates to the fact that the Gaussian-based *KDE* is not able, with a limited set of observations, to tightly fit the original box-based model.

Additionally, fairly quick convergence is observed: with 42 observations 90% of the maximum performance is obtained for both the Gaussian model/Gaussian estimator case and the Box model/Gaussian estimator case.

Finally, the results showed that when the amount of available data is small (< 11 observations), the constant estimate, on average, outperforms *NFGE*.

5.2.2 Diagnostic Study

In this section we evaluate the diagnostic performance of our approach in scenarios where the classical *SFL* approach tends to fail. The selected application for this study was a simple *HTTP* server, *webfs*[3], which was instrumented to generate state spectra. Additionally, we injected code to emulate the behavior of aging faults (*e.g.*, memory leaks, hard drive wear, *etc.*). The injected faults are parameterized over 3 variables: min, max, and $total$. For each execution of each injected fault a counter is incremented with a random value in the interval $[min, max]$. Whenever a counter reaches the value of its associated $total$ variable, the fault is triggered and the transaction fails. The counters may either be shared among a set of faults or, unless stated otherwise, independent.

To collect the feedback spectra required for generating the goodness models, we did a prior test run where the faults were targeted individually. For each injected fault execution, we recorded the number of previous invocations and process age, *i.e.*, the time since the *HTTP* server start. For each fault we created two univariate $\breve{g}_j(st)$ models using 20 feedback observations: one as a function of the number of invocations and the other as a function of the process age. Since we only created *NFGE* models for the components with injected faults, no state was recorded for the remaining components.

In a first scenario we only activated one fault. At this stage we tried to isolate the faulty component from a 5 element diagnostic report. The results showed that the *MLE* approach was only able to exonerate 1 out of 5 candidates. This was due to the fact that all transactions shared the same "bootstrap" sequence and the fault was injected in there. Since the components involved in that sequence exhibited the same activity pattern, *i.e.*, equal columns in the hit spectrum, it remained impossible to distinguish them. In contrast, *NFGE* was able to clearly isolate the real faulty component. When comparing the candidates' probabilities for both approaches, *NFGE* calculated a probability of 99.9999% for the actual diagnostic candidate whereas *MLE* calculated a probability $\approx 25\%$ for both the correct and the 3 remaining diagnostic candidates. This great difference in the probabilities' magnitudes is in accordance with the results obtained in the synthetic experiments.

In a second scenario, we enabled two faulty components for which we had created *NFGE* models. In this setup we generated a set of transactions that would eventually trigger one fault but not the other. Additionally, the two components were always activated in succession, generating low entropy hit spectrum (*i.e.*, several components share the same activation pattern). The goal of this test was to confirm that *NFGE* is both able to both indict and exonerate components depending on the execution context.

In this scenario, the *MLE* ranked equally the real error source and the component that should be exonerated. In contrast, *NFGE* was again able to both indict and exonerate components correctly.

Even though the previous setups produced positive results, it is important to note that the correct selection of the modeling variables is critical to obtain a good diagnostic accuracy. As previously stated we observed two distinct variables: the number of component executions and the process age. Even though both variables are able to model the goodness pattern of the injected faults, the degree to which each variable is able to cope with new scenarios may vary.

[3]`http://linux.bytesex.org/misc/webfs.html`

From this test setup it is easy to understand that even though correlated, the process age is not the cause of the fault activation: if a component is "old" but was never activated, the corresponding counter was never incremented. The process age variable indirectly encodes some aspects of the application workload. If an age-based model was constructed for a specific activation rate, *i.e.*, activations/time unit, its ability to produce accurate estimates under a different workload may be limited. On the other hand it is clear that the component activation count is more independent from the application's workload.

In the previous setups we used independent counters. If on the other hand we had a global counter and still used independent activation counters, the same variable would also implicitly encode workload patterns. Furthermore, if the modeling workload is substantially different from the diagnostic workload (and $\forall_{\alpha_j} : \alpha_j = 1$) it could happen that the components that caused the errors would be exonerated.

5.3 Summary

In this chapter we successfully addressed the limitations presented in Section 1.3.2.2 (page 23) thereby answering Research Questions 5 and 6 (page 23). Concretely, in this chapter:

- We addressed Research Question 5 by proposing a *KDE*-based approach that uses feedback observations to model the components' goodnesses as a non-linear function of the system's state (Section 5.1.1, page 71).
- We addressed Research Question 6 by generalizing both the hit spectrum abstraction and the state-of-the-art *SFL* approach to use information about the system's state in the diagnostic process (Section 5.1.2, page 74).
- We presented the conducted benchmarks showing that:
  - The *NFGE* modeling approach, in its best case scenario, was able to perform at 10% of average error (Section 5.2.1, page 77).
  - With 42 data points, the *NFGE* approach achieved 90% of its maximum observed performance (Section 5.2.1, page 77).
  - The system's state was successfully used to improve the diagnostic accuracy (Section 5.2.2, page 78).

```
-[--->+<]>---.+[----->+++<]>.+++++++.-----------.--[--->+<]>-.+++[->++
+<]>.-.-[--->+<]>-.---[->+++<]>.-[--->+<]>---.+++.-------.-+>+<+><+-+-
```

6 Related Work

6.1 Traditional Debugging

Data Dumping

A common low-tech diagnostic approach is the use of commands that dump user defined variables to output devices, colloquially referred to as *printf* commands. On the one hand, such commands are a very powerful tool to both detect and locate errors. First, the usage of such commands is normally pretty straightforward implying a small learning overhead. Also, as the semantics are equal to the ones of the target source code, its usage is intuitive. Second, as such commands are embedded in the source code, there is no need for additional tools. Third, as no assumptions are made, it is usable in almost all imaginable scenarios, being the *de facto* diagnosis fallback tool.

On the other hand, mostly due to the fact that the usage of dumping commands is very *ad hoc*, this technique can become ineffective. First, all dumping commands must be explicitly stated, cluttering the source code. Second, as all commands must be prepared prior to the application's execution, a missing print command implies a possible recompilation and re-execution of the application. Third, with the increase of debugging print commands the amount of debugging output can become quickly overwhelming. Forth, in order to efficiently use print commands it is required to have some intuition about the location of the fault.

Debuggers

In contrast to print commands, debuggers enabled developers to inspect the internal state of an application without any source code modifications.

The first debuggers, referred to as *post-mortem* debuggers, enabled developers to analyze the status of the registers and memory after the program's execution. Those tools had however a scalability issue: as the debugger did not have any information about the source code variables, understanding the information made available by the debugger was not a trivial task. In order to tackle that problem, symbolic debuggers were developed, having the capacity of using data generated by the compiler to map memory addresses onto actual variable names.

Post-mortem debuggers were later enhanced with custom breakpointing features. With custom breakpointing, developers were endowed with the ability of pausing the program execution at arbitrary locations, analyzing its status and finally resuming its execution. The final breakpoint related improvement was the introduction of conditional breakpoints. This feature enables developers to pause the program's execution based on arbitrary rules. A well known example of a debugger is *GNU*'s *GDB* [Stallman et al., 2002].

6.2 Software Testing

Assertions/Oracles

Assertions/oracles are mechanisms that check for invariant violations at run-time [Rosenblum, 1995]. Whenever an invariant is broken, the application's execution is halted. This behavior, on the one hand, prevents an error state from propagating to other components and, on the other hand, reduces the scope of search for errors. In contrast to dumping commands, with the increase in the number of assertions, the diagnosis of errors becomes easier. Despite implying a performance penalty in the target application, assertions can normally be disabled and thereby not included in release versions of the applications.

Assertions can be broadly divided in two groups: *pre-conditions* and *post-conditions*. Pre-conditions check for the validity of the inputs to a particular component. Post-conditions check whether or not the component performed as expected (*i.e.*, the validity of the output). While pre-conditions are often easy to define, its normally hard to define a "good" validity test for the output. We quoted good due to the fact that the meaning of "good" is context dependent. In fact, the quality of an assertion is normally a trade-off between correctness and complexity.

As an example consider the "quicksort" sorting algorithm ($\mathcal{O}\big(n \cdot log(n)\big)$ complexity). A simple postcondition would be that the size of the output matches the size of the input ($\mathcal{O}\big(1\big)$ complexity). This condition is able to detect outputs with wrong sizes, but fails to detect unsorted outputs. A more robust postcondition would be to check that $\forall_{A_n} : A_n >= A_{n-1}$ ($\mathcal{O}\big(n\big)$ complexity). This condition is able to detect unsorted outputs, but fails to detect outputs that have elements that are not present in the input. An even more robust post-condition would be to compute the output with a different algorithm or implementation and compare it with the output under test ($\mathcal{O}\big(n \cdot log(n)\big)$).

Oracles can also be used to assure dependability at run-time by implementing **Triple Modular Redundancy (*TMR*)** [Lyons and Vanderkulk, 1962, Johnson and Wirthlin, 2010]. *TMR* is a form of *N-modular* redundancy in which three components perform the same task and the system's output is determined by a majority-voting process, producing a single output. A *TMR* system not only increases the likelihood of detecting errors but also opens the possibility of masking errors in the event of a minority of the redundant components performing erroneously.

In order to automate the process of creating assertions, approaches that automatically detect invariants (*e.g.*, range) in arbitrary system variables (known as fault screeners) were proposed [Abreu et al., 2008, Ernst et al., 2000, 2001, 2007, Hangal and Lam, 2002, Pinto et al., 2015, Racunas et al., 2007].

Test Suites

Test suites are collections of tests cases aimed automatically validating the functional requirements of systems [Huizinga and Kolawa, 2007]. Test suites are normally composed of unit tests, *i.e.*, smaller tests that are focused on testing a particular functionality of the system in isolation.

In order to ease the creation of unit tests, a vast set of unit testing frameworks for several languages have been proposed. Well-known unit testing frameworks include:

- *Check*[1] for *C*.
- *Boost*'s Unit Test Framework[2] for *C++*.
- *JUnit*[3] for *Java*.
- *unittest*[4] for *Python*.

While test suites are normally manually created, they are often the basis of a large number of automatic techniques. For instance, in the scope of software testing, regression testing [Vokolos and Frankl, 1998] enables developers to easily uncover eventual faults introduced by changes made to a base version of some software by automatically running an appropriate test suite on the new version of the software.

Even though regression testing contributes to a higher software quality, it may become cost ineffective due to several reasons. First, if the test suite has a low fidelity, *i.e.*, a low capacity for correctly detecting errors, the resources needed to use it may not justify the marginal reliability improvement. Second, if a software grows to a fair size, the time needed to test it may become impractical. Third, a test suite may easily contain test redundancy leading to resource waste.

[1] `http://check.sourceforge.net/`
[2] `http://www.boost.org/doc/libs/release/libs/test/`
[3] `http://junit.org/`
[4] `https://docs.python.org/3/library/unittest.html`

To mitigate the problem of low fidelity test suites, the concept of mutation analysis was introduced [Richard A. DeMillo,Richard J. Lipton, 1978]. Mutation analysis works by iteratively injecting the system with one artificial bug and then executing the test suite. If the injected fault goes undetected, it shows a case where the test suite potentially fails to detect an error and therefore needs improvement. On the one hand, mutation analysis methods evaluate how good a test suite is at uncovering faults and, on the other hand, give insight into how and where a test suite needs improvement. Mutation analysis is also used as a way of automatically generating test cases [Fraser and Arcuri, 2011, Fraser and Zeller, 2012, Fraser and Arcuri, 2013, Campos et al., 2013].

To reduce amount of testing needed, methods for prioritizing a test suite (test case prioritization) have been developed [Rothermel et al., 1999, 2001, Bryce and Memon, 2007, González-Sanchez et al., 2011b, Elbaum et al., 2002, Sun et al., 2013, Huang et al., 2010]. Such approaches determine an execution order for the test cases such that the first tests to be executed are the ones that have the best chance to uncover errors. With a prioritized test suite it is possible to more efficiently test a software.

To maintain a test suite over time it is necessary to remove redundancies/inconsistencies. A similar problem in terms of concept is the problem of selecting a subset of the test suite that satisfies a set of constraints while maximizing/minimizing a particular score (*e.g.*, code coverage and/or oracle cost). This problem is referred to as the test suite minimization problem. In order to tackle test case minimization problem, automatic solutions that use constraint solvers have been proposed [Arito et al., 2012, Campos and Abreu, 2013, Mondal et al., 2015]. It is important to note, however, that the reduction in the test suite size may have a negative impact in its fault detection capabilities [Rothermel et al., 1998].

6.3 Instrumentation

Code Instrumentation

In order to obtain information about a system during its execution, it needs to be injected with instrumentation code that captures the desired data [Garlan and Perry, 1995]. Instrumentation code normally relies on a series of *trampolines* to an arbitrary function `foo()` that deals with the specifics of the data collection (Figure 6.1).

Instrumentation is used in a wide diversity of domains. In addition to most of the techniques presented in this chapter and in this thesis, instrumentation is used in [Kumar et al., 2005]:

- Demand-driven structural software testing [Misurda et al., 2005].
- Dynamic optimizers [Arnold and Ryder, 2001, Hong et al., 2015, Bala et al., 2000].
- Modeling computer architecture features [Cmelik and Keppel, 1995, Scott et al., 2003, Witchel and Rosenblum, 1996].
- Software security [Kiriansky et al., 2002, Scott and Davidson, 2002, Song et al., 2008].

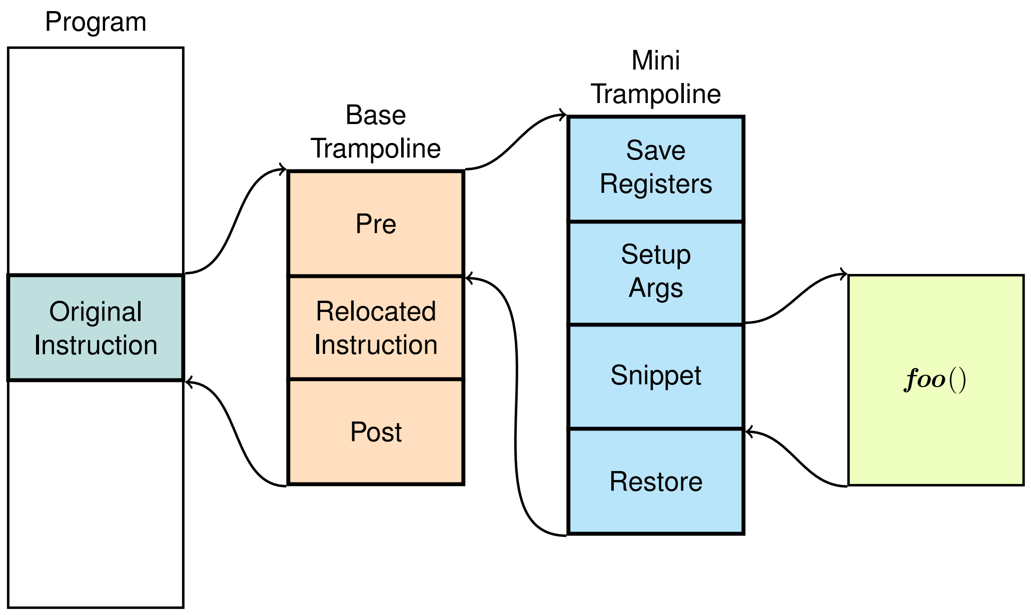


Figure 6.1: Instrumentation code insertion (adapted from [Tikir and Hollingsworth, 2002])

Several instrumentation techniques have been proposed. These techniques include:

- Manually placing instrumentation in a program before it executes.
- Static binary modification [Bus et al., 2004, Larus and Schnarr, 1995, Srivastava and Eustace, 1994].
- Dynamic/Live instrumentation in executing programs [Arnold and Ryder, 2001, Hollingsworth et al., 1997, Luk et al., 2005a,b, Lattner, 2002, Nethercote and Seward, 2003].

Code Coverage

Code coverage is an analysis method that determines which parts of a particular system have been executed (covered) during a test run [Miller and Maloney, 1963, Graham et al., 2006]. Well known examples of code coverage tools are *EMMA*[5] and *Cobertura*[6] (see [Yang et al., 2006] for more examples).

In the scope of *SFL*, by using a code coverage tool in conjunction with a test suite, it is possible to create spectra and thereby identify which components are more likely to be involved in the test suite failure. To solve the potential scalability issues that *SFL* techniques may have when instrumenting large software programs, a dynamic instrumentation approach, called **Dynamic Code Coverage (*DCC*)**, was proposed [Perez, 2012]. This technique automatically adjusts the instrumentation granularity of the system under analysis. First, the system is instrumented using a coarse granularity (*e.g.*, package level in *Java*). Then, *SFL* is executed and, based on the diagnostic results, *DCC* decides which components should be re-instrumented with a finer granularity level (*e.g.*, in *Java* one should instrument classes, then methods, and finally statements). After

[5]`http://emma.sourceforge.net/`
[6]`https://cobertura.github.io/cobertura/`

re-instrumenting the selected components, the tests that activate the re-instrumented components are re-executed. This loop is repeated until *SFL* yields an useful diagnostic report.

DCC, as an iterative technique, is aimed at improving the execution time of the fault localization procedure, and shows a substantial reduction of the execution time (27% on average) and the diagnostic report size (67% on average), when compared to normal *SFL* executions [Perez, 2012]. However, for small projects, the overhead of re-instrumenting and re-running tests may consume more time than performing a single iteration with a fine-grained instrumentation throughout the entire project. To prevent this, an approach that uses a lightweight topology model of the system to decide whether or not *DCC* should be used was proposed [Perez et al., 2012].

Code coverage is also used in the scope of **Graphical User Interface (*GUI*)** testing as an aid in determining whether a *GUI* has been adequately tested [Memon et al., 2001].

Profilers

Profiling is a dynamic analysis that gathers some metrics from the execution of a system, such as memory usage and frequency/duration of component activations. Even though the main use of profilers is in system optimization processes, it is also useful for debugging purposes, such as:

- Knowing if components are being called more or less often than expected.
- Finding if certain components execute slower than expected or if they contain memory leaks.
- Investigating the behavior of lazy evaluation strategies.

Well known profiling tools include *GNU*'s *gprof* [Graham et al., 1982], *Valgrind* [Nethercote and Seward, 2003], and *Java's VisualVM*[7].

Run-time Spectra Collection

Research on *SFL* has been mostly targeted at development-time environments. In such environments, the spectrum is normally used to diagnose errors occurring during the execution of a test suite. The availability of a test suite enables a simple approach to collect spectra due to the fact that every single unit test corresponds to a transaction in the spectrum abstraction. Also, as the purpose of a test suite is to evaluate the correctness of a program with regard to a specification, the error information is also "freely" available. Furthermore, due to the fact that unit tests normally target a set of tightly coupled components (normally contained within a single process) it is possible to perform a direct monitoring by injecting a code-level instrumentation.

[7]http://visualvm.java.net/

In contrast to development-time, in run-time environments the process of obtaining the spectra required to perform diagnosis is much more complex due to several aspects. First, performing active (and controlled) tests (*e.g.*, unit tests) on deployed systems is normally not possible without disrupting their services. The impossibility of using controlled tests to evaluate the correctness of the system, forces the spectra collection mechanisms to rely on coarse grained invariants (*e.g.*, response time, memory usage, *etc.*) to assert the validity of a set of components' activation. This is normally accomplished by encoding such constraints using an **Architecture Definition Language (*ADL*)** (*e.g.*, [Garlan et al., 2000]).

Second, as the concept of transaction is system-dependent, some mechanism must be used to relate component activity with transactions. A common approach is to use an *ADL* specification of the system in which the relevant communication paths between components are defined and, the activation of such paths, generates transactions. In [Casanova et al., 2011], the authors use a form of message sequence charts to define the relevant computations and how they relate to form transactions. In [Casanova et al., 2013], the authors propose an *ADL* that features a set of constructs that enable the encoding of rules describing transaction types. In order to ease the definition of the rules, this *ADL* makes use of object oriented concepts such as inheritance, enabling model reuse and refinement. In [Piel et al., 2012], the authors divide the system execution in transactions by setting an arbitrary transaction frequency. This approach requires that an appropriate frequency value is defined in order to produce useful spectra. On the one hand, if the frequency is too low, transactions that activate almost all components will be created[8]. On the other hand, if the frequency is too high, the components that triggered failures may not be contained in any of the failing transactions. In order to reduce the sensitivity of the approach with regard to the frequency parameter, the authors propose that an exponential distribution (with average equal to the target frequency) is used to generate random transaction durations.

Third, most systems operate continuously raising the challenge of determining which transactions are of interest for diagnostic purposes. In contrast to development-time, in a run-time environment the observations may age with time and reflect old states of the system (*e.g.*, having a failing transaction in the spectra caused by an already fixed problem) and hamper the diagnosis quality. On the one hand, if all the available spectra are used, it is possible that outdated information may degrade the diagnosis system performance. On the other hand, if too few transactions are used, the diagnosis system may not be able to isolate the correct candidate. In [Casanova et al., 2013], the authors discard the spectra every time a correction is made on the system and collect spectra until the entropy in the diagnosis report is below an arbitrary threshold. The entropy is used to characterize the (im)purity of a ranking of diagnosis candidates. The idea is to adapt the time window considering the entropy, knowing that a larger spectrum normally decreases the entropy of the ranking. In [Piel et al., 2012], the authors use a time-based sliding window at the end of which the transactions are discarded.

Fourth, most systems do not provide monitoring interfaces. In [Casanova et al., 2013], the authors instrument the system by intercepting operating system calls (*e.g.*, bind, accept, socket, *etc.*) which are then translated and fed to the rule system.

[8]The consequence of activating all components is that the resulting spectra has low entropy which, in practice, results in a low quality diagnosis.

6.4 Automated Diagnosis

Delta Debugging

Delta Debugging is a technique that tries to systematically simplify the input that leads a certain system to a failure [Zeller and Hildebrandt, 2002]. This is accomplished by iteratively reducing the size of the input until the smallest input that causes the execution to fail is reached [Zeller, 2002, Cleve and Zeller, 2005]. This is done under the assumption that that simpler inputs activates less components than more complex inputs and, therefore, renders the system easier to debug.

Input	1	2	3	4	5	6	7	8	Error
1	●	●	●	●	●	●	●	●	1
2	●	●	●	●	.	.	.	.	?
3	.	.	.	.	●	●	●	●	1
4	.	.	.	.	●	●	.	.	0
5	.	.	.	.	.	.	●	●	1
6	.	.	.	.	.	.	●	.	1
7	.	.	.	.	.	.	.	●	0

Figure 6.2: Delta debugging example (adapted from [Zeller and Hildebrandt, 2002] and [Perez et al., 2014])

Consider the example in Figure 6.2, in which a system takes as input a set of integers in the interval $[1, 8]$. Initially, the system is executed with an input consisting of all possible elements in the input space (input 1 in the figure). As this input makes the system fail, the next step is to split the input, as is the case in inputs 2 and 3. It is important to note that, when manipulating the system's input, one can reach one of three possible outcomes: passed (0), failed (1), and unresolved $(?)$. The latter occurs when the system receives an invalid input.

The next steps of the delta debugging algorithm are to keep attempting to reduce the size of the failure inducing inputs. In this example, input 3 is divided into inputs 4 and 5, and, in turn, input 5 is divided into inputs 6 and 7. As can be seen in the figure, input 6 is a minimal failure inducing input.

Since a vast amount of combinations may result in passed/unresolved outcomes, in [Misherghi and Su, 2006], the authors propose an improved algorithm, called *Hierarchical Delta Debugging*. This technique takes into account the input structure to reduce the number of inputs to examine. By initially using a coarser level of detail, the algorithm is able to prune large irrelevant portions of the input early in the minimization process. Besides speeding up the delta debugging process, the hierarchical technique also has the advantage of producing better (and more easily understandable) diagnostic reports, as the output is a structured tree [Perez et al., 2014].

Static/Dynamic Slicing

Static Slicing [Weiser, 1982, 1984] is a debugging approach that starts from the failure and uses the control and data flow of the system as a backwards reasoning method to narrow down the set of potential faulty components. This is accomplished by ignoring all components that have no data or control dependencies to the variables used in the error detection process. In this context, a *slice* is a subset of the system's components that directly or indirectly influences the values of a given set of state variables.

A problem of static slicing is that the slices computed by means of static analysis tend to be large. In [Lyle and Weiser, 1987], the authors were able to reduce the slices' size by constructing *dices*. A *dice* is the set difference between two static slices of a system. To further decrease the slices' size, in [Kolettis and Fulton, 1995], the authors propose an approach, called *Dynamic Slicing*, that relies on execution information to determine which components belong to the slice (or dice).

Dynamic slices occasionally omit components responsible for the system's failures. To mitigate this problem, in [Zhang and Gupta, 2004, Zhang et al., 2005, 2007], the authors introduced the concepts of dynamic dependence graph, implicit dependencies and relevant slicing, in [Agrawal et al., 1995], the authors propose the usage of execution slices and, in [Wong and Qi, 2006], the usage of inter-block data dependencies.

In [Wotawa, 2010], the authors improve dynamic slicing by combining it with *MBD* techniques. In [Hofer and Wotawa, 2012], the authors show that the aforementioned can be further improved by combining it with *SFL*. In [Xie et al., 2010, 2013], the authors introduce the concept of metamorphic slice with the goal of enabling the usage of *SFL* without the need for test oracles.

Similarity-based *SFL*

An alternative approach to *SFL* is to use similarity coefficients as a way of determining which columns of matrix A resemble the most with the error vector e the most or, in other words, which components have the activity pattern most similar to the error pattern. The underlying assumption of this approach is that a component with an activation pattern resembling the error pattern is likely to be at fault.

Similarity coefficient-based approaches are known for being extremely lightweight at the cost of only being capable of proposing single component candidates. Given that the candidate generation problem is implicitly solved by making such assumption, all M components (*i.e.*, all the single fault candidates) are ranked according to their similarity scores.

Several similarity coefficients do exist [Abreu et al., 2009a]. Well known examples include the *Jaccard* coefficient s_J used in the *Pinpoint* tool [Chen et al., 2002], the s_A coefficient used by the *AMPLE* tool [Dallmeier et al., 2005] and the s_T coefficient used in the *Tarantula* tool [Jones and Harrold, 2005]:

$$\boldsymbol{s_J}(j) = \frac{\boldsymbol{n}_{11}(j)}{\boldsymbol{n}_{11}(j) + \boldsymbol{n}_{01}(j) + \boldsymbol{n}_{10}(j)} \tag{6.1}$$

$$\boldsymbol{s_A}(j) = \left| \frac{\boldsymbol{n}_{11}(j)}{\boldsymbol{n}_{01}(j) + \boldsymbol{n}_{11}(j)} - \frac{\boldsymbol{n}_{10}(j)}{\boldsymbol{n}_{00}(j) + \boldsymbol{n}_{10}(j)} \right| \tag{6.2}$$

$$\boldsymbol{s_T}(j) = \frac{\frac{\boldsymbol{n}_{11}(j)}{\boldsymbol{n}_{11}(j)+\boldsymbol{n}_{01}(j)}}{\frac{\boldsymbol{n}_{11}(j)}{\boldsymbol{n}_{11}(j)+\boldsymbol{n}_{01}(j)} + \frac{\boldsymbol{n}_{10}(j)}{\boldsymbol{n}_{10}(j)+\boldsymbol{n}_{00}(j)}} \tag{6.3}$$

where:

- $\boldsymbol{n}_{11}(j)$ is the number of failed runs in which component j is involved.
- $\boldsymbol{n}_{10}(j)$ is the number of passed runs in which component j is involved.
- $\boldsymbol{n}_{01}(j)$ is the number of failed runs in which component j is not involved.
- $\boldsymbol{n}_{00}(j)$ is the number of passed runs in which component j is not involved.

Formally, n_{pq} is defined as:

$$\boldsymbol{n}_{pq}(j) = |\{i \mid A_{ij} = p \wedge e_i = q\}| \tag{6.4}$$

One of the best performing similarity coefficients for fault localization is the *Ochiai* coefficient [Abreu et al., 2007]. The *Zoltar* [Janssen et al., 2009] and *GZoltar* [Campos et al., 2012] fault localization tools use the *Ochiai* coefficient to quantify the resemblance between components' activity and the error vector. This coefficient was initially used in the molecular biology domain [e Silva MeyerDada al., 2004], and is defined as follows:

$$\boldsymbol{s_O}(j) = \frac{\boldsymbol{n}_{11}(j)}{\sqrt{\left(\boldsymbol{n}_{11}(j) + \boldsymbol{n}_{01}(j)\right) \cdot \left(\boldsymbol{n}_{11}(j) + \boldsymbol{n}_{10}(j)\right)}} \tag{6.5}$$

It is important to note, however, that, even though similarity scores range between 0 and 1, they are not considered to be probabilities [Li and Vitányi, 2013]. In fact, the *Ochiai* coefficient corresponds to the the cosine of the angle between 2 vectors in a n-dimensional space.

Due to its good performance at an extremely low cost, similarity-based *SFL* has already been applied in a variety of domains:

- In [Hofer et al., 2013], the authors compare the techniques presented in [Wotawa, 2010] (Spectrum-enhanced Dynamic Slicing), [Abreu et al., 2009b] (*SFL*), and [Abreu et al., 2012] (Constraint-based Debugging) when applied in spreadsheet debugging. Promising results were observed for both spectrum-based approaches.
- In [Abreu et al., 2014], the authors use *SFL* together with spreadsheets smells [Cunha et al., 2012a,b, Hermans et al., 2012a,b] to detect and diagnose problems in spreadsheets.

- In [Machado et al., 2013], the authors use *SFL* for debugging mobile applications.
- In [Passos et al., 2014, 2015], the authors apply *SFL* in the scope of multi-agent systems.
- In [Zoeteweij et al., 2007], the authors apply *SFL* in the scope of embedded systems.
- In [Perez and Abreu, 2014], the authors use *SFL* for software comprehension.

Candidate Generation

In [Reiter, 1987], the authors propose a breadth-first search algorithm to solve the *MHS* problem. Later, in [Wotawa, 2001, Greiner et al., 1989], some improvements over the base algorithm have been suggested.

In [Zhao and Ouyang, 2007] a method using set-enumeration trees to derive all *MHSs* in the context of model-based diagnosis is presented. While sound and complete, such algorithms do not scale to large problems. To overcome this problem, in [Feldman et al., 2008], the authors propose a stochastic search algorithm, that starts with a *HS* d for (U, S) and iteratively removes elements from d while guaranteeing that the resulting set still is a *HS*. By using a stochastic search, the algorithm trades-off soundness/completeness for computational efficiency.

In contrast to our approach, all of the above algorithms make use of a constraint solver to check whether or not a set d is a *HS*. On the one hand, such approaches do not require an explicit availability of set S, making them useful for scenarios where the problem is not explicitly stated but rather indirectly observed. On the other hand, as a great part of the “heavy-lifting” is performed by third-party libraries, their performance is largely limited by the performance of the underlying constraint solver. Furthermore, and contrary to our work, the mentioned algorithms do not use any other information aside from the problem statement (U, S) (recall that our heuristic is arbitrary and can use information outside of the problem statement).

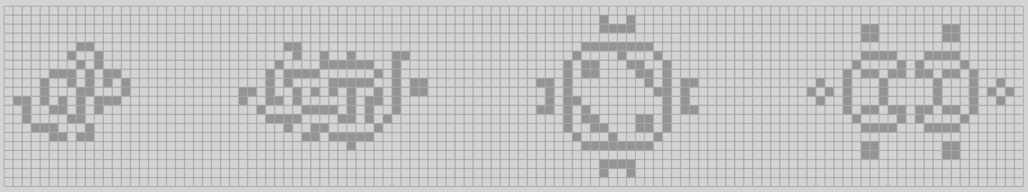
In [Fijany and Vatan, 2004, 2005], the *MHS* problem is mapped onto an $1/0$-integer programming problem. Contrary to our work, this approach does not use any other information aside from the problem statement (U, S).

In [Vinterbo and Øhrn, 2000, Li and Yunfei, 2002, Aickelin and Dowsland, 2004, Huang et al., 1994], several genetic algorithms to compute *MHSs* are proposed. While scalable to large problems, these algorithms do not guarantee soundness nor completeness.

Visualizations

Several authors acknowledged the fact that humans have difficulties in interpreting a diagnostic report as presented in Definition 6 (page 9) [Jones, 2004]. To improve the user-friendliness and thereby the utility of automated diagnostic tools, several visual approaches were proposed:

- In [Jones et al., 2002, Jones and Harrold, 2005], the authors present a technique that uses color to visually map the participation of each component in the outcome of the execution of the program with a test suite.
- In [Orso et al., 2004], the authors propose a generalization of approach presented in [Jones et al., 2002] that enables a color-based representation of bi-dimensional component data by both using the hue and the brightness values.
- In [Bouillon et al., 2007], the authors present a color coded call-graph in which the same color code is used.
- In [Janssen et al., 2009], the authors present a tool that colors each line in the application's source code according to its *SFL* score.
- In [Campos et al., 2012], the authors improve the tool by adding two visualizations: *Treemap* and *Sunburst* [Riboira et al., 2011].
- In [Gouveia et al., 2013], the authors re-implement the same visualizations using *HTML5* to improve the tool's portability.



```
++++[+++++>---<]>.+[--->+<]>+++.-.-----------.+++++++++.+++++++++.--.--
--------.++++++.-.+++++.>><>+<-<<--->.<<->.<>>>>-<<>+-<+><-><>-<+-<<>+-+
```

7 Conclusions

In this section we draw conclusions about the work developed in the scope of this thesis.

This chapter is divided as follows. First, we answer the research questions introduced in Section 1.3. Second, we list the contributions that resulted from the developed work. Finally, we discuss possible paths of future research.

7.1 Research Questions

Research Question 1 — Is it possible to optimize *Staccato* to minimize redundant/superfluous computations?

We found that the *Staccato* algorithm could be improved to achieve a higher computational efficiency. To this end we proposed 3 optimizations. The first optimization prevents multiple examinations of the same set. The second optimization preemptively filters elements guaranteed not to form *MHSs*. The third optimization prevents the examination of search tree branches guaranteed not to yield any *MHSs*.

The conducted benchmarks showed that the proposed optimizations resulted, on average, in a $34000\times$ performance improvement for small problems and in a $186\times$ improvement for large problems.

Research Question 2 — Is it possible to parallelize *Staccato* as way of reducing the diagnostic latency?

The *Staccato* algorithm can be successfully parallelized. To achieve this goal, we proposed a Map-Reduce parallelization of *Staccato*. The parallel algorithm makes use of a pseudo-random generator to evenly distribute the computations among threads without needing any communication apart from the initial seed sharing.

The conducted benchmarks showed that the proposed parallelization approach scales with minimal overheads: the algorithm exhibits a similar throughput per thread in both small/large problems and sequential/parallel setups.

Research Question 3 — How to encode fuzzy error symptoms?

We proposed a fuzzy logic generalization over the classical binary error representation. Instead of classifying transactions in terms of pass/fail, we suggest the usage of set membership functions ($\boldsymbol{\mu}_{\tilde{\boldsymbol{x}}}$). Such functions map an arbitrary input onto the interval $[0, 1]$, enabling the representation of 3 types of transaction outcomes: correct ($\boldsymbol{\mu}_{\tilde{\boldsymbol{F}}} = 0$), incorrect ($\boldsymbol{\mu}_{\tilde{\boldsymbol{F}}} = 1$) and degraded ($0 < \boldsymbol{\mu}_{\tilde{\boldsymbol{F}}} < 1$).

A consequence of the new error model is that a degraded transaction exhibits both correct and incorrect behaviors simultaneously, however with different degrees. This is due to the fact that the pass and fail sets are complimentary and, therefore, the sum of both set membership functions must be equal to 1 (*i.e.*, $\boldsymbol{\mu}_{\tilde{\boldsymbol{P}}}(x) = 1 - \boldsymbol{\mu}_{\tilde{\boldsymbol{F}}}(x)$).

Research Question 4 — How to improve *SFL* to make use of the fuzzy error information?

To use the fuzzy error information we improve the *SFL* framework by making use of the concept of probability of a fuzzy event, which is defined as:

$$\boldsymbol{Pr}(\alpha) = \sum_{x \in \Omega} \boldsymbol{\mu}_{\tilde{\boldsymbol{x}}}(\alpha) \cdot \boldsymbol{Pr}(x) \tag{7.1}$$

where Ω is the set of possible outcomes which, in the case of error diagnosis, is equal to $\{P, F\}$.

Applying the above formula to the *SFL* framework results in the following generalization:

$$\boldsymbol{Pr}(d \mid A, e) = \boldsymbol{Pr}(d) \times \frac{\prod_{i \in 1..N} \overbrace{e_i}^{\boldsymbol{\mu}_{\tilde{\boldsymbol{F}}}(\alpha)} \cdot \overbrace{\big(1 - \boldsymbol{G}(d, A_i)\big)}^{\boldsymbol{Pr}(x=F|A,d)} + \overbrace{(1 - e_i)}^{\boldsymbol{\mu}_{\tilde{\boldsymbol{P}}}(\alpha)} \cdot \overbrace{\boldsymbol{G}(d, A_i)}^{\boldsymbol{Pr}(x=P|A,d)}}{\boldsymbol{Pr}(A, e)} \tag{7.2}$$

The classical *SFL* approach (see Equation (1.14), page 18) is a special case of the proposed generalization when $\forall_{e_i} : e_i \in \{0, 1\}$.

The conducted benchmarks showed that, for our setup, the fuzzy approach improved the diagnostic quality in 65% of the test cases and performed at least as good as the classical approach in 94% of the test cases. Furthermore, the fuzzy diagnostic was, on average, 20% more accurate than classical approach.

Research Question 5 — How to enable *SFL* to adapt to different systems based on previous diagnostic experience?

To enable *SFL* to make use of previous diagnostic experience, we proposed an improved goodness modeling approach. To this end, we introduced the concept of a feedback loop in *SFL*, consisting of a pass (Fb_{0j}) and a fail (Fb_{1j}) observation lists for each component j. Each list is defined as:

$$Fb_{ej} = \langle st_1, ..., st_k, ..., st_K \rangle \tag{7.3}$$

where st represents a set of system variables' values in which component j was observed to perform either nominally or erroneously.

Each list is then used to create a *KDE* as follows:

$$\hat{\boldsymbol{f}}_{\boldsymbol{ej}}(st) = \frac{1}{bw} \sum_{st' \in Fb_{ej}} \boldsymbol{K}\left(\frac{st - st'}{bw}\right) \tag{7.4}$$

Intuitively, the *KDE* works like a continuous histogram, estimating how the pass/fail states are spread across the entire state space.

The final modeling step consist of using both the pass and fail *KDEs* to estimate the component's goodness. Formally, the improved goodness model is defined as:

$$\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) = \frac{\hat{\boldsymbol{f}}_{\boldsymbol{0j}}(st)}{\hat{\boldsymbol{f}}_{\boldsymbol{0j}}(st) + \hat{\boldsymbol{f}}_{\boldsymbol{1j}}(st)} \tag{7.5}$$

Research Question 6 — How to incorporate information about the system's state in *SFL*?

To make use of the system state in the diagnostic process, we generalized the spectrum's activity matrix as:

$$A_{ij} = \begin{cases} \clubsuit, & \text{if } c_j \text{ does not have a } \breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) \text{ model and was involved in transaction } i \\ \langle st_1, \cdots, st_k \rangle, & \text{if } c_j \text{ has a } \breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) \text{ model and was involved in transaction } i \\ \emptyset, & \text{otherwise} \end{cases} \tag{7.6}$$

The improved spectrum enables the capture of the system's state which, in turn, is plugged onto the improved goodness models.

To incorporate the improved spectrum and goodness models onto the *SFL* framework, we define the generalize the transaction goodness function as:

$$\boldsymbol{G}(d, A_i) = \prod_{j \in (d \cap A_i)} \begin{cases} g_j, & \text{if} A_{ij} = \clubsuit \\ (1 - \alpha_j) \cdot g_j + \alpha_j \cdot \breve{\boldsymbol{G}}(j, A_{ij}), & \text{otherwise} \end{cases} \tag{7.7}$$

$$\breve{\boldsymbol{G}}(j, S) = \prod_{st \in S} \breve{\boldsymbol{g}}_{\boldsymbol{j}}(st) \tag{7.8}$$

where $\alpha_j \in [0, 1]$ is a model confidence parameter. This parameter enables the *SFL* framework to fallback to the *MLE* goodness model as the state-aware goodness model gradually becomes obsolete. In fact, the classical *SFL* approach (see Equation (1.15), page 18) is a special case of the proposed generalization when $\forall_{\alpha_j} : \alpha_j = 0$.

7.2 Contributions

Overall, the work developed for this thesis resulted in 3 main contributions:

1. We developed a *SFL* candidate generation algorithm that is both more efficient than the state-of-the-art algorithm (Section 2.1, page 27) and can make use of multiple *CPUs* to compute diagnostic candidates (Section 3.1, page 47). Furthermore, we released a reference implementation under the GNU Lesser General Public License[1], available at `https://github.com/npcardoso/MHS2`.

2. We proposed a fuzzy logic-based approach to error detection enabling a more accurate representation of performance degradation errors (Section 4.1.1, page 57). Furthermore, we improved the spectrum-based reasoning framework for fault localization to incorporate the improved error representation (Section 4.1.2, page 60).

3. We proposed a *KDE*-based approach to model the components' goodnesses as a function of the system state (Section 5.1.1, page 71). Furthermore, we improved the spectrum-based reasoning framework for fault localization to incorporate the improved goodness estimation (Section 5.1.2, page 74).

In addition to the main contributions, in the scope of this thesis:

- We introduced the state spectrum data structure, which generalizes over a set of existent types of spectra (*e.g.*, hit spectrum, count spectrum, *etc.*) and enables a more precise abstraction of the system's executions.

- We introduced the concept of a feedback loop in the scope of *SFL*.

7.3 Future Work

In this section we discuss future research topics related to the material presented in this thesis.

7.3.1 Goodness Modeling

Feedback Generation

Given a set of state spectra and their respective correct diagnostic candidates, it should be possible to automatically extract information to generate the feedback spectra. To illustrate how this could be accomplished in practice, consider the state spectrum in Figure 7.1 for which the correct diagnostic candidate was determined to be $d = \{1, 2\}$.

[1]`https://www.gnu.org/licenses/lgpl.html`

i	A			e
	c_1	c_2	$\cdots$	
1	$\langle 0.42 \rangle$	$\emptyset$	$\cdots$	0
2	$\langle 0.86, 1.86 \rangle$	$\emptyset$	$\cdots$	0
3	$\langle 0.14 \rangle$	$\langle \cdots \rangle$	$\cdots$	0
4	$\langle 0.58, 1.46 \rangle$	$\langle \cdots \rangle$	$\cdots$	0
5	$\langle 1.05 \rangle$	$\emptyset$	$\cdots$	1
6	$\langle 1.35 \rangle$	$\emptyset$	$\cdots$	1
7	$\langle 1.79 \rangle$	$\emptyset$	$\cdots$	1
8	$\langle 0.49, 1.34 \rangle$	$\emptyset$	$\cdots$	1
9	$\langle 1.67 \rangle$	$\langle \cdots \rangle$	$\cdots$	1
10	$\langle 0.49, 0.81 \rangle$	$\langle \cdots \rangle$	$\cdots$	1

Figure 7.1: State spectrum

Analyzing the spectrum, we can see that the states of the transactions in which c_1 was activated can be divided into 3 categories: nominal, error, and undetermined. The categorization is a function of 3 factors: the number of activations of the target component, the number of activated faulty components, and the outcome of the transaction (Figure 7.2).

Component States	Faulty Components	Transaction Outcome	
One	One	Pass	Nominal
More than one	One	Pass	Nominal
One	More than one	Pass	Nominal
More than one	More than one	Pass	Nominal
One	One	Fail	Error
More than one	One	Fail	Undetermined
One	More than one	Fail	Undetermined
More than one	More than one	Fail	Undetermined

Figure 7.2: State categorization

The nominal and error states can be directly added to the feedback spectrum whereas, the undetermined states cannot. The simplest approach for handling undetermined states is to simply ignore them. The major drawback of this approach is that the available information is not used to its full extent.

A more convoluted approach is to make use of a clustering algorithm such as **K-Nearest Neighbors (*KNN*)** [Altman, 1992]. In short, *KNN* classifies a point based on the most frequent class among its K nearest (classified) neighbors. Figure 7.3 illustrates how the undetermined states in Figure 7.1 would be classified by *KNN* with $K = 3$. It is worth to note that all undetermined states should be reclassified every time new diagnosed state spectra is available for feedback extraction.

Another possible approach is to still use the *KNN* algorithm but, instead of all states having equal weight in the goodness calculation, they may have different weights. To this

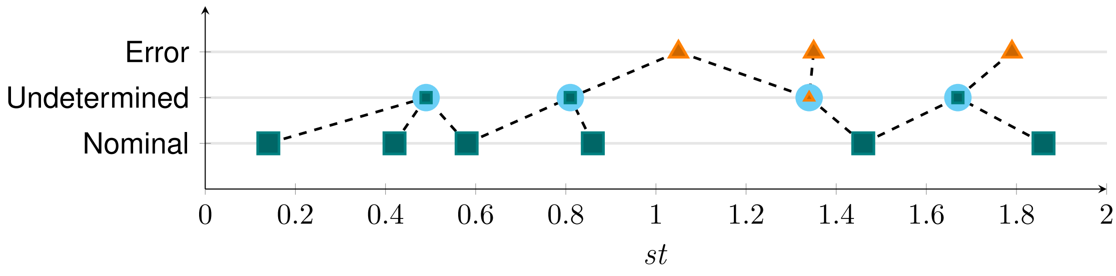


Figure 7.3: *KNN*-based feedback spectrum extraction example

end, we generalize Equation (5.1) as:

$$Fb_{ej} = \langle (st_1, w_1), ..., (st_k, w_k), ..., (st_K, w_K) \rangle \tag{7.9}$$

where w_k represents the weight of observation st_k. To account for this generalization, we redefine Equation (5.2) as:

$$\boldsymbol{\hat{f}_{ej}}(st) = \frac{1}{bw} \sum_{(st',w) \in Fb_{ej}} w \times \boldsymbol{K}\left(\frac{st - st'}{bw}\right), w \in [0,1] \tag{7.10}$$

Under this generalization both the pass and fail states should have $w = 1$. Each undetermined state should be added to both the pass and fail feedback spectra with weights w_p and w_f such that the sum of both weights is equal to 1. The weights of the undetermined states should be calculated based on their K-nearest neighbors.

Let p and f be the lists of pass and fail neighbor states for an underarm state st, respectively, and $\boldsymbol{dist}(a,b)$ be an arbitrary distance metric (*e.g.*, Euclidean, Manhattan, Minkowski, *etc.*). Two possible strategies for calculating w are (note that $w_f = 1 - w_p$):

Number of neighbors: $w_p(st, p, f) = \dfrac{|p|}{|p| + |f|}$

Distance to neighbors: $w_p(st, p, f) = \dfrac{\sum_{st' \in p} \boldsymbol{dist}(st, st')^{-1}}{\sum_{st' \in (p \cup f)} \boldsymbol{dist}(st, st')^{-1}}$

Since this method for generating feedback spectra has not been tested, future work in this scope would include evaluating the improvements introduced by this approach. Furthermore, and since this is just one possible approach of generating feedback spectra, more research should done towards automating this process.

Goodness Model Confidence

A problem inherent to the usage of feedback data to create goodness models is that the available observations may not cover all state space evenly. As a consequence, the model's reliability/confidence may not be constant throughout the state space.

As an illustrative example, consider both the pass and fail *KDEs* depicted in Figure 7.4. We can see that, for this example, there are zones in the state space for which the sum of both *KDEs* has a large magnitude and, in other zones, the sum of both *KDEs* has a very small magnitude. Intuitively, we expect the model to perform more accurately for $st \in [9, 18]$ than, for instance, $st \in [0, 9]$. In particular, for $st \in [5, 7]$, the sum's magnitude is approximately equal to 0 and, as a consequence, the model is expected to perform poorly.

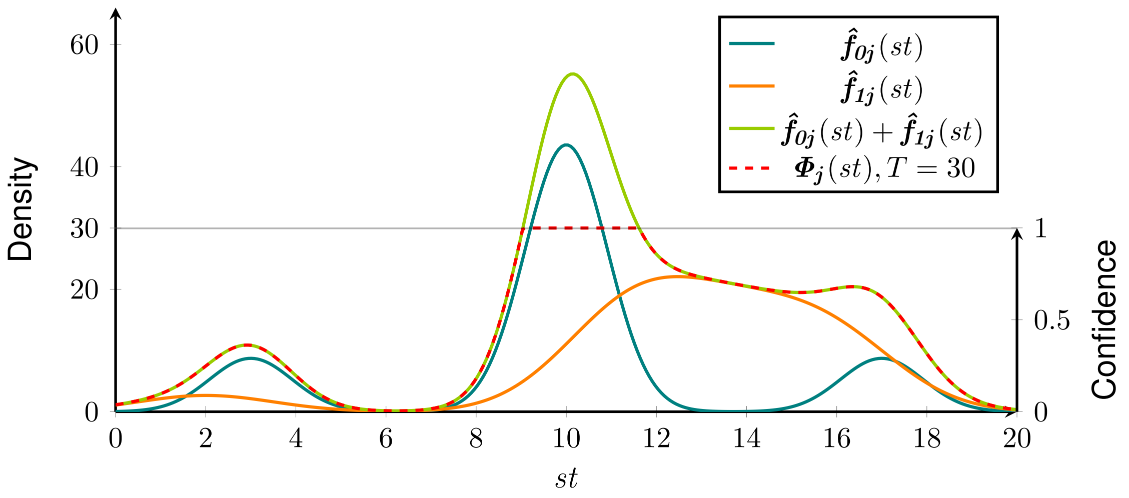


Figure 7.4: Variable model confidence example

To mitigate this problem and as discussed in Section 5.1.2 (page 74), it is possible to leverage the goodness estimations of both the *NFGE* and *MLE* approaches. In contrast to Equation (5.10) (page 75), in which we can define the model's confidence as a constant, to solve this problem it is required to define the model's confidence as a function of the state st. In order to convey this idea to the diagnostic algorithm, we generalize Equation (5.7) (page 74) as:

$$\breve{G}(j, S) = g_j \times \prod_{st \in S} 1 - \Phi_j(st) + \prod_{st \in S} \Phi_j(st) \cdot \breve{g}_j(st) \tag{7.11}$$

where $\Phi_j(st) \in [0, 1]$ estimates the confidence we have on the goodness calculated by $\breve{g}_j(st)$. Equation (5.7) is special case of Equation (7.11) when $\Phi_j(st) = 1$.

A possible approach to estimate $\Phi_j(st)$ is:

$$\Phi_j(st) = \frac{min(\hat{f}_{0j}(st) + \hat{f}_{1j}(st), T)}{T} \tag{7.12}$$

where T is the number of observations required to have full confidence on the model's estimation. In Figure 7.4 $\Phi_j(st)$ is plotted for $T = 30$.

Goodness Model Observations' Aging

Another problem inherent to the usage of feedback data is related to the fact that components may evolve over time. As a consequence, older feedback observations may become obsolete and, consequently decrease the diagnostic accuracy.

To account for this fact, it is necessary to record the time of occurrence of each feedback observation. This can be accomplished by generalizing Equation (7.9) (page 98) as:

$$Fb_{ej} = \langle (st_1, w_1, t_1), ..., (st_k, w_k, t_k), ..., (st_K, w_K, t_K) \rangle \tag{7.13}$$

where t_k represents the time of occurrence of observation st_k.

To make use of t_k in the modeling process, Equation (7.11) can be generalized as:

$$\hat{\boldsymbol{f}}_{\boldsymbol{ej}}(st) = \frac{1}{bw} \sum_{(c,w,t) \in Fb_{ej}} w \times \big(1 - \boldsymbol{Ag_j}(t)\big) \times \boldsymbol{K}\left(\frac{st - c}{bw}\right) \tag{7.14}$$

where $\boldsymbol{Ag_j} \in [0, 1]$ is an arbitrary aging function for component j. Equation (7.11) is special case of Equation (7.14) when $\boldsymbol{Ag_j}(t) = 0$.

Assuming that this approach is valid, future work on this topic would include devising methods for estimating the aging functions.

Ensemble Modeling

Ensemble learning methods use multiple learning algorithms to obtain better predictive performance than could be obtained from any of the constituent learning algorithms [Opitz and Maclin, 1999]. Applying this concept to *SFL* and assuming other goodness modeling approaches will be proposed in the future, it is possible to improve $\breve{\boldsymbol{G}}(j, S)$ so it can make use of multiple $\breve{\boldsymbol{g}}_{\boldsymbol{j}}(st)$ models per component, by weight averaging and normalizing the result of multiple estimators. For this purpose, let $\breve{\boldsymbol{g}}_{\boldsymbol{jx}}(st)$ and W_{jx} be the x^{th} goodness estimator and respective weight for component j. The improved $\breve{\boldsymbol{G}}(j, S)$ function can be redefined as:

$$\breve{\boldsymbol{G}}(j, S) = \prod_{st \in S} \frac{W_{j1} \cdot \breve{\boldsymbol{g}}_{\boldsymbol{j1}}(st) + \cdots + W_{jx} \cdot \breve{\boldsymbol{g}}_{\boldsymbol{jx}}(st)}{W_{j1} + \cdots + W_{jx}} \tag{7.15}$$

Aside from formulating alternative goodness modeling approaches, future work on this topic would include devising a method to estimate the weight parameters in the above equation.

7.3.2 Candidate Generation

Future work in the scope of candidate generation would include the following:

- Improve the data structure used to encode D. We envision this data structure to function similarly to an hashtable: the *hashtrie* data structure should be composed of multiple tries which are used to divide the load according to an *HS* hashing function.

- Analyze the algorithm's performance with a larger set of computation resources as well as in a fully distributed environment.

- Explore the possibility of using the feedback data set to create better heuristics to further improve the throughput of the candidate generation algorithm.

- Explore the possibility of using frameworks such as *OpenCL*[2] or *CUDA*[3] to parallelize *MHS*^{2}.

7.3.3 Candidate Ranking

Similarity-based *SFL*

A potentially interesting research would be to apply the concept of fuzzy error to similarity-based *SFL*. This could be accomplished by redefining n_{pq} (see Equation (6.4), page 90) as:

$$\boldsymbol{n}_{pq}(j) = \sum_{A_{ij}=p} \begin{cases} e_i & \text{if } q = 1 \\ 1 - e_i & \text{otherwise} \end{cases} \tag{7.16}$$

To illustrate this approach consider again the spectrum depicted in Figure 4.4 (page 60). Figure 7.5 presents the $\boldsymbol{n}_{pq}(j)$ values for both c_1 and c_2 under Equations (6.4) and (7.16).

pq	$\boldsymbol{n}_{pq}(j)$			
	Equation (6.4)		Equation (7.16)	
	c_1	c_2	c_1	c_2
00	1	1	1	0.2
01	0	0	0	0.8
10	1	1	0.2	1
11	1	1	1.8	1

Figure 7.5: $\boldsymbol{n}_{pq}(j)$ comparison

[2] https://www.khronos.org/opencl/

[3] https://www.nvidia.com/object/cuda_home_new.html

We can see that the $\boldsymbol{n}_{pq}(j)$ values for both components under Equation (6.4) are equal. In practice this translates into ties in the ranking, independently of the chosen similarity coefficient. In contrast the $\boldsymbol{n}_{pq}(j)$ values for both components under Equation (7.16) are different. Figure 7.6 presents the components' scores for the *Jaccard* (see Equation (6.1), page 90), *Tarantula* (see Equation (6.3), page 90), and *Ochiai* (see Equation (6.5), page 90) coefficients.

Similarity coefficient	Equation (6.4)		Equation (7.16)	
	c_1	c_2	c_1	c_2
$\boldsymbol{s_J}(j)$	0.50	0.50	0.90	0.36
$\boldsymbol{s_T}(j)$	0.67	0.67	0.86	0.40
$\boldsymbol{s_O}(j)$	0.71	0.71	0.95	0.53

Figure 7.6: Similarity coefficients' comparison

We can see that, as expected, for Equation (6.4) both components are tied. In contrast, and as in Section 4.1.2 (page 60), all 3 similarity coefficients successfully ranked c_1 ahead of $c2$, thus improving the diagnostic accuracy.

Since this approach has not been validated, future work would include evaluating the improvements (if any) introduced by this generalization.

Prior estimation

The estimation of the components' prior probability of erroneous behavior (see Section 1.2.2, page 17) is a topic that, to the best of our knowledge, has almost no research devoted to it. This may attributed to the fact that, given enough observations, the impact of the prior probabilities becomes, most of the times, negligible. However, when the spectrum features ambiguity groups (*i.e.*, sets of components sharing equal activation patterns), *SFL* is unable to differentiate the elements of such groups of, thus decreasing the diagnostic accuracy. By improving the prior estimation, which currently is estimated as a constant and equal for all components, the occurrence of ambiguity groups should become less frequent, thereby improving the diagnostic accuracy.

A possible approach is to take advantage of the feedback spectrum proposed in Section 5.1.1 (page 71) and define p_j as:

$$p_j = \frac{|Fb_{1j}|}{|Fb_{0j}| + |Fb_{1j}|} \tag{7.17}$$

Since this is just one among a possibly large number of alternative approaches, more research should be devoted to devising better methods of estimating p_j.

Oracle Confidence

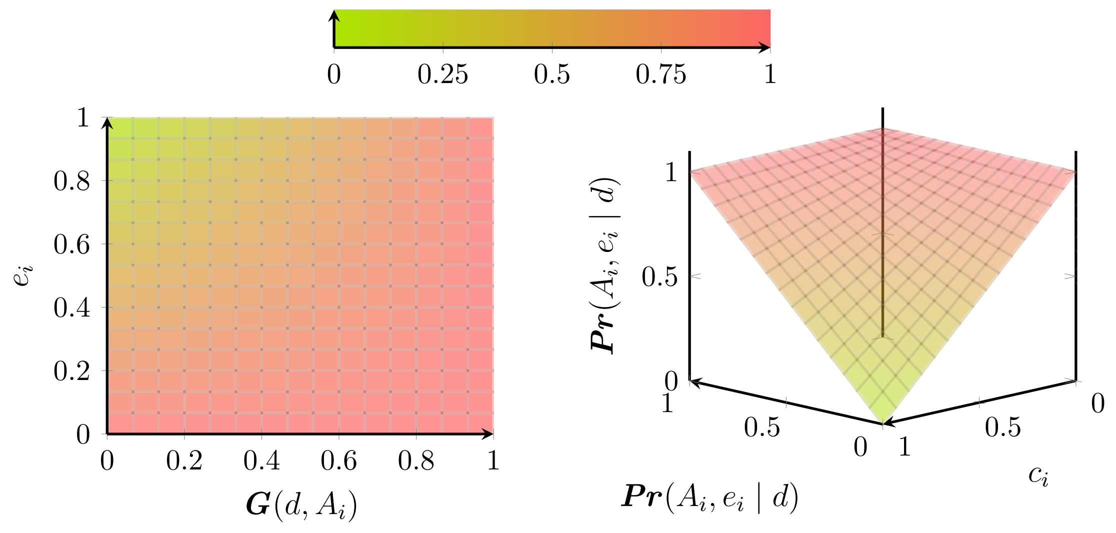
A limitation of spectrum-based approaches is related to the assumption that all assertions are equally trustworthy for diagnosis purposes. To illustrate this limitation consider the hit spectrum presented in Figure 7.7. Consider that in this example different oracles were used in each transaction. Additionally, consider that the error detection mechanism' confidence was available, as encoded in the confidence column of the hit spectrum (C). C must be contained in the interval $[0, 1]$, where $C = 0$ represents no confidence and $C = 1$ represents the maximum possible confidence. Using the approach explained Section 1.2.2 (page 17), we see that c_1 and c_2 are ranked equally. However, intuitively, we would expect c_1 to be ranked ahead of c_2 due to c_2 having a stronger evidence of nominal behavior than c_1 (t_3 has a higher oracle confidence than t_2).

i	A		e	C
	c_1	c_2		
1	●	●	1	1
2	●	.	0	0.5
3	.	●	0	1

Figure 7.7: Confidence hit spectrum example

To solve this limitation, the values in the confidence vector must then be used to weigh the impact of each observation in the diagnosis. In practice this can be accomplished by creating a generalization of Equation (1.14) (page 18) and Equation (4.4) (page 61).

$$\boldsymbol{Pr}(A_i, e_i, c_i \mid d) = (1 - c_i) + \big(c_i \cdot \boldsymbol{Pr}(A_i, e_i \mid d)\big) \tag{7.18}$$



(a) 2D view **(b)** 3D view

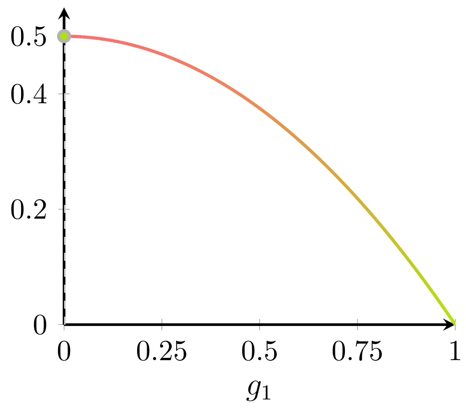
Figure 7.8: Likelihood function plot

Using the above generalization, the probabilities of the two minimal candidates are calculated as follows:

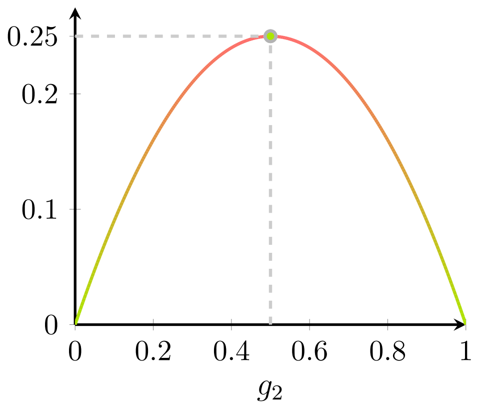
$$\begin{aligned}
\boldsymbol{Pr}(A, e, c \mid \{1\}) &= \underbrace{\big((1 - c_1) + c_1 \cdot \boldsymbol{Pr}(A_1, e_1 \mid \{1\})\big)}_{t_1} \\
&\times \underbrace{\big((1 - c_2) + c_2 \cdot \boldsymbol{Pr}(A_2, e_2 \mid \{1\})\big)}_{t_2} \\
&= \underbrace{\big((1 - 1) + 1 \cdot (1 - g_1)\big)}_{t_1} \\
&\times \underbrace{\big((1 - 0.5) + 0.5 \cdot g_1\big)}_{t_2} \\
&= (1 - g_1) \times (0.5 + 0.5 \cdot g_2)
\end{aligned} \tag{7.19}$$

$$\begin{aligned}
\boldsymbol{Pr}(A, e, c \mid \{2\}) &= \underbrace{\big((1 - c_1) + c_1 \cdot \boldsymbol{Pr}(A_1, e_1 \mid \{2\})\big)}_{t_1} \\
&\times \underbrace{\big((1 - c_3) + c_3 \cdot \boldsymbol{Pr}(A_3, e_3 \mid \{2\})\big)}_{t_3} \\
&= \underbrace{\big((1 - 1) + 1 \cdot (1 - g_2)\big)}_{t_1} \\
&\times \underbrace{\big((1 - 1) + 1 \cdot g_2\big)}_{t_3} \\
&= (1 - g_2) \times g_2
\end{aligned} \tag{7.20}$$

By performing a *MLE* for both functions it follows that $\boldsymbol{Pr}(A, e, c \mid \{1\})$ is maximized for $g_1 = 0$ and $\boldsymbol{Pr}(A, e \mid \{2\})$ for $g_2 = 0.5$ (Figure 7.9). Applying the maximizing values to both expressions, it follows that $\boldsymbol{Pr}(\{1\} \mid A, e, c) = 5 \times 10^{-4}$ and $\boldsymbol{Pr}(\{2\} \mid A, e, c) = 2.5 \times 10^{-4}$, entailing the ranking $\langle\{1\}, \{2\}\rangle$.



(a) $\boldsymbol{Pr}(A, e, c \mid \{1\})$



(b) $\boldsymbol{Pr}(A, e, c \mid \{2\})$

Figure 7.9: Likelihood plots

Since this generalization is conjectural, future work in this scope would include evaluating the improvements (if any) introduced by this generalization. Furthermore, and assuming that this approach yields positive results, research should done towards automating the process of estimating the oracles' confidence.

```
++++++++[>++++>++++++>++++++++>++++++++++>++++++++++++<<<<<-]>>>>++.>+
++++.+.-.+++++++++++++.-------------.++++++++++.-----------.++.++++++++
++++++.[>]<[[-]<]<>.<.-+<+----<>>>-<<-+<>>>++>>+<>-+++<->+<-<-->->+<<-
```

References

[Abreu et al., 2014] Abreu, R., Cunha, J., Fernandes, J. P., Martins, P., Perez, A., and Saraiva, J. (2014). Smelling faults in spreadsheets. In *Proceedings of the 2014 International Conference on Software Maintenance and Evolution*, ICSM'14, pages 111–120. (Cited on page 90)

[Abreu and Van Gemund, 2009] Abreu, R. and Van Gemund, A. J. C. (2009). A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis. In *Proceedings of the 8^{th} Symposium on Abstraction, Reformulation, and Approximation*, SARA'09. (Cited on pages 13, 20, 27, 38 and 42)

[Abreu et al., 2008] Abreu, R., González-Sanchez, A., Zoeteweij, P., and Van Gemund, A. J. C. (2008). On the performance of fault screeners in software development and deployment. In Gonzalez-Perez, C. and Jablonski, S., editors, *Proceedings of the 3^{rd} International Conference on Evaluation of Novel Approaches to Software Engineering*, ENASE'08, pages 123–130. (Cited on page 83)

[Abreu et al., 2012] Abreu, R., Riboira, A., and Wotawa, F. (2012). Constraint-based debugging of spreadsheets. In *Proceedings of the 15^{th} Ibero-American Conference on Software Engineering*, CIbSE'12, pages 9–16. (Cited on page 90)

[Abreu et al., 2007] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. C. (2007). On the accuracy of spectrum-based fault localization. In *Proceedings of the 2^{nd} International Academic And Industrial Conference on Testing – Practice And Research Techniques*, TAICPART'07, pages 89–98. (Cited on pages 38 and 90)

[Abreu et al., 2009a] Abreu, R., Zoeteweij, P., and Van Gemund, A. J. C. (2009a). A new bayesian approach to multiple intermittent fault diagnosis. In *Proceedings of the 21^{st} International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 653–658. (Cited on pages 9, 17, 18, 21 and 89)

[Abreu et al., 2009b] Abreu, R., Zoeteweij, P., Golsteijn, R., and Van Gemund, A. J. C. (2009b). A practical evaluation of spectrum-based fault localization. *Journal of*

Systems and Software, 82(11):1780–1792. (Cited on pages 4 and 90)

[Agrawal et al., 1995] Agrawal, H., Horgan, J. R., London, S., and Wong, W. E. (1995). Fault localization using execution slices and dataflow tests. In *Proceedings of the 1995 International Symposium on Software Reliability Engineering*, ISSRE'95, pages 143–151. (Cited on page 89)

[Aickelin and Dowsland, 2004] Aickelin, U. and Dowsland, K. A. (2004). An indirect genetic algorithm for a nurse-scheduling problem. *Computers & Operations Research*. (Cited on page 91)

[Allan, 2001] Allan, R. (2001). *A History of the Personal Computer: the People and the Technology*. Allan Pub, London, Ont. (Cited on page 1)

[Altman, 1992] Altman, N. S. (1992). An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185. (Cited on page 97)

[Arito et al., 2012] Arito, F., Chicano, F., and Alba, E. (2012). On the application of sat solvers to the test suite minimization problem. In *Proceedings of the 4^{th} International Conference on Search Based Software Engineering*, SSBSE'12, pages 45–59, Berlin, Heidelberg. (Cited on page 84)

[Arnold and Ryder, 2001] Arnold, M. and Ryder, B. G. (2001). A framework for reducing the cost of instrumented code. *ACM SIGPLAN Notices*, 36(5):168–179. (Cited on pages 84 and 85)

[Avizienis et al., 2004] Avizienis, A., Laprie, J.-C., Randell, B., and Landwehr, C. E. (2004). Basic concepts and taxonomy of dependable and secure computing. *Transactions on Dependable Secure Computing*, 1(1):11–33. (Cited on pages 2 and 4)

[Bala et al., 2000] Bala, V., Duesterwald, E., and Banerjia, S. (2000). Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35:1–12. (Cited on page 84)

[Bigus et al., 2002] Bigus, J. P., Schlosnagle, D. A., Pilgrim, J. R., Mills, W. N., and Diao, Y. (2002). Able: a toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3):350–371. (Cited on page 3)

[Bouillon et al., 2007] Bouillon, P., Krinke, J., Meyer, N., and Steimann, F. (2007). EzUnit: A Framework for Associating Failed Unit Tests With Potential Programming Errors. *Proceedings of the 8^{th} International Conference on Agile Processes in Software Engineering and eXtreme Programming (XP 2007), Como, Italy 2007*. (Cited on page 92)

[Bryce and Memon, 2007] Bryce, R. C. and Memon, A. M. (2007). Test suite prioritization by interaction coverage. In *Proceedings of the 2007 Workshop on Domain Specific Approaches to Software Test Automation*, DOSTA '07, pages 1–7. (Cited on page 84)

[Bus et al., 2004] Bus, B. D., Chanet, D., Sutter, B. D., Put, L. V., and Bosschere, K. D. (2004). The design and implementation of fit: a flexible instrumentation toolkit. In *Proceedings of the 5^th Workshop on Program Analysis for Software Tools and Engineering*, PASTE'04, pages 29–34. (Cited on page 85)

[Campos and Abreu, 2013] Campos, J. and Abreu, R. (2013). Encoding Test Requirements as Constraints for Test Suite Minimization. In *Proceedings of the 10^{th} International Conference on Information Technology: New Generation*, ITNG'13, pages 317–322. (Cited on page 84)

[Campos et al., 2013] Campos, J., Abreu, R., Fraser, G., and d'Amorim, M. (2013). Entropy-based test generation for improved fault localization. In *International Conference on Automated Software Engineering*, ASE'13. (Cited on page 84)

[Campos et al., 2012] Campos, J., Riboira, A., Perez, A., and Abreu, R. (2012). GZoltar: An Eclipse Plug-in for Testing and Debugging. In *Proceedings of the 2012 International Conference on Automated Software Engineering*, ASE'12, pages 378–381. (Cited on pages 90 and 92)

[Cardoso and Abreu, 2013a] Cardoso, N. and Abreu, R. (2013a). MHS^2: A map-reduce heuristic-driven minimal hitting set search algorithm. In *Proceedings of the 2013 International Conference on Multicore Software Engineering, Performance, and Tools*, MUSEPAT'13, pages 25–36. (Cited on page 24)

[Cardoso and Abreu, 2013b] Cardoso, N. and Abreu, R. (2013b). A distributed approach to diagnosis candidate generation. In *Proceedings of the 16^{th} Portuguese Conference on Artificial Intelligence*, EPIA'13, pages 175–186. (Cited on page 23)

[Cardoso and Abreu, 2013c] Cardoso, N. and Abreu, R. (2013c). A kernel density estimate-based approach to component goodness modeling. In *Proceedings of the 27^{th} AAAI Conference on Artificial Intelligence*, AAAI'13. (Cited on page 24)

[Cardoso and Abreu, 2014a] Cardoso, N. and Abreu, R. (2014a). An efficient distributed algorithm for computing minimal hitting sets. In *Proceedings of the 25^{th} International Workshop on Principles of Diagnosis*, DX'14. (Cited on page 23)

[Cardoso and Abreu, 2014b] Cardoso, N. and Abreu, R. (2014b). Enhancing reasoning approaches to diagnose functional and non-functional errors. In *Proceedings of the 25^{th} International Workshop on Principles of Diagnosis*, DX'14. (Cited on page 24)

[Carey et al., 1999] Carey, J., Gross, N., Stepanek, M., and Port, O. (1999). Software hell. In *Business Week*, pages 391–411. (Cited on page 17)

[Casanova et al., 2013] Casanova, P., Garlan, D., Schmerl, B., and Abreu, R. (2013). Diagnosing architectural run-time failures. In *Proceedings of the 8^{th} International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS'13, pages 103–112. (Cited on pages 9, 21, 57 and 87)

[Casanova et al., 2011] Casanova, P., Schmerl, B. R., Garlan, D., and Abreu, R. (2011). Architecture-based run-time fault diagnosis. In *Proceedings of the* 5^{th} *European Conference Software Architecture*, ECSA'11, pages 261–277. (Cited on page 87)

[Chao et al., 2004] Chao, K., Shah, N., Godwin, N., Younas, M., and Laing, C. (2004). Exception diagnosis in agent-based grid computing. In *Proceedings of the 2004 International Conference on Systems, Man & Cybernetics*, SMC'04, pages 3213–3219. (Cited on page 3)

[Chen et al., 2013] Chen, C., Gross, H.-G., and Zaidman, A. (2013). Improving service diagnosis through increased monitoring granularity. In *Proceedings of the* 7^{th} *International Conference on Software Security and Reliability*, SERE'13, pages 129–138. (Cited on pages 21 and 62)

[Chen et al., 2002] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., Fox, O., and Brewer, E. (2002). Pinpoint: Problem determination in large, dynamic internet services. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, DSN'2002, pages 595–604. (Cited on page 89)

[Cleve and Zeller, 2005] Cleve, H. and Zeller, A. (2005). Locating causes of program failures. In *Proceedings of the 2005 International Conference on Software Engineering*, ICSE'05, pages 342–351. (Cited on page 88)

[Cmelik and Keppel, 1995] Cmelik, B. and Keppel, D. (1995). *Shade: a Fast Instruction-Set Simulator for Execution Profiling*. Springer. (Cited on page 84)

[Cunha et al., 2012a] Cunha, J., Fernandes, J. P., Martins, P., Mendes, J., and Saraiva, J. (2012a). Smellsheet detective: a tool for detecting bad smells in spreadsheets. In *Proceedings of the 2012 Symposium on Visual Languages and Human-Centric Computing*, VL/HCC'12, pages 243–244. (Cited on page 90)

[Cunha et al., 2012b] Cunha, J., Fernandes, J. P., Ribeiro, H., and Saraiva, J. (2012b). Towards a catalog of spreadsheet smells. In *Computational Science and Its Applications*, pages 202–216. Springer. (Cited on page 90)

[Dallmeier et al., 2005] Dallmeier, V., Lindig, C., and Zeller, A. (2005). Lightweight defect localization for java. In *Proceedings of the 2005 European Conference on Object-Oriented Programming*, ECOOP'05, pages 528–550. (Cited on page 89)

[Dean and Ghemawat, 2004] Dean, J. and Ghemawat, S. (2004). Mapreduce: Simplified data processing on large clusters. In *Proceedings of the 2004 Symposium on Opearting Systems Design & Implementation*, OSDI'04, pages 137–150. (Cited on pages 20 and 47)

[Dijkstra, 1982] Dijkstra, E. W. (1982). Self-stabilization in spite of distributed control. In *Selected writings on computing: a personal perspective*, pages 41–46. Springer. (Cited on page 2)

[Elbaum et al., 2002] Elbaum, S., Malishevsky, A. G., and Rothermel, G. (2002). Test case prioritization: a family of empirical studies. *Transactions on Software Engineering*, 28(2):159–182. (Cited on page 84)

[Ernst et al., 2001] Ernst, M. D., Cockrell, J., Griswold, W. G., and Notkin, D. (2001). Dynamically discovering likely program invariants to support program evolution. *Transactions on Software Engineering*, 27(2):99–123. (Cited on page 83)

[Ernst et al., 2000] Ernst, M. D., Czeisler, A., Griswold, W. G., and Notkin, D. (2000). Quickly detecting relevant program invariants. In *Proceedings of the* 22^{nd} *International Conference on Software Engineering*, ICSE'00, pages 449–458, New York, NY, USA. (Cited on page 83)

[Ernst et al., 2007] Ernst, M. D., Perkins, J. H., Guo, P. J., McCamant, S., Pacheco, C., Tschantz, M. S., and Xiao, C. (2007). The daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69:35–45. (Cited on page 83)

[Feldman et al., 2008] Feldman, A., Provan, G., and Van Gemund, A. J. C. (2008). Computing minimal diagnoses by greedy stochastic search. In *Proceedings of the* 23^{rd} *AAAI Conference on Artificial Intelligence*, AAAI'08, pages 911–918. (Cited on pages 13 and 91)

[Fijany and Vatan, 2004] Fijany, A. and Vatan, F. (2004). New approaches for efficient solution of hitting set problem. In *Proceedings of the 2004 Winter International Synposium on Information and communication technologies*, WISICT'04. (Cited on page 91)

[Fijany and Vatan, 2005] Fijany, A. and Vatan, F. (2005). New high performance algorithmic solution for diagnosis problem. In *Proceedings of the 2005 Aerospace Conference*, IEEEAC'05. (Cited on page 91)

[Fraser and Arcuri, 2011] Fraser, G. and Arcuri, A. (2011). Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the* 19^{th} *Symposium and the* 13^{th} *European Conference on Foundations of Software Engineering*, ESEC'11/FSE'11, pages 416–419. (Cited on page 84)

[Fraser and Arcuri, 2013] Fraser, G. and Arcuri, A. (2013). Whole test suite generation. *Transactions on Software Engineering*, 39(2):276–291. (Cited on page 84)

[Fraser and Zeller, 2012] Fraser, G. and Zeller, A. (2012). Mutation-driven generation of unit tests and oracles. *Transactions on Software Engineering*, 38(2):278–292. (Cited on page 84)

[Garey and Johnson, 1990] Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co. (Cited on pages 12 and 32)

[Garlan et al., 2004] Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B. R., and Steenkiste, P. (2004). Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54. (Cited on page 3)

[Garlan et al., 2000] Garlan, D., Monroe, R. T., and Wile, D. (2000). Acme: Architectural description of component-based systems. In Leavens, G. T. and Sitaraman, M., editors, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press. (Cited on page 87)

[Garlan and Perry, 1995] Garlan, D. and Perry, D. E. (1995). Introduction to the special issue on software architecture. *Transactions in Software Engineering*, 21(4):269–274. (Cited on page 84)

[Garlan et al., 2001] Garlan, D., Schmerl, B., and Chang, J. (2001). Using gauges for architecture-based monitoring and adaptation. In *Proceedings of the 2001 Working Conference on Complex and Dynamic Systems Architecture*. (Cited on page 3)

[Ghosh et al., 2007] Ghosh, D., Sharman, R., Rao, H. R., and Upadhyaya, S. (2007). Self-healing systems - survey and synthesis. *Decision Support Systems in Emerging Economies*, 42(4):2164–2185. (Cited on pages 2 and 21)

[González-Sanchez et al., 2011a] González-Sanchez, A., Abreu, R., Gross, H., and Van Gemund, A. J. C. (2011a). Prioritizing tests for fault localization through ambiguity group reduction. In *Proceedings of the 26^{th} International Conference on Automated Software Engineering*, ASE'11, pages 83–92. (Cited on page 38)

[González-Sanchez et al., 2011b] González-Sanchez, A., Piel, E., Abreu, R., Gross, H.-G., and Van Gemund, A. J. C. (2011b). Prioritizing tests for software fault localization. *Software: Practice and Experience*, 41(10):1105–1129. (Cited on page 84)

[Gouveia et al., 2013] Gouveia, C., Campos, J., and Abreu, R. (2013). Using *HTML5* visualizations in software fault localization. In 1^{st} *Working Conference on Software Visualization*, VISSOFT, pages 1–10. (Cited on page 92)

[Graham et al., 2006] Graham, D., van Veenendaal, E., Evans, I., and Black, R. (2006). *Foundations of Software Testing: ISTQB Certification*. Cengage Learning Business Press. (Cited on page 85)

[Graham et al., 1982] Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: a call graph execution profiler. *ACM SIGPLAN Notices*, 17:120–126. (Cited on page 86)

[Greiner et al., 1989] Greiner, R., Smith, B. A., and Wilkerson, R. W. (1989). A correction to the algorithm in Reiter's theory of diagnosis. *Artificial Intelligence*, 41(1):79–88. (Cited on page 91)

[Hailpern and Santhanam, 2002] Hailpern, B. and Santhanam, P. (2002). Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12. (Cited on pages 1 and 2)

[Hangal and Lam, 2002] Hangal, S. and Lam, M. S. (2002). Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th international conference on Software engineering*, pages 291–301. (Cited on page 83)

[Harrold et al., 1998] Harrold, M. J., Rothermel, G., Wu, R., and Yi, L. (1998). An empirical investigation of program spectra. In *Proceedings of the 1998 Workshop on Program Analysis for Software Tools and Engineering*, PASTE'98, pages 83–90. (Cited on pages 9 and 10)

[Hermans et al., 2012a] Hermans, F., Pinzger, M., and van Deursen, A. (2012a). Detecting and visualizing inter-worksheet smells in spreadsheets. In *Proceedings of the 34^{th} International Conference on Software Engineering*, ICSE'12, pages 441–451. (Cited on page 90)

[Hermans et al., 2012b] Hermans, F., Pinzger, M., and van Deursen, A. (2012b). Detecting code smells in spreadsheet formulas. In *Proceedings of the 28^{th} International Conference on Software Maintenance*, ICSM'12, pages 409–418. (Cited on page 90)

[Hofer et al., 2013] Hofer, B., Riboira, A., Wotawa, F., Abreu, R., and Getzner, E. (2013). On the empirical evaluation of fault localization techniques for spreadsheets. In *Fundamental Approaches to Software Engineering*, volume 7793 of *Lecture Notes in Computer Science*, pages 68–82. Springer-Verlag. (Cited on page 90)

[Hofer and Wotawa, 2012] Hofer, B. and Wotawa, F. (2012). Spectrum enhanced dynamic slicing for better fault localization. In *Proceedings of the 20^{th} European Conference on Artificial Intelligence*, ECAI'12, pages 420–425. (Cited on page 89)

[Hollingsworth et al., 1997] Hollingsworth, J. K., Niam, O., Miller, B. P., Xu, Z., Gonçalves, M. J., and Zheng, L. (1997). Mdl: a language and compiler for dynamic program instrumentation. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT'97, pages 201–212. (Cited on page 85)

[Hong et al., 2015] Hong, D.-Y., Hsu, C.-C., Chou, C.-Y., Hsu, W.-C., Liu, P., and Wu, J.-J. (2015). Optimizing control transfer and memory virtualization in full system emulators. *Transactions on Architecture and Code Optimization*, 12(4):47. (Cited on page 84)

[Horn, 2001] Horn, P. (2001). Autonomic computing: *IBM*'s perspective on the state of information technology. (Cited on pages 1 and 9)

[Huang et al., 2010] Huang, C.-Y., Chang, J.-R., and Chang, Y.-H. (2010). Design and analysis of gui test-case prioritization using weight-based methods. *Journal of Systems and Software*, 83:646–659. (Cited on page 84)

[Huang et al., 1994] Huang, W.-C., Kao, C.-Y., and Horng, J.-T. (1994). A genetic algorithm approach for set covering problems. In *Proceedings of the 1994 International Conference on Evolutionary Computation*. (Cited on page 91)

[Huizinga and Kolawa, 2007] Huizinga, D. and Kolawa, A. (2007). *Automated Defect Prevention: Best Practices in Software Management*. Wiley. (Cited on page 83)

[Janssen et al., 2009] Janssen, T., Abreu, R., and Van Gemund, A. J. C. (2009). Zoltar: A toolset for automatic fault localization. In *Proceedings of the 2009 International Conference on Automated Software Engineering*, ASE'09, pages 662–664. (Cited on pages 90 and 92)

[Johnson and Wirthlin, 2010] Johnson, J. M. and Wirthlin, M. J. (2010). Voter insertion algorithms for fpga designs using triple modular redundancy. In *Proceedings of the 18^{th} International Symposium on Field Programmable Gate Arrays*, FPGA'10, pages 249–258. (Cited on page 83)

[Jones, 2004] Jones, J. A. (2004). Fault localization using visualization of test information. In *Proceedings of the 26^{th} International Conference on Software Engineering*, ICSE '04, pages 54–56. (Cited on page 91)

[Jones and Harrold, 2005] Jones, J. A. and Harrold, M. J. (2005). Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20^{th} International Conference on Automated Software Engineering*, ASE'05, pages 273–282. (Cited on pages 89 and 92)

[Jones et al., 2002] Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24^{th} International Conference on Software Engineering*, ICSE '02, pages 467–477. *ACM*. (Cited on page 92)

[Kasick et al., 2010] Kasick, M. P., Tan, J., Gandhi, R., and Narasimhan, P. (2010). Black-box problem diagnosis in parallel file systems. In *Proceedings of the 8^{th} Conference on File and Storage Technologies*, FAST, pages 43–56. (Cited on pages 3 and 6)

[Kephart et al., 2007] Kephart, J., Kephart, J., Chess, D., Boutilier, C., Das, R., Kephart, J. O., and Walsh, W. E. (2007). An architectural blueprint for autonomic computing. *Internet Computing*, 18(21). (Cited on pages 2 and 3)

[Kephart and Chess, 2003] Kephart, J. O. and Chess, D. M. (2003). The vision of autonomic computing. *Computer*, 36(1):41–50. (Cited on page 3)

[Kiriansky et al., 2002] Kiriansky, V., Bruening, D., Amarasinghe, S. P., et al. (2002). Secure execution via program shepherding. In *USENIX Security Symposium*, volume 92. (Cited on page 84)

[De Kleer, 2009] De Kleer, J. (2009). Diagnosing multiple persistent and intermittent faults. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI'09, pages 733–738. (Cited on pages 9, 17 and 21)

[De Kleer and Williams, 1987] De Kleer, J. and Williams, B. C. (1987). Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130. (Cited on pages 3, 4 and 7)

[De Kleer and Williams, 1992] De Kleer, J. and Williams, B. C. (1992). Readings in model-based diagnosis. In *Readings in Model-Based Diagnosis*, pages 100–117. Morgan Kaufmann Publishers Inc. (Cited on pages 4 and 13)

[Ko and Myers, 2008] Ko, A. J. and Myers, B. A. (2008). Debugging reinvented: Asking and answering why and why not questions about program behavior. In *Proceedings of the 2008 International Conference on Software Engineering*, ICSE'08, pages 301–310. (Cited on page 2)

[Kolettis and Fulton, 1995] Kolettis, N. and Fulton, N. D. (1995). Software rejuvenation: Analysis, module and applications. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, FTCS'95, pages 381–390, Washington, DC, USA. (Cited on page 89)

[Kumar et al., 2005] Kumar, N., Childers, B. R., and Soffa, M. L. (2005). Low overhead program monitoring and profiling. In *Software Engineering Notes*, volume 31, pages 28–34. (Cited on page 84)

[Laddaga et al., 2003] Laddaga, R., Robertson, P., and Shrobe, H. (2003). Introduction to self-adaptive software: Applications. In *Self-adaptive software: applications*, pages 1–5. Springer. (Cited on page 2)

[Larus and Schnarr, 1995] Larus, J. R. and Schnarr, E. (1995). Eel: Machine-independent executable editing. *ACM SIGPLAN Notices*, 30:291–300. (Cited on page 85)

[Lattner, 2002] Lattner, C. (2002). Llvm: An infrastructure for multi-stage optimization. Master's thesis, Computer Science Dept., University of Illinois at Urbana-Champaign, Urbana, IL. (Cited on page 85)

[Li and Yunfei, 2002] Li, L. and Yunfei, J. (2002). Computing minimal hitting sets with genetic algorithm. Technical report, DTIC Document. (Cited on page 91)

[Li and Vitányi, 2013] Li, M. and Vitányi, P. (2013). *An Introduction to Kolmogorov Complexity and Its Applications*. Springer Science & Business Media. (Cited on page 90)

[Linger et al., 1998] Linger, R. C., Mead, N. R., and Lipson, H. F. (1998). Requirements definition for survivable network systems. In *Proceedings of the 3rd International Conference on Requirements Engineering*, ICRE'98, pages 14–23. (Cited on page 2)

[Luk et al., 2005a] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005a). Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200. (Cited on page 85)

[Luk et al., 2005b] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005b). Pin: Building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices*, 40(6):190–200. (Cited on page 85)

[Lyle and Weiser, 1987] Lyle, J. and Weiser, M. (1987). Automatic program bug location by program slicing. In *Proceedings of the 1987 International Conference on Computers and Applications*, ICCA'87, pages 877–883. (Cited on page 89)

[Lyons and Vanderkulk, 1962] Lyons, R. E. and Vanderkulk, W. (1962). The use of triple-modular redundancy to improve computer reliability. *Journal of Research and Development*, 6(2):200–209. (Cited on page 83)

[Machado et al., 2013] Machado, P., Campos, J., and Abreu, R. (2013). Mzoltar: Automatic debugging of android applications. In *Proceedings of the 2013 International Workshop on Software Development Lifecycle for Mobile*, DeMobile'13, pages 9–16. (Cited on page 91)

[Mayer and Stumptner, 2003] Mayer, W. and Stumptner, M. (2003). Model-based debugging using multiple abstract models. In *Proceedings of the 2003 International Workshop on Automated and Analysis-Driven Debugging*, AADEBUG'03, pages 55–70. (Cited on pages 3 and 4)

[Memon et al., 2001] Memon, A. M., Soffa, M. L., and Pollack, M. E. (2001). Coverage criteria for gui testing. In *Proceedings of the 8^{th} European Software Engineering Conference*, ESEC'01/FSE'01, pages 256–267. (Cited on page 86)

[Miller and Maloney, 1963] Miller, J. C. and Maloney, C. J. (1963). Systematic mistake analysis of digital computer programs. *Communications of the ACM*, 6(2):58–63. (Cited on page 85)

[Misherghi and Su, 2006] Misherghi, G. and Su, Z. (2006). HDD: Hierarchical Delta Debugging. In *Proceedings of the 2006 International Conference on Software Engineering*, ICSE'06, pages 142–151. (Cited on page 88)

[Misurda et al., 2005] Misurda, J., Clause, J., Reed, J. L., Childers, B. R., Soffa, M. L., et al. (2005). Demand-driven structural testing with dynamic instrumentation. In *Proceedings of 27^th International Conference on Software Engineering*, ICSE'05, pages 156–165. (Cited on page 84)

[Mohammadi and Hashtrudi-Zad, 2007] Mohammadi, R. and Hashtrudi-Zad, S. (2007). A recursive algorithm for diagnosis in hierarchical finite-state machines. In *Proceedings of the 2007 International Conference on Systems, Man and*

Cybernetics, SMC'07, pages 1345–1350. (Cited on page 3)

[Mondal et al., 2015] Mondal, D., Hemmati, H., and Durocher, S. (2015). Exploring test suite diversification and code coverage in multi-objective test case selection. In 8^{th} *International Conference on Software Testing, Verification and Validation*, ICST15, pages 1–10. (Cited on page 84)

[Nethercote and Seward, 2003] Nethercote, N. and Seward, J. (2003). Valgrind: a program supervision framework. In *Proceedings of the* 3^{rd} *Workshop on Runtime Verification*, RV'03. (Cited on pages 85 and 86)

[Opitz and Maclin, 1999] Opitz, D. and Maclin, R. (1999). Popular ensemble methods: An empirical study. *Journal of Artificial Intelligence Research*, 11:169–198. (Cited on page 100)

[Orso et al., 2004] Orso, A., Jones, J. A., Harrold, M. J., and Stasko, J. (2004). Gammatella: Visualization of program-execution data for deployed software. In *Proceedings of the* 26^{th} *International Conference on Software Engineering*, ICSE '04, pages 699–700. (Cited on page 92)

[Parzen, 1962] Parzen, E. (1962). On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076. (Cited on page 72)

[Passos et al., 2014] Passos, L. S., Abreu, R., and Rossetti, R. J. F. (2014). Sensitivity analysis of spectrum-based fault localisation for multi-agent systems. In *Proceedings of the* 25^{th} *International Workshop on Principles of Diagnosis*, DX'14. (Cited on page 91)

[Passos et al., 2015] Passos, L. S., Abreu, R., and Rossetti, R. J. F. (2015). Spectrum-based fault localisation for multi-agent systems. In *Proceedings of the* 24^{th} *International Joint Conference on Artificial Intelligence*, IJCAI'15, pages 1134–1140. (Cited on page 91)

[Perez, 2012] Perez, A. (2012). *Dynamic Code Coverage with Progressive Detail Levels*. MSc Thesis, Faculdade de Engenharia da Universidade do Porto. (Cited on pages 85 and 86)

[Perez and Abreu, 2014] Perez, A. and Abreu, R. (2014). A diagnosis-based approach to software comprehension. In *Proceedings of the* 22^{nd} *International Conference on Program Comprehension*, pages 37–47. (Cited on page 91)

[Perez et al., 2014] Perez, A., Abreu, R., and Wong, E. (2014). A survey on fault localization techniques. (Cited on pages iv and 88)

[Perez et al., 2012] Perez, A., Riboira, A., and Abreu, R. (2012). A topology-based model for estimating the diagnostic efficiency of statistics-based approaches. In *Proceedings of the 2012 International Workshop on Program Debugging*, IWPD'12, pages 171–176. (Cited on page 86)

[Piel et al., 2012] Piel, E., Gonzalez-Sanchez, A., Gross, H.-G., Van Gemund, A. J. C., and Abreu, R. (2012). Online spectrum-based fault localization for health monitoring and fault recovery of self-adaptive systems. In *Proceedings of the* 8^{th} *International Conference on Autonomic and Autonomous Systems*, ICAS'11, pages 64–73. (Cited on page 87)

[Pierce, 1965] Pierce, W. (1965). *Failure-tolerant Computer Design*. Academic Press. (Cited on page 2)

[Pietersma et al., 2006] Pietersma, J., Feldman, A., and Van Gemund, A. J. (2006). Modeling and compilation aspects of fault diagnosis complexity. In *AUTOTESTCON*, pages 502–508. (Cited on page 9)

[Pinto et al., 2015] Pinto, P., Abreu, R., and Cardoso, J. M. P. (2015). Fault detection in C programs using monitoring of range values: Preliminary results. *CoRR*. (Cited on page 83)

[Polsson, 2015] Polsson, K. (2015). Chronology of personal computers. `http://pctimeline.info/`. Accessed October 2015. (Cited on page 1)

[Psaier and Dustdar, 2011] Psaier, H. and Dustdar, S. (2011). A survey on self-healing systems: Approaches and systems. *Computing*, 91(1):43–73. (Cited on pages iii and 2)

[Racunas et al., 2007] Racunas, P., Constantinides, K., Manne, S., and Mukherjee, S. S. (2007). Perturbation-based fault screening. In *Proceedings of th* 13^{th} *International Symposium on High Performance Computer Architecture*, HPCA'07, pages 169–180. (Cited on page 83)

[Reiter, 1987] Reiter, R. (1987). A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95. (Cited on pages 3, 4, 7, 12, 13 and 91)

[Riboira et al., 2011] Riboira, A., Abreu, R., and Rodrigues, R. (2011). An opengl-based eclipse plug-in for visual debugging. In *Proceedings of the* 1^{st} *Workshop on Developing Tools as Plug-ins*, TOPI'11, pages 60–60, New York, NY, USA. (Cited on page 92)

[Richard A. DeMillo,Richard J. Lipton, 1978] Richard A. DeMillo,Richard J. Lipton, F. G. S. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41. (Cited on page 84)

[Rosenblatt, 1956] Rosenblatt, M. (1956). Remarks on some nonparametric estimates of a density function. *The Annals of Mathematical Statistics*, 27(3):832–837. (Cited on page 72)

[Rosenblum, 1995] Rosenblum, D. S. (1995). A practical approach to programming with assertions. *Transactions on Software Engineering*, 21(1):19–31. (Cited on page 82)

[Rothermel et al., 1998] Rothermel, G., Harrold, M. J., Ostrin, J., and Hong, C. (1998). An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 34–43. (Cited on page 84)

[Rothermel et al., 1999] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (1999). Test case prioritization: An empirical study. In *Proceedings of the 1999 International Conference on Software Maintenance*, ICSM'99, pages 179–188. (Cited on page 84)

[Rothermel et al., 2001] Rothermel, G., Untch, R. H., Chu, C., and Harrold, M. J. (2001). Prioritizing test cases for regression testing. *Transactions on Software Engineering*, 27(10):929–948. (Cited on page 84)

[Salehie and Tahvildari, 2005] Salehie, M. and Tahvildari, L. (2005). Autonomic computing: Emerging trends and open problems. *Software Engineering Notes*, 30(4):1–7. (Cited on pages 1 and 7)

[Santelices et al., 2009] Santelices, R. A., Jones, J. A., Yu, Y., and Harrold, M. J. (2009). Lightweight fault-localization using multiple coverage types. In *Proceedings of the 31^{st} International Conference on Software Engineering*, ICSE'09, pages 56–66. (Cited on pages 4 and 10)

[Scott and Davidson, 2002] Scott, K. and Davidson, J. (2002). Safe virtual execution using software dynamic translation. In *Proceedings of the 18^{th} Annual Computer Security Applications Conference*, ACSAC'02, pages 209–218. (Cited on page 84)

[Scott et al., 2003] Scott, K., Kumar, N., Velusamy, S., Childers, B., Davidson, J., and Soffa, M. L. (2003). Reconfigurable and retargetable software dynamic translation. In *Proceedings of the 1^{st} Conference on Code Generation and Optimization*, CGO'03, pages 36–47. (Cited on page 84)

[Shvachko et al., 2010] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *Proceedings of the 26^{th} Symposium on Mass Storage Systems and Technologies*, MSST'10, pages 1–10. (Cited on pages 3 and 6)

[Silverman, 1986] Silverman, B. W. (1986). *Density Estimation for Statistics and Data Analysis*. Chapman and Hall. (Cited on page 72)

[Song et al., 2008] Song, D., Brumley, D., Yin, H., Caballero, J., Jager, I., Kang, M. G., Liang, Z., Newsome, J., Poosankam, P., and Saxena, P. (2008). Bitblaze: a new approach to computer security via binary analysis. In *Information systems security*, pages 1–25. Springer. (Cited on page 84)

[Srivastava and Eustace, 1994] Srivastava, A. and Eustace, A. (1994). *ATOM: a System for Building Customized Program Analysis Tools*, volume 29. *ACM*. (Cited on page 85)

[Stallman et al., 2002] Stallman, R., Pesch, R., and Shebs, S. (2002). *Debugging with GDB: the GNU Source-Level Debugger*. A GNU Manual. GNU Press. (Cited on page 82)

[Steimann et al., 2013] Steimann, F., Frenkel, M., and Abreu, R. (2013). Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA'2013, pages 314–324. (Cited on page 65)

[Sun et al., 2013] Sun, W., Gao, Z., Yang, W., Fang, C., and Chen, Z. (2013). Multi-objective test case prioritization for gui applications. In *Proceedings of the 28^{th} Symposium on Applied Computing*, SAC '13, pages 1074–1079. (Cited on page 84)

[Tan et al., 2010] Tan, J., Pan, X., Marinelli, E., Kavulya, S., Gandhi, R., and Narasimhan, P. (2010). Kahuna: Problem diagnosis for mapreduce-based cloud computing environments. In *Proceedings of the 12^{th} Network Operations and Management Symposium*, NOMS, pages 112–119. (Cited on pages 3 and 6)

[Tikir and Hollingsworth, 2002] Tikir, M. M. and Hollingsworth, J. K. (2002). Efficient instrumentation for code coverage testing. In *Proceedings of the 2002 International Symposium on Software Testing and Analysis*, ISSTA'02, pages 86–96. (Cited on pages iv and 85)

[e Silva MeyerDada al., 2004] , Silva MeyerDada., Garcia, A. A. F., de Souza, A. P., and De Souza, C. L. (2004). Comparison of similarity coefficients used for cluster analysis with dominant markers in maize. *Genetics and Molecular Biology*, 27:83–91. (Cited on pages 38 and 90)

[Vinterbo and Øhrn, 2000] Vinterbo, S. A. and Øhrn, A. (2000). Minimal approximate hitting sets and rule templates. *International Journal of Approximate Reasoning*, 25(2):123–143. (Cited on page 91)

[Vokolos and Frankl, 1998] Vokolos, F. I. and Frankl, P. G. (1998). Empirical evaluation of the textual differencing regression testing technique. In *Proceedings of the 1998 International Conference on Software Maintenance*, ICSM'98, pages 44–53, Washington, DC, USA. (Cited on page 83)

[Weiser, 1982] Weiser, M. (1982). Programmers use slices when debugging. *Communications ACM*, 25(7):446–452. (Cited on page 89)

[Weiser, 1984] Weiser, M. (1984). Program slicing. *Transactions on Software Engineering*, SE-10(4):352–357. (Cited on page 89)

[Witchel and Rosenblum, 1996] Witchel, E. and Rosenblum, M. (1996). Embra: Fast and flexible machine simulation. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 68–79. (Cited on page 84)

[Wong and Qi, 2006] Wong, W. E. and Qi, Y. (2006). Effective program debugging based on execution slices and inter-block data dependency. *Journal of Systems and Software*, 79(7):891–903. (Cited on page 89)

[Wotawa, 2001] Wotawa, F. (2001). A variant of reiter's hitting-set algorithm. *Information Processing Letters*, 79(1):45–51. (Cited on pages 13 and 91)

[Wotawa, 2010] Wotawa, F. (2010). Fault localization based on dynamic slicing and hitting-set computation. In *Proceedings of the* 10^{th} *International Conference on Quality Software*, QSIC'10, pages 161–170. (Cited on pages 89 and 90)

[Wotawa et al., 2002] Wotawa, F., Stumptner, M., and Mayer, W. (2002). Model-based debugging or how to diagnose programs automatically. In *Proceedings of the 2002 International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems*, IAE/AIE'02, pages 746–757. (Cited on pages 3 and 4)

[Xie et al., 2013] Xie, X., Wong, W. E., Chen, T. Y., and Xu, B. (2013). Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879. (Cited on page 89)

[Xie et al., 2010] Xie, X., Wong, W. E., Chen, T. Y., Xu, B., et al. (2010). Spectrum-based fault localization without test oracles. In *Proceedings of the* 11^{th} *International Conference on Quality Software*, QSIC,11, pages 1–10. (Cited on page 89)

[Yang et al., 2006] Yang, Q., Li, J. J., and Weiss, D. (2006). A survey of coverage based testing tools. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 99–103. (Cited on page 85)

[Yilmaz et al., 2008] Yilmaz, C., Paradkar, A., and Williams, C. (2008). Time will tell: Fault localization using time spectra. In *Proceedings of the 30th international conference on Software engineering*, pages 81–90. (Cited on page 10)

[Zadeh, 1965] Zadeh, L. A. (1965). Fuzzy sets. *Information and Control*, 8(3):338–353. (Cited on page 58)

[Zadeh, 1968] Zadeh, L. A. (1968). Probability measures of fuzzy events. *Journal of Mathematical Analysis and Applications*, 23(2):421–427. (Cited on page 60)

[Zeller, 2002] Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 2010 Symposium on Foundations of Software Engineering*, ESEC'10/FSE'10, pages 1–10. (Cited on page 88)

[Zeller and Hildebrandt, 2002] Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *Transactions on Software Engineering*, 28(2):183–200. (Cited on pages iv and 88)

[Zhang and Gupta, 2004] Zhang, X. and Gupta, R. (2004). Cost effective dynamic program slicing. *ACM SIGPLAN Notices*, 39(6):94–106. (Cited on page 89)

[Zhang et al., 2005] Zhang, X., He, H., Gupta, N., and Gupta, R. (2005). Experimental evaluation of using dynamic slices for fault location. In *Proceedings of the 2005 International Workshop on Automated and Analysis-Driven Debugging*, AADEBUG'05, pages 33–42. (Cited on page 89)

[Zhang et al., 2007] Zhang, X., Tallam, S., Gupta, N., and Gupta, R. (2007). Towards locating execution omission errors. In *Proceedings of the 2007 Conference on Programming Language Design and Implementation*, PLDI'07, pages 415–424. (Cited on page 89)

[Zhao and Ouyang, 2007] Zhao, X. and Ouyang, D. (2007). Improved algorithms for deriving all minimal conflict sets in model-based diagnosis. In *International Conference on Intelligent Computing*, ICIC'07, pages 157–166. (Cited on page 91)

[Zoeteweij et al., 2007] Zoeteweij, P., Abreu, R., Golsteijn, R., and Gemund, A. J. V. (2007). Diagnosis of embedded software using program spectra. In *Proceedings of the 14^{th} International Conference and Workshops on the Engineering of Computer-Based Systems*, ECBS'07, pages 213–220. (Cited on page 91)

```
----[---->+<]>++.[--->+<]>----.+++++++++++.+++[->+++<]>++.+++.+.--.+++
+++.++++.--------.+++++++++.++++++.-----.++++.[--->+<]----+++++++++>>>
```

Agradecimentos

E cá está... o resultado de quase 5 anos da minha vida.

Embora possa parecer tudo muito mecânico e trivial, a realidade é que a formalidade da escrita oculta toda a complexidade da experiência vivida. Houve momentos de imensa felicidade e momentos de enorme desespero. Gosto de fazer a comparação do processo com o Paradoxo de Zeno:

> *"Suppose Homer wishes to walk to the end of a path. Before he can get there, he must get halfway there. Before he can get halfway there, he must get a quarter of the way there. Before traveling a quarter, he must travel one-eighth; before an eighth, one-sixteenth; and so on. (...) This description requires one to complete an infinite number of tasks, which Zeno maintains as an impossibility."*
>
> – Wikipedia[1]

Estabelecendo o paralelismo, inicialmente (*i.e.*, imediatamente após a depressão de se perceber que não se capta nada do assunto), com o entusiasmo e energia de quem começa, a tarefa era agradável e grandes progressos foram feitos. No entanto, à medida que o tempo foi passando e a energia se foi esgotando, a tarefa tornou-se cada vez mais penosa e, apesar de, na prática, o objetivo final estar cada vez mais próximo, a minha perceção era de que eu não iria conseguir. O fim de cada tarefa dava origem ao início de outra, a qual era desempenhada com menor eficiência que a tarefa antecedente. Apesar de estar extremamente grato pela experiência, reconheço que foi bastante mais difícil do que eu esperava. Tenho a certeza de que a concretização desta tarefa só foi possível por ter tido ao meu lado as pessoas a quem passo a agradecer.

Gostaria de começar por agradecer à minha melhor amiga e amor da minha vida, Lígia "Maria" Massena, por ter estado ao meu lado e me ter apoiado em todos os momentos deste desafio.

[1] À qual gostaria também de expressar o meu sincero agradecimento.

Ao meu amigo e orientador, Rui “Grande Líder Kim Jong-Rui” Abreu, por acompanhar de perto todo este percurso e ter a paciência necessária para lidar comigo. Fico extremamente grato por ter tido um “chefe” que me tenha dado não só a liberdade mas também o incentivo para explorar todos os meus hobbies.

Aos meus ídolos, o meu Avô Licínio, o meu amigo de longa data Nuno Cunha, e o meu grande amigo Vítor Almeida. As suas experiências de vida foram e continuam a ser uma fonte de inspiração para mim. Em particular, o meu Avô inspira-me pela sua determinação e tenacidade, o Nuno pelo seu sentido de justiça, integridade e conhecimento enciclopédico, e o Vítor pela sua generosidade e incrível engenho.

À minha sogra, Iolanda “Sogra” Macena, que é mais do que uma mãe para mim e com quem partilho uma profunda amizade, pelo seu apoio incondicional.

Ao meu sogro, Horácio “Macho” Massena, que apesar do início de relação “tenso”, é uma pessoa que atualmente me trata como filho e tem um orgulho genuíno dos meus feitos.

À minha mãe e à minha Avó Olivinha, pelo apoio dado durante estes anos.

À minha “avógra”, Olga “Companheira” Chicket, pelo seu papel instrumental numa fase de grande adversidade.

Aos padrinhos, Mário “Padrinho” Tavares e Conceição “Madrinha” Tavares, pelo carinho e por marcarem presença em momentos significativos, ainda que pudessem ser de pouco interesse para eles (*e.g.*, a defesa da minha tese).

Ao meu “bro”, André “Beat” Silva, que esteve presente na grande maioria dos momentos determinantes do meu doutoramento. Foi um elemento estruturante neste processo e a quem devo uma grande parte dos meus sucessos. Partilhámos uma enorme diversidade de vivências, desde momentos de alegria extrema (*e.g.*, “quod erat demonstrandum”, “colar o pistão”, *etc.*), a momentos de puro pânico (*e.g.*, o episódio do gringo possuído, o evento pós-telepizza, *etc.*). Num aspeto mais mundano, e entre muitas outras coisas, o Beat inspirou-me a construir uma bateria e a aprender a tocá-la, coisa que atualmente contribui bastante para a minha felicidade.

Ao meu companheiro de final de corrida, Lúcio “Primaço” Passos, pelo seu apoio, motivação e inesgotável boa disposição. Foi bom ter alguém igualmente miserável para poder perceber o meu ponto de vista.

Ao meu amigo de sempre, João Pedro “Musty” Dias, por todo o apoio dado e por me tentar ensinar, ainda que com sucesso limitado, a importância de relaxar: “O Musty de amanhã pensa no assunto...”.

Ao José Luís “Oh, Cala-te!” Pereira e à Neuza “Oh, Cala-te!” Florim, um casal com quem, de forma rápida e inesperada, criei uma forte amizade. Apesar da distância, estão lá no fundo do meu coração robótico.

As minhas discussões com o Musty e com o Zé e a Neuza, por estarem sempre assentes em formas muito diferentes de ver o mundo[2] obrigaram-me a reconsiderar muitas “verdades absolutas”. Em retrospetiva, sinto que o facto de atualmente considerar a existência de um 0.5 se deve em grande parte a eles.

Ao Márcio “Márcinho” Sá, companheiro de bricolage, por estar sempre disponível para me ajudar, ao ponto de se matar (quase literalmente) a trabalhar.

Ao Elói “Braahh” Barros e Nuno “Té Looogooo...” Gaspar pelo apoio, boa disposição e prontidão para o que desse e viesse. Té looogooooo!!!

À Raquel Almeida, com quem vivi uma experiência que me ficará marcada para o resto da vida, e no processo me ensinou a reconhecer o valor dos amigos.

À Idalina “Lina” Silva, pelos seus valiosos conselhos e por me ter facilitado imensamente a vida ao resolver-me prontamente problemas que estavam para alem das suas obrigações (*e.g.*, reservar/fazer check-in em voos para conferências, enviar-me documentos para a FCT, *etc.*).

To Luuk Eliens, a guy with whom I immediately felt connected to and had the pleasure to work with. I truly admire his determination when pursuing a goal. I hope one day we can work together again.

Ao Vítor e Marly Pinto, pelo seu carinho e amizade e por me brindarem consistentemente com alegria e boa disposição.

Ao Mauari Quintero, à Katherine Velosa e à “Sammy”, uns amigos improváveis, que me deram todo o seu carinho e apoio.

Ao Vasco Mota e à Isabel Pinto, pelo companheirismo e experiências vividas em conjunto.

Ao Carlos “Carlinhos” Gouveia, que foi um gajo para quem eu trabalhei uma tarde e, em troca, recebi uma enorme consideração e reconhecimento que prevaleceram ao teste do tempo.

Às meninas do Subway, Maria Mendes, Sara Moreira, Marisa Soares, Ana Albergaria, e ao Rúben Quintal, pela boa disposição, simpatia e, sobretudo, pelas enumeras vezes que me alimentaram (com “extra-pimentos”) durante estes anos.

[2]A minha visão: 011001101001001100. A deles: arcos-íris e unicórnios (caricaturado).