University of California
Santa Barbara

Toward Autonomous Verification Systems

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Kuo-Kai Hsieh

Committee in charge:

Professor Li-C. Wang, Chair
Professor Malgorzata Marek-Sadowska
Professor Luke Theogarajan
Doctor Jayanta Bhadra

September 2018

The Dissertation of Kuo-Kai Hsieh is approved.

Professor Malgorzata Marek-Sadowska

Professor Luke Theogarajan

Doctor Jayanta Bhadra

Professor Li-C. Wang, Committee Chair

July 2018

Toward Autonomous Verification Systems

Kuo-Kai Hsieh

Acknowledgements

I express my deepest appreciation to my advisor, Professor Li-C. Wang, for his guidance over the past five years. It was him who not just let me realize the way to conduct research, but deeply influenced my philosophy of life. I loved the time we discussed research challenges as well as shared our life experiences.

I would like to thank my colleagues at NXP, especially Wen Chen, Monica Farkash and Jayanta Bhadra, for their mentoring throughout my internship. They helped me a lot in establishing my experimental environment, preparing for publications, and finding out projects that met both my research interests and the company's needs. My journey of research would be totally different without them.

I would like to acknowledge my committee members, Professor Malgorzata Marek-Sadowska and Professor Luke Theogarajan for their input and advice to my career development.

I also want to thank my lab members, Jay Shan, Matt Nero and Ahmed Wahba for their friendship and help when I was busy in research and TA. I felt comfortable working in our lab. I am fortunate to have my best friends, especially Fan Lin, Chieh-Chi Kao, Wei-Ting Lin, Chih-Cheng Chang and Yuxiang Wang, who took care of me, supported me when I felt down and shared our life together.

Last but not least, I want to thank my parents, my sister Yi-Kai and my wife Chia-Ling for their wholeheartedly support so I could dedicate myself to pursuing a doctoral degree in a foreign country.

Curriculum Vitæ

Kuo-Kai Hsieh

Education

2013 - 2018	Ph.D. in Electrical and Computer Engineering, University of California, Santa Barbara.
2013 - 2015	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2006 - 2010	B.S. in Electrical Engineering, National Taiwan University

Publications

1. **K.-K. Hsieh**, L.-C. Wang, W. Chen, and J. Bhadra, "Learning to produce direct tests for security verification using constrained process discovery," in *Proceedings of the 54th Annual Design Automation Conference, DAC 2017, Austin, TX, USA, June 18-22, 2017*, pp. 34:1–34:6, 2017.
2. **K.-K. Hsieh**, M. Farkash, W. Chen, L.-C. Wang, and J. Bhadra, "Automatic investigation of power inefficiencies," in *Proceedings of the Design and Verification Conference, DVCon 2017*, 2017.
3. **K.-K. Hsieh**, S. Siatkowski, L.-C. Wang, W. Chen, and J. Bhadra, "Feature extraction from design documents to enable rule learning for improving assertion coverage," in *22nd Asia and South Pacific Design Automation Conference, ASP-DAC 2017, Chiba, Japan, January 16-19, 2017*, pp. 51–56, 2017.
4. W. Chen, **K.-K. Hsieh**, L.-C. Wang, and J. Bhadra, "Data driven test plan augmentation for platform verification," *IEEE Design Test*, vol. PP, no. 99, pp. 1–1, 2017.
5. **K.-K. Hsieh**, W. Chen, L.-C. Wang, and J. Bhadra, "On application of data mining in functional debug," in *The IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2014, San Jose, CA, USA, November 3-6, 2014*, pp. 670–675, 2014.

Abstract

Toward Autonomous Verification Systems

by

Kuo-Kai Hsieh

Design verification has been a challenging problem due to the increasing complexity of modern system-on-chip (SoC) designs and it is considered one of the costliest processes in hardware design flow. This dissertation investigates a major labor-intensive task, generating tests targeting at a given coverage point, in simulation-based verification, and proposes an autonomous software system capable of completing the task. A key feature of the proposed system is its learning capability – it can learn from examples provided by human engineers to improve itself. There are three major components in the proposed system: test generation, a knowledge database, and rule learning algorithms. The proposed system is able to retrieve information from the database, use the information to analyze simulation results, and generate new tests based on the analysis. Several machine learning techniques are used in the proposed verification system. For test generation, a novel method, constrained process discovery, is used to learn a test case generation model from manually developed tests. The test case generation model can create new tests and increase its test generation capability by learning from tests developed by humans. For creating a knowledge database, text mining methods are used to extract important design features from design documents. Experiments showed that the extracted signals can be utilized as observation points to infer important hardware events. Last, a novel rule learning method, VeSC-CoL, is proposed to analyze simulation results. VeSC-CoL can handle extremely imbalanced data, which is common in verification, while traditional rule learning methods cannot.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This work investigates methods to construct a smart software system to deal with a verification task in hardware verification. The task to deal with is to create tests to hit a given coverage point. In the proposed system, learning from engineers is a critical and unique capability. In the ordinary process of software systems that deal with the same task, engineers need to intervene every time the systems have trouble processing a coverage point. After problems are solved, if engineers do not modify the systems, they will still fail to process the same or similar coverage points again, in other words, repeating its mistakes. However, the software system proposed in this study is able to learn from experience, which means that once engineers intervene and provide new tests that hit the coverage point, the system is expected to learn from the new tests and be able to solve similar or more difficult tasks in the future.

In addition, the software system proposed in this study is expected to function as a low-level human engineer who is able to ask for help "wisely. In the real world, when an inexperienced human engineer fails to deal with a task, normally he/she will reach to an experienced engineer for help via two kinds of questions as Figure 1.1 illustrated in the following. While the first approach is to simply report a failure and ask for the next

step, in the second approach, in addition to reporting a failure only, an inexperienced engineer would also provide his/her analysis of the task. As the experienced engineer in the first approach has to analyze the task from scratch on his own, the second approach is preferred since with the initial analysis in hand, the experienced engineer spends less time to deal with the task. Moreover, if an inexperienced engineer is capable of identifying which sub-task fails in the analysis, experienced engineers could solve the problem much more quickly. Hence, the proposed system is designed to function as an engineer who can ask for help using the second approach, saving experienced engineers in the real world time and energy during the intervention.

Figure 1.1: Two approaches that people seek for help from others

The proposed system uses the second approach – it can report failure and provide analysis. The original task is partitioned into two sub-tasks. While the proposed system is not able to create tests to hit a coverage point, it asks for help from human engineers as well as points out which sub-tasks fails. In addition, human engineers can query information in the failed sub-task to save analysis time.

Several machine learning techniques are used in the proposed system. There are three key challenges to construct the proposed system. The first challenge is test generation. The proposed system needs a method to create tests. Besides, the system must be able to learn from tests that provided by experienced human engineers. After learning, its test

generation capability increases and the system is able to create more complex tests. The second challenge is to identify design knowledge that is required to solve the task and formally represent such design knowledge in the system. The third challenge is about the information the system can provide to human while failing to deal with a coverage point. The information must be true, without any guessing, and helpful for experienced engineers.

Theoretically, when enough examples are provided, the proposed system can deal with every coverage points that experienced engineers can deal with. The system can learn from many human engineers and accumulate knowledge from all of them.

The rest of this dissertation is organized as follows. Chapter 2 describes the required background knowledge of this work. Chapter 3 discusses the problem, related works and the overview of the proposed verification system. Chapter 4 proposes a method combining constraint solving and process mining to deal with the challenge of test generation. Chapter 5 discusses the challenge of design knowledge and demonstrates a text mining method that can extract design signals from design documents. Chapter 6 explores rule learning methods to deal with extremely imbalanced data and to provide reasonable information to human engineers while not able to complete the task. Finally, Chapter 7 concludes this work.

Chapter 2

Background

2.1 Functional Verification

Functional verification is a chip design process in which the goal is to make sure the design implementation meets design specification. The design implementation is typically at Register-Transfer Level (RTL) and written in hardware description languages such as Verilog and VHDL. The process of functional verification usually starts from a verification plan, which specifies design features to verify and the criteria of successful verification of these features. Then verification engineers use various methods to execute the verification plan.

In general, there are two categories of methods in functional verification, simulation approaches, and formal approaches. The idea of simulation approaches is to have tests, or input stimuli, then simulate the tests with design implementation to check the design behavior. The main problem of simulation approaches is that we do not know if a design under verification is indeed bug-free. It is possible that a design has a bug but the bug is not activated in the simulation. On the other hand, the idea of formal approaches is mathematical proof. Formal approaches first convert a design into a mathematical model,

then mathematically proves some properties are correct. For example, a property can be "the value of signal A always equals to the value signal B." If the design specification can be converted to properties and all the properties are proved correct in the implementation, then it is guaranteed the implementation is bug-free.

2.1.1 Formal Approaches

To apply formal approaches, typically there are two steps. The first step is to convert a design into a mathematical model and the second step is to prove that a given property is correct in the converted model. Examples of formal approaches include explicit-state model checking [1], symbolic model checking [1], and symbolic simulation[2]. Although there are off-the-shelf tools that can handle the two steps, in practice, formal approaches are only utilized in limited verification tasks.

A critical problem of the applicability of formal approaches is scalability. It has been shown that the complexity of formal approaches is in general NP-hard. Hence to apply formal approaches in practice, a converted mathematical model must be simple or a property to be verified must be simple. When a property to be verified is not simple, people reduce a converted mathematical model by abstracting the functionality of a design under verification. This abstraction method introduces another problem – it is non-trivial to verify the correctness of the abstraction. Also, there is a high entry barrier of applying formal approaches because the abstraction is conducted manually and requires people having enough discipline about formal languages and a design under verification. Another critical problem is to have a set of properties that are equivalent to the design specification. Again, the equivalence is checked by human and there is no systematic way to verify that a set of properties is equivalent to the design specification. These are the reasons why formal approaches are not the mainstream approaches in

functional verification.

2.1.2 Simulation-based Approaches

In industry, simulation-based approaches are the primary methods in functional verification due to its scalability, especially when the complexity of a design under verification is high, e.g. a System-on-Chip (SoC). There are three standard components in simulation-based approaches and simulation environments – test generation, checkers, and coverage collection. Nowadays, simulation-based approaches are mature such that there is an industrial-standard simulation-based verification methodology, Universal Verification Methodology (UVM) [3]. Figure 2.1 illustrates an overview of an simulation environment and the three components are briefly described below.

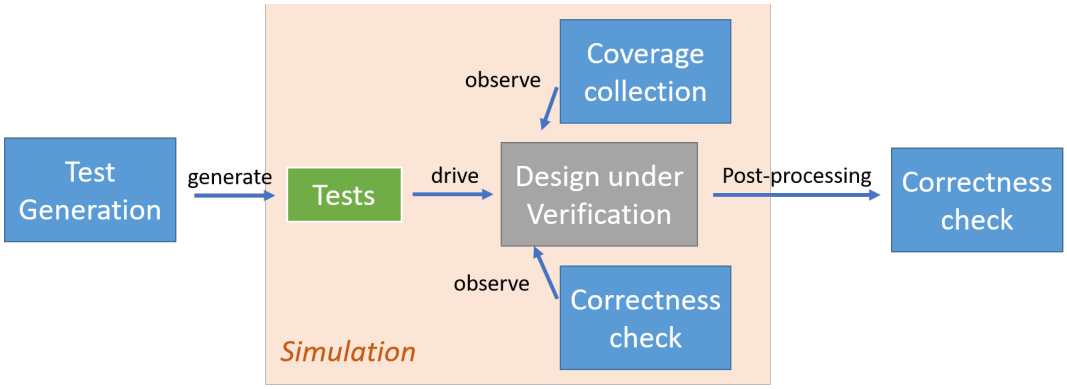

Figure 2.1: Overview of an simulation environment

The goal of test generation is to have tests for simulation. Tests can be produced manually by verification engineers, or generated by test generators. Modern test generators utilize constraint-random approaches. The idea of constraint-random approaches is to overcome problems in pure random test generation. For pure-random test generation, a generated tests can violate design assumptions and there is no way to have tests that can stress a specific design functionality. For constraint-random approaches, constraints

are used to limit the generated tests to the design assumptions as well as to target at certain design behaviors. Constraints solvers are used to produce tests that satisfy the input constraints [4]. Nowadays, test generators are manually designed by engineers and it requires deep knowledge about a design under verification to write test generators. In industry, there is an ongoing group, Portable Stimuli Working Group [5], that tries to come out a standard methodology for test generation.

The goal of checkers is to check the correctness of each simulation run. There are several types of checkers. Checkers can be implemented together in a test, which makes the test self-checked. Checkers can be implemented as a monitor of a design under verification and check the design behavior on-the-fly during simulation. Checkers can be implemented as a post-processing software that checks design correctness by analyzing simulation logs. Checkers are typically generated manually by designers and verification engineers.

The goal of coverage collection is to know what functionality and events are simulated during simulation. There are two categories of coverage, structural coverage ,and functional coverage [6]. Structural coverage is defined according to the structure of HDL code. For example, code coverage monitors whether each line of code is executed during simulation. Finite state machine coverage monitors whether each state and each transition is traversed in simulation. Functional coverage is defined by human and it can monitor specific and complex events. For example, a functional coverage point can monitor whether a specific waveform appears during simulation. Typically all test items in a test plan have corresponding coverage points.

When the checkers are properly implemented and the required coverage is properly defined, the goal of functional verification becomes hitting all the defined coverage points, which is the basic concept of Coverage-Driven Verification [6]. One key challenge of coverage-driven verification is to produce tests that can hit the defined coverage points.

This work proposes a verification system to deal with this challenge.

2.2 Rule Learning

Given a set of samples of different classes, rule learning algorithms aim at distinguishing samples from different classes. Samples are represented by features. There are two categories of rule learning algorithms, predictive rule learning, and descriptive rule learning [7]. The main difference between the two categories is that predictive rule learning methods aim at prediction, while descriptive rule learning methods focus on an explainable rule, e.g. an IF-ELSE relation, that can distinguish different classes. An explainable rule is easily understood by humans. For example, a decision-tree based algorithm can be a descriptive rule learning algorithm as an easily understandable rule can be extracted as described in the next paragraph. Random forest [8] is a predictive rule learning algorithm because there is no easily understandable rule can be converted from the model. The prediction result of random forest is based on voting among many decision trees. The voting part makes it not descriptive.

This work focuses on only descriptive rule learning methods on two classes and binary features. Two classes are used to represent whether a coverage point is hit or not. Only binary features are considered because pre-processing techniques can be used to convert numerical and categorical features to binary features and this can isolate problems in learning algorithms.

There are many off-the-shelf rule learning algorithms. Examples include subgroup discovery [9], CN2 [10], ID3 [11] and Classification And Regression Trees (CART) [12]. These methods, in general, can be viewed as algorithms related to decision trees but using different methods for splitting nodes and terminating tree growth. Hence it is worth to describe the method to extract a rule from a decision tree.

Figure 2.2 illustrates the method to extract a rule to a node in a decision tree. Each path starting from the root of a decision tree model corresponds to a rule. The rule is the conjunction of the decisions along the path. Note that a path does not necessarily end at a leaf. Suppose the blue node contains only samples in the target class, a rule can be extracted from the path starting from the root to the blue node, $f_A = false \wedge f_B = true$, where $f_A = false$ is the first decision and $f_B = true$ is the second decision.

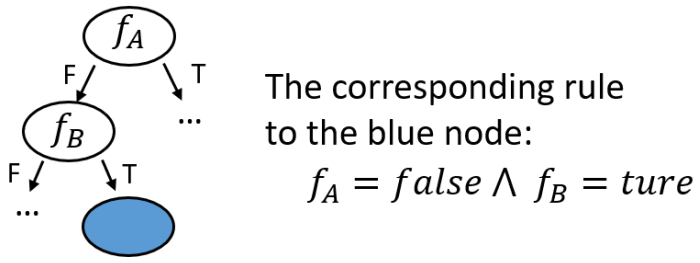

Figure 2.2: Rule extraction from a node in a decision tree

If samples of the target class are in multiple nodes, the extracted rule is simply the disjunction of rules extracted from each node having samples of the target class. An example is illustrated in Figure 2.3.

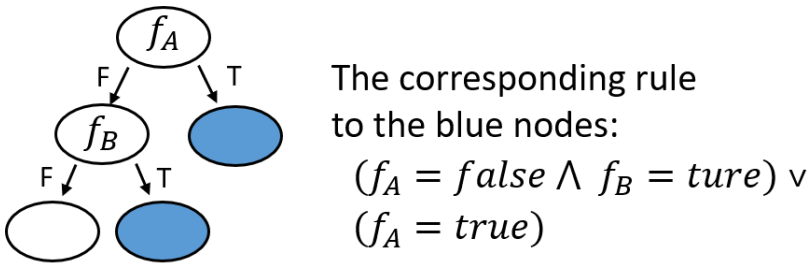

Figure 2.3: Rule extraction from a decision tree

2.3 Boolean Algebra

This section defines some terminologies of Boolean Algebra that will be used in this work. More details of Boolean Algebra can be found in textbooks such as [13].

- **Boolean Values**. Boolean values are *true* and *false*. Conventionally *true* can be written as 1, and *false* can be written as 0.

- **Boolean Variable**. x is a Boolean variable if it can represent only Boolean values.

- **Boolean Function**. f is a n-variable Boolean function if the domain of f is $\{0,1\}^n$ and the range of f is $\{0,1\}$ or its non-empty subset. Conventionally, f is written as $f(\mathbf{x})$ or $f(x_1, x_2, \ldots, x_n)$ where $\mathbf{x}$ is a n-dimensional Boolean vector and each x_i is a Boolean variable.

- **Basic Operations**.

  - AND (Conjunction), a two-variable function, denoted $x \wedge y$, $x * y$, or simply xy.

    $x \wedge y = 1$ if $x = 1$ and $y = 1$, otherwise $x \wedge y = 0$.

  - OR (Disjunction), a two-variable function, denoted $x \vee y$ or $x + y$.

    $x \wedge y = 1$ if $x = 1$ or $y = 1$, otherwise $x \wedge y = 0$.

  - NOT (Negation, Complement), a one-variable function, denoted $\neg x$ or x'

    $\neg x = 1$ if $x = 0$, otherwise $\neg x = 0$.

- **Literal**. A literal is a Boolean variable, e.g. x, or a Boolean variable with negation, e.g. $\neg x$.

- **Satisfiable Assignment**. Given a Boolean function f, $\mathbf{x}$ is a satistiable assignment of f if $f(\mathbf{x}) = 1$.

- **Clause**. A clause is a disjunction of one or more literals. For example, $x_1 + x_2 + x_3$.

- **Conjunction Normal Form (CNF)**. Conjunction Normal Form (CNF) is a conjunction of one or more clauses. It is also known as Product of Sum (PoS). CNF is

a Boolean function representation.

- **Term**. A term is a conjunction of one or more literals. For example, $x_1x_2x_3$.
- **Disjunctive Normal Form (DNF)**. Disjunctive Normal Form (DNF) is a disjunction of one or more terms. It is also known as Sum of Products (SoP). DNF is a Boolean function representation.
- **k-term DNF**. k-term DNF is a DNF with exactly k terms.

2.4 Boolean Satisfiability Problem

The Boolean satisfiability (SAT) problem is that given a Boolean function f represented in CNF, determine if there is an satisfiable assignment and find an satisfiable assignment if it exists. This problem is proved to be NP-complete [14]. Informally speaking, there is no efficient algorithm to deal with SAT and the runtime can be very long.

SAT is a fundamental problem of constraint solving problems. We can imagine that each clause in CNF is like a constraint to be satisfied. A clause evaluates to 1 if and only if it is satisfied. Then, if there is a satisfiable assignment of the CNF, all the clauses must evaluate to 1 with the assignment, which means there is a solution that satisfies all constraints. In theory, every constraint solving problem in discrete domain can be converted to a SAT problem.

Designing SAT solvers is an active research topic and there are annual SAT competitions [15] where people can find state-of-the-art SAT solvers and their performance comparisons. Usually, the SAT solvers in the competition are open-sourced and people can freely use them. This work only utilizes SAT solvers and does not discuss methods to solve the SAT problem.

2.4.1 SAT Encoding Techniques

SAT encoding, or converting a problem in discrete domain to a SAT problem, is not a trivial task. [16] discusses several techniques for SAT encoding, in which two most important techniques are *thinking in circuits* and *translating circuits to CNF*.

Thinking in circuits can be understood as representing a constraint in digital circuits. For example, given two $32-bit$ unsigned integers A and B, the CNF of $A > B$ can be converted from a comparator whose output is 1 if and only if the first input number is larger than the second input number. It A is connect to the first input and B is connected to the second input, we want a constraint that the output of the comparator is 1. The comparator circuit can be obtained by logic synthesis tools.

For *Translating circuits to CNF*, the key idea is variable substitution. Instead of representing the primary outputs of a circuit explicitly by the primary input variables, a variable is introduced for the output of each gate and some clauses are added to maintain the relation of variables. For example, for a 2-input AND gate $z = a \wedge b$, the relation of z, a, b can be maintained by clauses $(z \vee \neg a \vee \neg b) \wedge (\neg z \vee a) \wedge (\neg z \vee b)$. It is easy to verify by enumeration that the CNF is 1 if and only if $z = a \wedge b$.

Using the above techniques, we can convert every Boolean formula to a CNF and the size of formula grows polynomially. For example, to convert a DNF to a CNF, we can first convert a DNF into a two-level AND-OR circuit, then apply *Translating circuits to CNF* to get a CNF.

2.5 Binary Decision Diagram

Binary Decision Diagram (BDD) [17] is a canonical representation of Boolean functions. BDD is also a data structure for efficiently storing and manipulating Boolean functions [18]. It is generally assumed that BDD refers to reduced, ordered BDD. BDD

is widely used in software that requires handling Boolean functions such as symbolic simulation, symbolic model checking, equivalence check, representing a set and performing set operations, etc.

The basic idea of BDD is to represent a function in a graph model. Figure 2.4 illustrates the idea where left branches are *false branch* and right branches are *true branch*, represents its parent node is assigned false or true respectively. The represented function is $f = a'bc + ab$. To get the value of an assignment, just trace the tree down to the corresponding leaf. For example, given an assignment $(a, b, c) = (1, 1, 1)$, f evaluates to 1 from the formula. From the graph, $(a, b, c) = (1, 1, 1)$ leads to the right-most leaf, which is the same as function evaluation.

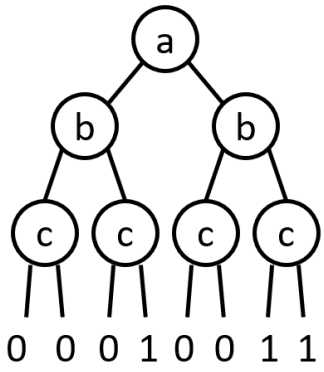

Figure 2.4: Idea of binary decision diagram

To make BDD manipulation efficient, it is required that BDD is reduced and ordered. Reduced refers to removing redundant nodes (a node is redundant if both its child nodes are the same) and sharing as many nodes as possible. Ordered refers to a fixed variable ordering in the graph. For example, variable a can appear only in level 1, variable b can appear only in level 2, etc. More implementation details can be found in [18].

Here uses two BDD operations to show the advantages of BDD and they can be found in [18]. The first operation is equivalence checking. Since BDD is a canonical

representation and it is reduced, equivalence checking of two BDDs takes constant time (just check whether they have the same root node). The second application is calculating the number of satisfiable assignments of a BDD. This task can be done based on a recursive call on the tree-like structure and the overall complexity is the number of nodes of the given BDD.

Chapter 3

Overview of The Proposed Method

3.1 Problem Overview

This work investigates a task in functional verification at the system-level that given a coverage point, produce a test that can hit the coverage point. A similar problem is Coverage-Directed Test Generation (CDTG), which is a methodology that aims at accelerating coverage closure by guiding test generation with coverage result analysis. The primary difference between this work and CDTG is that CDTG assumes test generation methods are available and fixed, while this work treats test generation as a system component to be developed. Since this work deals with verification at system-level, formal approaches do not work in practice and we focus on simulation-based approaches.

In general, the flow of CDTG is depicted in Figure 3.1. It is assumed that a constrained-random test generation method is fixed and available. In the first iteration, some tests are randomly generated and simulated to obtain initial coverage result. Then CDTG methods are used to generate test parameters for test generators to generate tests in the next iteration. This iteration process continues until there is no coverage gain. Previous works in CDTG focuses on developing methods to analyze coverage re-

sults for obtaining test parameters. For example, [19] proposes a CDTG method based on Bayesian networks, [20] proposes a CDTG method based on Genetic Algorithm, [21] proposes a CDTG method based on Markov Models, and [22] proposes a CDTG method based on rule learning.

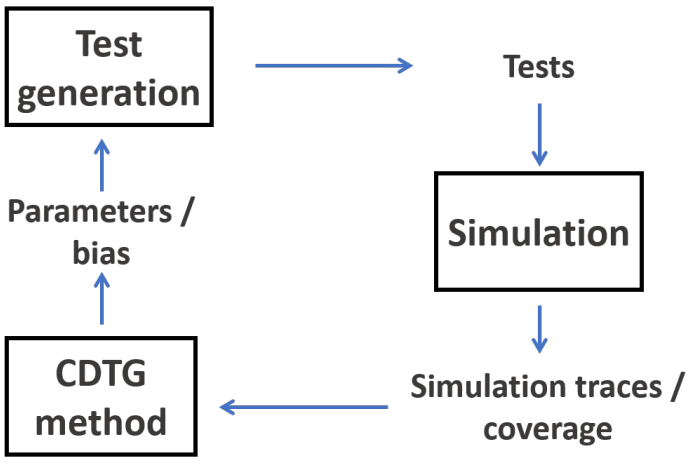

Figure 3.1: General flow of coverage directed test generation

The goal of this work is to investigate a software system that can generate tests for a given coverage points. Moreover, the software system can learn from examples to increase its capability. This is similar to that a junior engineer learn from examples provided by senior engineers. This software system differs from previous CDTG methods in terms of (1) in addition to providing test parameters, the system can increase the capability of test generation methods from examples and (2) when the system fails to deal with a given coverage point, instead of simply return "I do not know how to solve it," it provides its analysis to Senior engineers.

Our investigation starts from using descriptive rule learning methods whose learning models are explainable, i.e. a human can easily understand the model. For a descriptive rule learning model, given a model input, it is easy to know why the model produces its output. In the next section, a related work [22] about rule learning based CDTG

applications is described.

3.2 Rule Learning Applications in CDTG

Figure 3.2 shows the structure of the proposed method in [22]. This work demonstrates that rule learning methods can be applied in functional verification to assist engineers in generating tests for increasing coverage. The basic idea of the proposed method in [22] is to learn a rule composed of important signal values, then engineers can create test parameters based on the learned rule to generate tests in the next iteration.

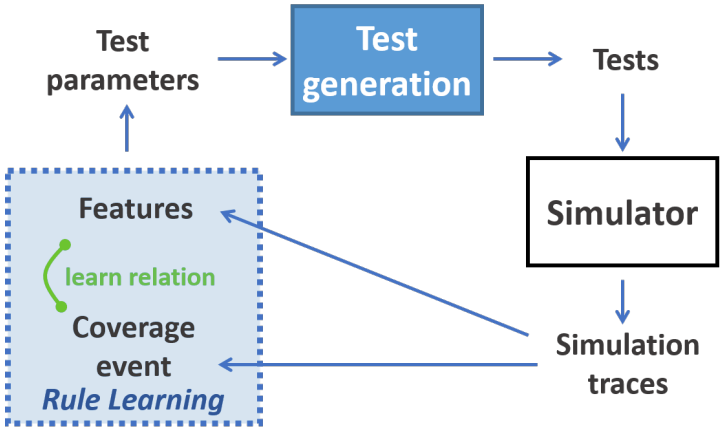

Figure 3.2: Overview of rule learning application in CDTG

In Figure 3.2, simulation traces are pairs of time and value for each design signal, coverage events are pairs of time and hit-or-not for a given coverage point, and features are the time-value pairs of selected signals and their combinations in simulation traces. The used rule learning algorithm is subgroup discovery [9].

Each sample in the data for rule learning is encoded as Boolean features and Boolean label. Each sample corresponds to a timeframe, which is like taking a snapshot of the circuit at a given time. A positive sample represents a given coverage point is hit at that timeframe, while a negative sample represents a given coverage point is not hit at that

timeframe. Features are defined by engineers manually. Discretization techniques are used to convert numerical features and categorical features to Boolean features. Also, a feature can capture temporal relations, e.g. a feature can represent signal transition that the feature is true if and only if the signal is high in the current timeframe and is low in the previous timeframe.

After learning a rule composed of the features, verification engineers generate test parameters based on the learned rule. Typically the definition of features is related to test parameters so it is not hard to come out the test parameters.

However, there are still several challenges to overcome in [22]. (1) Converting the learned rule to test parameters may not be trivial. (2) It requires domain knowledge to select features. (3) It is hard to analyze the reason of failures of sub-group discovery learning results. (4) Rule learning methods do not work well on extremely imbalanced data, i.e. one or few positive samples and many negative samples, which is a common situation in verification. Also, [22] discusses only the rule learning application and does not discuss situations that test generation methods do not have ability to generate the intended tests.

3.3 The Proposed Method

This dissertation further investigates these challenges and proposes a software system that can leverage rule learning methods to increase coverage. Moreover, the proposed system has the ability to learn from examples to increase its capability in terms of test generation and coverage analysis.

The framework of the proposed system is shown in Figure 3.3, where there are three key components: test generation, a signals database, and rule learning methods. A test generation method is proposed such that it is controllable and can increase its capability

by learning from new test samples. A signals database stores important design signals and their relationship. This database is introduced to deal with the feature selection problem for rule learning. Last, a new rule learning method is developed to deal with extremely imbalanced data. Also, it does not have any hyper-parameters and optimization heuristics so it is easy to analyze a failed learning result.

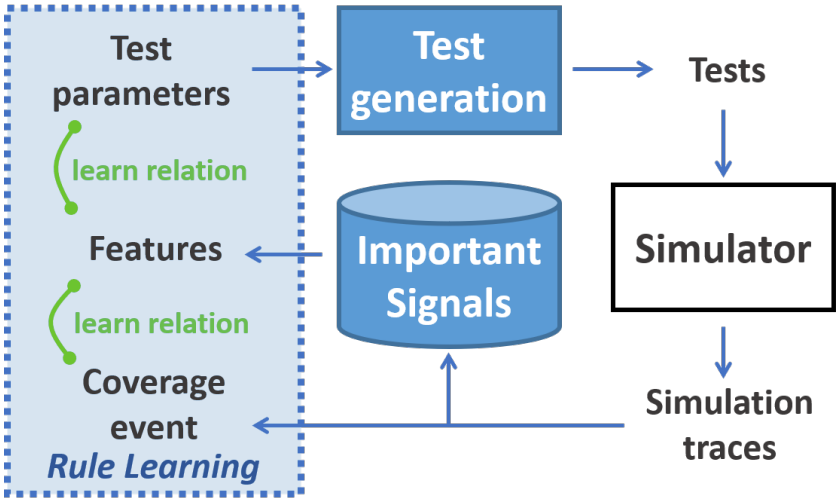

Figure 3.3: The proposed verification system

In Chapter 4, a method called constrained process discovery is proposed to learn a test generator from manually developed tests. The learned test generator is able to increase its capability, i.e. generating tests for testing new design functionality, by learning from tests provided by experienced engineers. In addition, the test generator is controllable by its input parameters so we can direct test generation with coverage analysis results.

There are three key ideas of constrained process discovery. The first key idea is to introduce primitives – each primitive is a block of code. With primitives defined, a test can be represented symbolically by a sequence of primitives. Then, the problem becomes to learn a finite state model that is able to generate the sequence of primitives. However, the learned model can easily generate tests that violate design assumptions, which are called invalid tests. Hence, the second key idea is to introduce constraints

among primitives to overcome the challenge of getting rid of invalid tests. In addition to the above constraints, test parameters are represented as constraints, e.g. some primitive must appear or not appear in the generated tests. The third key idea is to convert a learned finite state model and constraints into a SAT problem for generating tests that satisfy both the model and constraints.

Chapter 5 discusses a signals database to store important design signals and the relationship of the signals that can be utilized in data preparation for rule learning. The database stores sets of signals, where each set represents a particular relationship of the signals, e.g. the combination of the signals can infer a coverage point. In addition to manually inputting design signals and their relationship, we propose using text mining methods to extract the important signals and relations from design documents, to establish an initial database. Regarding the learning capability of the proposed system, the database can increase its data by itself from the results of rule learning methods. For example, if a rule is successfully learned, a new set of signals is stored. Moreover, the verification system can generate new tests randomly and run rule learning methods to capture new signal relationships. Also, the database can increase its data from experienced engineers – recording the combination of signals experienced engineers used. The database can be view as a place for knowledge accumulation.

Chapter 6 discusses the problems of decision tree based rule learning methods in the proposed verification system and proposes a rule learning method, VeSC-CoL, to deal with the problems. It is shown that decision tree based methods cannot properly handle extremely imbalanced data, i.e. one or few positive samples and many negative samples. Also, a critical problem of applying machine learning methods in practice, in general, is that when a learning result does not work, to improve the learning result requires knowledge and experience of the learning method. For example, some possible approaches to improve learning results are gathering more data, finding necessary features, tuning

hyper-parameters, etc. The decision varies from person to person based on one's experience. In contrast, VeSC-CoL can handle extremely imbalanced data and suggest the next actions to improve the learning results. An important property of VeSC-CoL is that it is guaranteed to find the simplest model that fits data. If there are multiple simplest models, VeSC-CoL can return all of them for analysis. This property enables VeSC-CoL to handle extremely imbalanced data or even learning with only negative samples.

There are four situations where the proposed system reports unable to generate tests to a given coverage points, as illustrated in Figure 3.4. There are two rule learning tasks in the proposed system. The first learning task is to learn a relation between a coverage point and design signals. The second learning task is to learn a relation between test parameters and a rule composed of design signals. For each learning task, a failure can be due to two reasons: (1) the candidate rules are too complex, and (2) there are too many candidate rules for analysis. These two reasons can be determined by VeSC-CoL while traditional rule learning methods cannot provide such information.

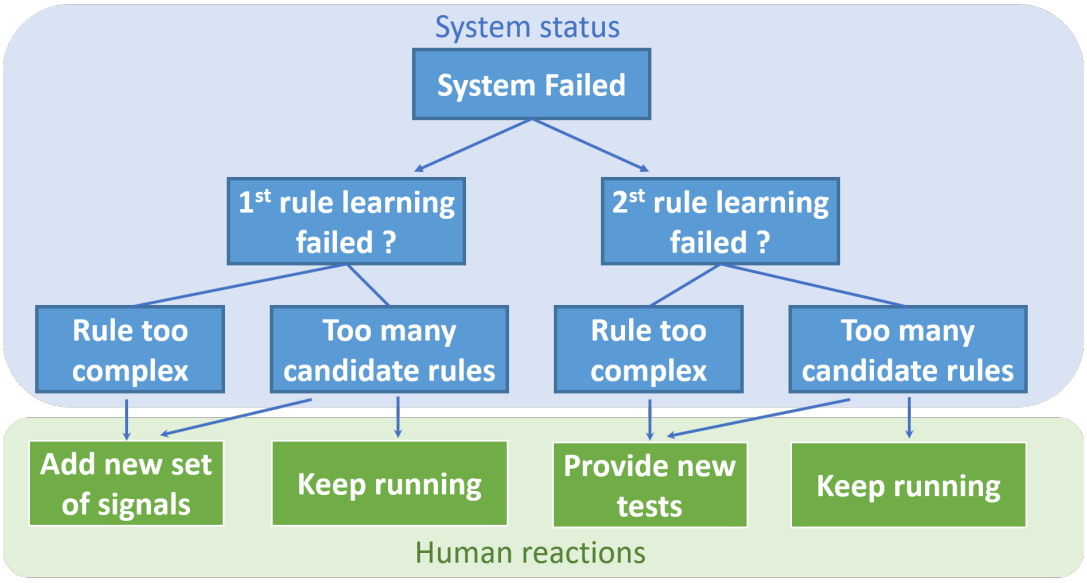

Figure 3.4: Four failure situations

If it is the first learning failed and due to candidate rules too complex, the proposed system asks engineers for a better set of design signals. Then, engineers can select a new set of signals in the signals database or add more design signals to the signals database. If it is the first learning failed and due to too many candidate rules, the proposed system reports this situation and engineers can decide to let the system run a few more iterations to reduce the number of candidates, or if running more iterations is time-consuming, add a new set of signals. Similarly, for the second learning failed, if a failure is due to rule too complex, the proposed system asks engineers to provide a test that can satisfy the rule obtained from the first learning result. If a failure is due to too many candidate rules, engineers can decide to run more iterations or provide new tests.

Chapter 4

Learning to Generate Tests

4.1 Introduction

The proposed verification system requires a test generation method that has the following three properties: (1) it can be controlled by some parameters, (2) it can increase its capability by learning from new test examples, and (3) it has the ability of generalization to generate tests different from the test examples, so the test generation method does not just memorize the test examples. This chapter proposes a method to construct a test generator possessing these properties.

The approach presented in this work is rooted in *grammatical inference* [23]. However, this work presents a novel implementation that combines *process discovery* [24] and constraint solving to achieve the learning.

4.2 Grammatical Inference

Without loss of generality, assume each direct test is a C program. To learn from a set of C programs, we need a way to represent the programs. This representation is

the basis for learning and decides the *learnability* of the learning problem we formulate. In this work, each C program is represented by a set of *primitives*. One can think of a primitive as a parametrized script that, when it is called, produces a piece of C code. These primitives serve as a TPI (Test Programming Interface) for a person to write direct tests.

With primitives defined, a direct test can be represented symbolically. For example, a test can be represented as a sequence of *steps*, e.g. [A,B,C,...]. After primitive encoding, each test then can be viewed as a "sentence" example derived from an unknown formal language [25]. In other words, primitives are *words* of the unknown language. Then, *grammatical inference* [23] can be applied to discover an automata model (such as a finite automaton) to describe this language based on a given set of examples.

The *learnability* problem in grammatical inference asks whether a model can be learned with a finite number of samples. Define in-model samples and out-model samples as the samples complying with and not complying with the model to be learned, respectively. The main result of [26] points out that if only in-model samples are available, the only learnable class is the set of finite-length languages, i.e. there is a bound on the maximum length of a sentence. If both in-model and out-model samples are available, then all classes up to the Context-Sensitive grammar in the Chomsky Hierarchy can be learned [25]. In this work, we consider the case that only in-model samples are available, i.e. all the available direct tests comply with the hidden model.

Most grammatical inference algorithms do not handle only in-model samples because to invalidate a learned model, it requires out-model samples, i.e. Regular Positive Negative Inference (RPNI) [27] and Biermann & Feldman's algorithm [28]. Without out-model samples, grammatical inference algorithms can return the most general model, i.e. a model that accepts every sequence. To learn from only in-model samples, there is a research field, process discovery.

4.2.1 Process Discovery

Since the work in [26], the learnability of a finite automaton is among those that received the most attention [29]. More recently, *process discovery* emerged as a separate field targeting business applications. Process discovery is applied to learn a process model from an event log recording instances of business transactions [30]. Each instance is represented as a sequence of transaction steps, similar to our representation of a test as a sequence of primitive steps. In process discovery, a common representation for the process model is Petri Net [30] where the graph model allows loops and concurrency. Hence, learning such a process model is as hard as learning a finite automaton.

Process discovery and the proposed approach have fundamentally different objectives. Process discovery is for discovering business intelligence from event logs. Hence, it is important for the learning model to be interpretable. Simplicity of the model to enable visualization is a key consideration. Our goal for the model is to enable test generation. Therefore, it is not necessary for our learning to produce a model summarizing all the learned information into a single interpretable model. This difference enables us to develop a novel learning approach described in the next section.

4.3 Learning from Test Examples

To illustrate the basic idea of learning in process discovery, consider the following simple example. Suppose A-G represent the primitives. Suppose we have three tests: [A,B,C,D,H], [B,C,E,F,D,H], and [A,B,C,E,G,D,H]. Fig. 4.1 shows a process model learned from these three tests.

This model is built based on a so-called *prefix rule* [24]. A prefix rule decides whether two steps with the same name should be represented with the same node in the process model. Suppose one test contains a segment αX and another test contains a segment

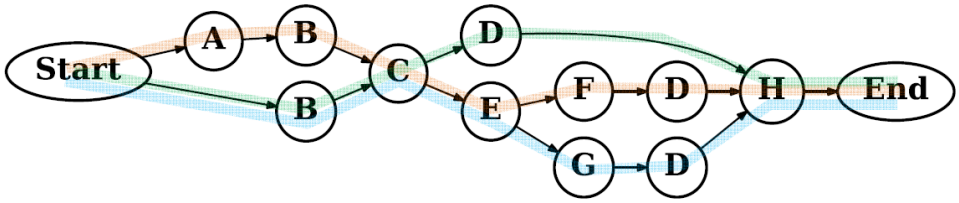

Figure 4.1: Process model using 1-prefix rule

βX, where α and β each is a sequence of one or more steps and X is a step. Given a length requirement $l \geq 0$, let α_l be the last l steps in α and β_l be the last l steps in β. An *l-prefix rule* means that the two X steps would be represented by one node in the process model if $\alpha_l = \beta_l$.

Fig. 4.1 is based on the 1-prefix rule. For example, there is one C node in the model, representing the three C steps in the three tests. This is because every time C is involved, the step before is always B. Hence, the 1-prefix rule infers that there is only one way to use the C primitive, resulting in one C node in the model.

A process discovery algorithm essentially decides if two or more steps should be represented as a single node [30]. Observe that merged nodes can also cause new instances to be included. Including new instances not shown in the training set is called *generalization* in machine learning. In Fig. 4.1, three new tests are highlighted. For example, because of the merged node C, the two segments [A,B] and [E,F,D,H] can be combined to produce the new test [A,B,C,E,F,D,H].

Consider now a new test [S_1, S_2, B,C,D,H, S_3, S_4] is provided for learning. Fig. 4.2 shows the resulting model by adding this new test (with 1-prefix rule). S_1 to S_4 are new primitives. It is interesting to observe that the resulting model contains two new tests (as highlighted) involving the new S's primitives.

If we consider Fig. 4.1 as the verification knowledge learned from direct tests and the new test as a penetration test example provided by an expert, Fig. 4.2 illustrates how

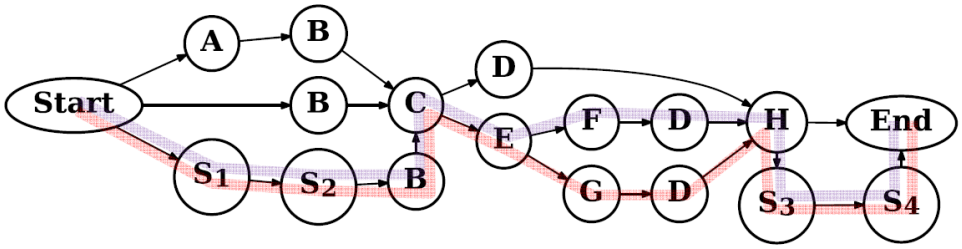

Figure 4.2: Process model by adding one more test

process learning can generalize from one penetration test to more penetration tests. Of course, if more penetration tests are provided, generalization can also take place among them.

Note that a process model generated based on a prefix rule can be viewed as a deterministic finite automaton (DFA). If concurrency is allowed in the tests (i.e. two segments are executed concurrently), then the model can be treated as a nondeterministic finite automaton (NFA). However, since each NFA can be converted into a DFA, for the rest of the discussion, we consider a process model as a DFA.

4.3.1 Constrained process discovery

Instead of learning a single process model as that in process discovery, our approach splits the learning into two parts: (1) learning an *upper-bound model*, and (2) learning a set of constraints. Fig. 4.3 depicts this approach.

The goal of an upper-bound model is to capture a bound on the space of all possible tests such that a desired test is ensured to be in the space. However, because it is an upper bound, the model can include many undesirable tests. A separate *constraints database* is maintained to impose constraints between and among primitives. Constraints can be learned independently of the process learning. Then, for test generation, the upper-bound model and the constraints from the database are combined for constraint solving. Each solution represents a test. In this work, we use a Boolean satisfiability (SAT) solver for

constraint solving.

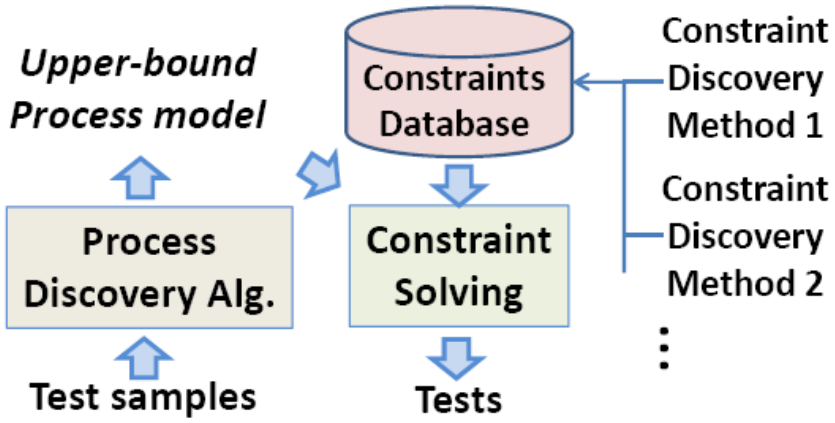

Figure 4.3: Overview of the proposed method

The constraints database provides a natural place to accumulate and share verification knowledge. This knowledge can be added manually, based on previous learning sessions or based on a separate constraint discovery method. It is intuitive to observe that as more constraints are included, the test space becomes smaller, enabling more focus tests to be generated. Unlike process discovery where a process model (e.g. a Petri Net model) stores all the learned information, in Fig. 4.3 the information is split into two parts and there is no single structure to represent all learned information.

In process discovery, one major concern is to control the *underfitting* and *overfitting* of a model [24]. In our context, underfitting means the model contains undesirable tests and overfitting means the model excludes some desirable tests. If a prefix rule is used, the key concern becomes choosing an l for the best tradeoff between underfitting and overfitting. However, because of the inflexibility of such a learning algorithm, the resulting model usually has both issues, containing undesirable tests and missing some desirable tests.

Our approach starts with an underfitting model (the upper-bound model) which is gradually refined with constraints. Suppose every constraint added to the database is valid (e.g. validated by a person before adding it to the database). Then, the approach

ensures no desirable tests would be missed. As more constraints are added, the resulting model (implicitly existing) becomes closer to the desirable model. It is important to recognize that the gap between this resulting model and the desirable model is reflected in the loss of efficiency, i.e. Fig. 4.3 would produce undesirable tests that are not perceived as useful by the user.

4.3.2 The upper-bound model

Observe that for using an l-prefix rule, a larger l imposes a more stringent requirement for merging multiple steps into a single node. Hence, a larger l also leads to a less generalized model. Therefore, the upper-bound model based on a prefix rule is the 0-prefix rule model. Algorithm 1 depicts the detail of generating the upper-bound model using the 0-prefix rule.

Algorithm 4.1: Learning an upper-bound model

Input: a set of direct tests T
Output: a process model M

```
1  M ← empty, Add states start and end to M;
2  foreach t in T do
3  |   q0 ← start;
4  |   foreach q in t do
5  |   |   if state q not in M then
6  |   |   |   Add state q to M;
7  |   |   end
8  |   |   if arc (q0, q) not in M then
9  |   |   |   Add arc (q0, q) to M;
10 |   |   end
11 |   |   q0 ← q;
12 |   end
13 |   if arc (q0, end) not in M then
14 |   |   Add arc (q0, end) to M;
15 |   end
16 end
```

4.3.3 Constraint examples

A constraint describes a dependency relationship among multiple primitives. Table 4.1 illustrates four types of constraints to describe a relationship, which are a subset of temporal logic where $\mathcal{B}$ denotes *before* and $\mathcal{F}$ denotes *future*.

Table 4.1: Four constraints to describe a relationship

X $\rightarrow \mathcal{B}$ Y	If X is executed, Y is executed before X.
X $\rightarrow \mathcal{F}$ Y	If X is executed, Y is executed after X.
X $\rightarrow \mathcal{B}_k$ Y	If X is executed, Y is executed within k steps before X.
X $\rightarrow \mathcal{F}_k$ Y	If X is executed, Y is executed within k steps after X.

The constraints can be used with a negation "¬" to describe a relationship. Fig. 4.4 shows four example constraints between two primitives. Another useful example to forbid primitive X to be used twice, i.e. preventing loop back to X, is the constraint (X $\rightarrow \neg\mathcal{B}$ X).

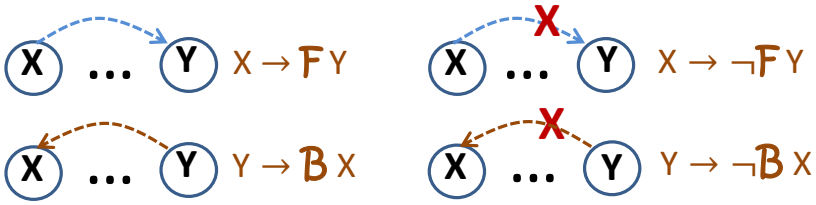

Figure 4.4: Example constraints between 2 primitives

Constraints involving more than two primitives can be added as well, for example manually or by a constraint discovery method such as frequent episode mining [31] which might discover that a segment [A,B,C] occurs frequently. As a result, the constraint that A,B,C should be used together and in the particular sequence can become a recommended constraint for a user to include or exclude.

4.4 Test Generation - SAT Encoding

To use a SAT solver for test generation, we need the three sets of clauses: (1) those to encode the 0-prefix process model, (2) those to encode the cross-primitive constraints, and (3) those to ensure generation of new tests.

4.4.1 Encoding the process model

Inspired by [14], we use an approach that is similar to the proof of NP-completeness, but instead of encoding a Turing machine, we encode a DFA (i.e. a process model).

Let N be the number of states in the DFA and L be the maximum length of the generated tests. The proposition symbols are

- Q_t^i, for $1 \leq$ i $\leq$ N, for $1 \leq$ t $\leq$ L.

Q_t^i is True if and only if the DFA is at state q_i at step t.

The encoding of transition relation is composed of two components. First, for each state q_i, the next state of the machine can only be the states that are directly connected from q_i. Thus, transition relation at step t is encoded as $(Q_t^i \rightarrow \vee_j Q_{t+1}^j)$, where j belongs to the set of state indices of all the child states of q_i. The overall encoding for all the steps and all the states is

$$\Pi_t \Pi_i (\neg Q_t^i \vee (\vee_j Q_{t+1}^j)). \tag{4.1}$$

The number of clauses of this component is O(LN).

Second, the machine cannot stay in more than one state at a time, i.e. *Onehot0* constraint. A naive encoding for this property is $\Pi_t \Pi_{i,j} (\neg Q_t^i \vee \neg Q_t^j)$ for all the pairs of the states and all the steps, however, the naive encoding requires O(LN^2) clauses that potentially leads to a complexity problem in reality. To deal with the complexity

problem, we use an approach described in [32], where the number of clauses reduces to O(N) at the expense of extra O(N) symbols. The extra proposition symbols are

- H_t^i, for $1 \leq$ i $\leq$ N, for $1 \leq$ t $\leq$ L.

H_t^i is True if and only if one of Q_t^j is True for $j \leq i$. Equivalently, H_t^i is True if and only if H_t^{i-1} is True or Q_t^i is True. Note that H_t^0 is set to False. Then the $Onehot0$ property can be encoded as at most one of H_t^i and Q_t^{i+1} is True. The overall encoding is

$$\Pi_t \Pi_i (\neg H_t^i \vee H_t^{i-1} \vee Q_t^i)(H_t^i \vee \neg H_t^{i-1})(H_t^i \vee \neg Q_t^i) \tag{4.2}$$

and

$$\Pi_t \Pi_i (\neg H_t^i \vee \neg Q_t^{i+1}). \tag{4.3}$$

Overall, the number of clauses of this component is O(LN) and the number of extra symbols is O(LN).

The encoding for the start state and the end state will be discussed in Sec. 4.4.2.1.

4.4.2 Encoding the constraints database

There are four types of constraints in the constraints database, $\rightarrow \mathcal{B}$, $\rightarrow \mathcal{F}$, $\rightarrow \mathcal{B}_k$ and $\rightarrow \mathcal{F}_k$. The following describes their encoding separately. The similar encoding approach can be applied to encoding the constraints with negation. We omit this part due to space limitation.

To encode $q_i \rightarrow \mathcal{B} q_j$, a naive method is $\Pi_t (Q_t^i \rightarrow \vee_{s<t} Q_s^j)$. This encoding ensures that if the machine is at q_i at step t, there exist $s < t$ such that the machine is at q_j at step s. However, the naive encoding may lead to a complexity problem because the number of literals required for each constraint is O(L^2). We propose another encoding method to reduce the number of literals to O(L) at the expense of extra O(L) symbols.

We introduce new proposition symbols

- B_t^j, for 1 ≤ j ≤ N, for 1 ≤ t ≤ L.

B_t^j is True if and only if the machine is at state q_j at some steps < t. The property of the new symbols are maintained by the following relations: (1) B_1^j is False. (2) B_t^j is True if and only if B_{t-1}^j is True or Q_{t-1}^j is True. The first relation is encoded as $(\neg B_1^j)$. The second relation is encoded as

$$\Pi_t(\neg B_t^j \vee B_{t-1}^j \vee Q_{t-1}^j)(B_t^j \vee \neg B_{t-1}^j)(B_t^j \vee \neg Q_{t-1}^j). \tag{4.4}$$

With the help of symbols B_t^j, the encoding of $q_i \rightarrow \mathcal{B}q_j$ becomes $\Pi_t(Q_t^i \rightarrow B_t^j)$, whose final encoding is

$$\Pi_t(\neg Q_t^i \vee B_t^j). \tag{4.5}$$

This new encoding for $q_i \rightarrow \mathcal{B}q_j$, including the encoding of the relation of new symbols, requires O(L) clauses, which is the same as the naive encoding, but the number of literals is reduced to O(L). Overall, let C_b be the number of constraints of this type, the number of clauses required is O(LC_b) and the number of extra symbols is O($L * min(C_b, N)$). Note that the number of extra symbols is always no larger than O(LN).

The method to encode $q_i \rightarrow \mathcal{F}q_j$ is similar to encoding $q_i \rightarrow \mathcal{B}q_j$. We introduce new proposition symbols

- F_t^j, for 1 ≤ i ≤ N, for 1 ≤ t ≤ L,

where F_t^j is True if and only if the machine is at state q_j at some steps > t. The encoding for the relation of the new symbols is

$$\Pi_t(\neg F_t^j \vee F_{t+1}^j \vee Q_{t+1}^j)(F_t^j \vee \neg F_{t+1}^j)(F_t^j \vee \neg Q_{t+1}^j), \tag{4.6}$$

and the encoding of $q_i \rightarrow \mathcal{F}q_j$ is

$$\Pi_t(\neg Q_t^i \vee F_t^j). \tag{4.7}$$

The idea for encoding $q_i \rightarrow \mathcal{B}_k q_j$ is straightforward, $\Pi_t(Q_t^i \rightarrow \vee_s Q_s^j)$ for s in $\{t-1, t-2, ..., t-k\}$. Hence, the corresponding SAT clauses are

$$\Pi_t(\neg Q_t^i \vee (\vee_s Q_s^j)). \tag{4.8}$$

There are O(L) clauses for each constraint. Let C_{bk} be the number of constraints of this type. Then the overall number of clauses of this type is O(LC_{bk}).

Encoding $q_i \rightarrow \mathcal{F}_k q_j$ is similar to encoding $q_i \rightarrow \mathcal{B}_k q_j$. The corresponding SAT clauses are

$$\Pi_t(\neg Q_t^i \vee (\vee_s Q_s^j)) \tag{4.9}$$

for s in {t+1, t+2, ..., t+k}.

4.4.2.1 Generating new tests

To generate a test, we set the start state to be the first state. The corresponding clause is

$$(Q_1^{start}). \tag{4.10}$$

To ensure the generated test reaches the end state, we add a clause

$$(B_L^{end} \vee Q_L^{end}). \tag{4.11}$$

Recall that B_L^{end} is True if and only if the machine is at q^{end} before step L. With this constraint, the length of the generated tests is not fixed to L but can be any length

smaller than or equal to L.

To ensure the generated test are not the same with the training direct tests and the tests already generated, we add additional clauses for each test that has been seen. Let $\alpha = q_{t_1} q_{t_2} \dots q_{t_M}$ be a test with length M. To avoid generating α, we add an clause

$$(\neg Q_1^{t_1} \vee \neg Q_2^{t_2} \vee \dots \vee \neg Q_M^{t_M}). \tag{4.12}$$

Let τ be the number of tests that have been seen. The number of clauses of this type is τ.

Table 4.2 summarizes the number of symbols and the number of clauses of the SAT encoding.

Table 4.2: The number of symbols and clauses of the proposed SAT encoding.

# symbols for states	$O(LN)$
# symbols for Onehot0	$O(LN)$
# symbols for constraints	$O(L * min(C, N))$
# symbols, overall	$O(LN)$
# clauses for transitions	$O(LN)$
# clauses for Onehot0	$O(LN)$
# clauses for constraints	$O(LC)$
# clauses for new tests	$O(\tau)$
# clauses, overall	$O(L(N + C) + \tau)$

L is the maximum length. N is the number of states. C is the number of constraints. τ is the number of tests that have been seen.

4.5 Experimental Results

We implemented the approach based on an in-house simulation based RTL verification environment for a commercial dual-core microcontroller SoC. There are 194 verification primitives and each of them corresponds to a block of C code. The tests in the verification

environment are written in terms of these primitives, then compiled into object code and executed by the cores.

There are 22 cross-primitive constraints. These constraints were added manually when the primitives were developed. In the first experiment, for learning we took 30 direct tests used for verifying the on-chip system controller module that manages resource allocation, power modes, and security policies. The test length is between 46 and 63 primitives.

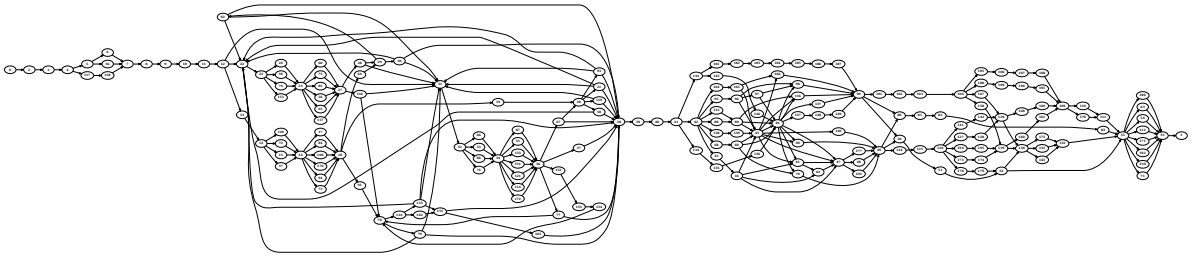

Figure 4.5: The upper-bound process model from the 30 direct tests

Fig. 4.5 shows the resulting upper-bound process model using the 0-prefix rule. There are 196 states in the graph including the start state and the end state. The model together with the database is then given to the SAT solver, zChaff[33], to generate tests. The maximum length L is set to 70, which is larger than the maximum length of the 30 direct tests. The SAT encoding involves 41090 symbols and 110659 clauses. Note that if using the naive encoding for $Onehot0$, the number of clauses would exceed 10^7. The run time of the SAT solver is negligible. It takes less than one second to generate a test.

The effectiveness is measured based on the coverage of a set of coverage points (CPs) defined by the verification engineers for the system controller. The original 30 direct tests cover 167 CPs. Then, 500 new tests were randomly generated by the proposed method. Fig. 4.6 shows the newly-covered CPs by those tests cumulatively as each new test is produced and simulated. Together, the 500 new tests cover additional 85 CPs. The result shows that the newly generated tests are capable of covering new functionality.

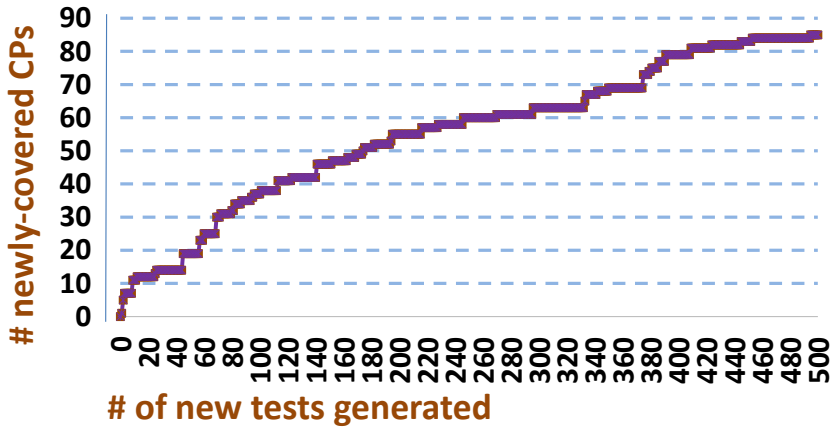

Figure 4.6: Coverage improvement

Table 4.3 summarizes the finding with additional results. The benefit is shown by seeing that new tests produced from the process model can always improve the coverage.

Table 4.3: Additional coverage results

# manual tests	# originally covered CPs	# generated tests	# newly covered CPs
30	167	500	85
35	216	500	64
40	244	100	64

Fig. 4.7 then shows the result from a separate experiment. 30 different direct tests for verifying the system controller were used for learning. While the earlier results show coverage improvement, this result illustrates coverage frequency improvement (and also shows that the coverage improvement is not specific to a particular set of direct tests in use for learning). In this experiment, 100 new tests were generated. They cover 68 additional CPs. More importantly, the coverage frequency is improved across almost all CPs. This frequency improvement shows that the new tests can cover the similar functionality of the original tests, i.e. they capture the same intent of the original 30 tests.

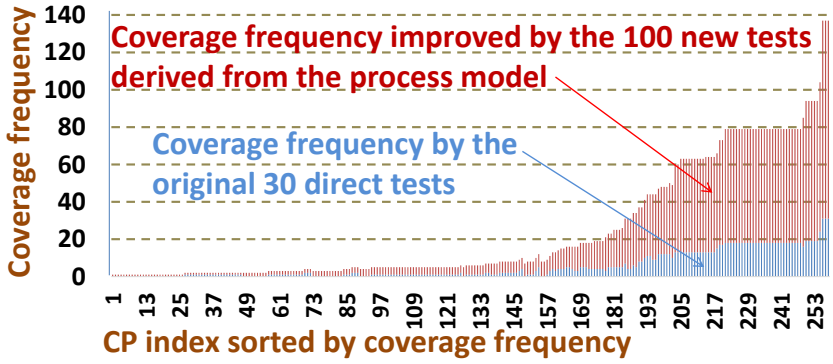

Figure 4.7: Coverage count improvement

4.6 Summary

In this work, a novel approach called constrained process discovery is proposed for learning from the direct tests developed by experts. After the learning, the software machine functions as a surrogate for the experts to generate new direct tests. The software machine comprises two components, an upper-bound model, and a constraints database. The constraints database serves as a knowledge center for accumulating verification knowledge.

A SAT solver is used for generating tests that comply with both the upper-bound model and constraints database. The SAT encoding is based on the concept of step-extension. Two techniques are used to reduce the complexity in terms of the size of SAT encoding. The number of symbols of the proposed encoding method is O(LN) and the number of clauses of the proposed encoding method is O(L(N+C)), where L is the maximum length of the generated tests, N is the number of states and C is the number of constraints.

The proposed test generator has the three required properties in the proposed verification system: (1) It can be controlled by test parameters where test parameters are treated as additional constraints, (2) It can learn from new test samples by adding new

states to its upper-bound model, and (3) It can generate tests different from the test samples as long as test samples have shared primitives.

The proposed approach is implemented in a verification environment for a commercial SoC. Experiment results show that the proposed approach can generate new tests, and the new tests can cover new design functionality.

Chapter 5

Extracting Design Signals from Documents

5.1 Introduction

In the proposed verification system, a signal database is required to store design signals that have high-level meaning, i.e. understandable by human engineers, and some combination of the signals if the combination is meaningful. The reasons for this is threefold. First, rule learning algorithms cannot process a large number of signals in a design under verification, so signal selection is required. Second, when human involvement is required to generate tests according to a learned rule learning result, if the rule is composed of low-level signals, it is hard for a human to understand the meaning of the design states satisfying the rule. Last, the performance of rule learning algorithm is better when irrelevant signals are not presented in training data, hence a smaller set of signals helps.

In addition to establishing a signal database manually, this chapter proposes a method to establish a signal database from design documents. It is assumed that the signals described in design documents are at high-level, important, and meaningful to human

engineers. Intuitively, design documents are concise and only important signals are presented. Also, design documents can be views as places where experienced engineers put their knowledge.

5.2 Text Mining and Signal Mapping

The proposed approach is illustrated in Figure 5.1, where there are two steps: text mining and signal mapping. Text mining techniques are utilized process design document to extract words that is relevant to design signals. Then, signal mapping methods are employed to map the extracted words to real design signals.

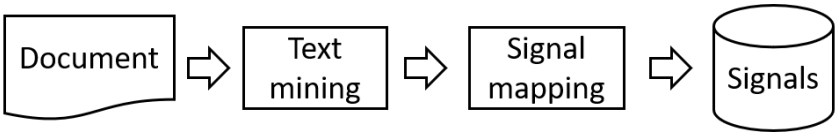

Figure 5.1: The overall flow of the proposed approach.

5.2.1 Text Mining

The goal of this procedure is to extract words that are relevant to design signal names. Text mining is a process of extracting information from text and typically it involves natural language processing. There are two natural language processing techniques used in our approach: tokenization and part-of-speech tagging (POS tagging). The tagging results are then passed to a selection method to extract words of interest.

5.2.1.1 Tokenization

Tokenization is a technique used to segment a string into substrings. There are two types of tokenization: sentence tokenization and word tokenization. Sentence tokeniza-

tion aims at partitioning text into sentences, while word tokenization splits a sentence into words.

The left side of Fig. 5.2 shows an example of sentence tokenization, where the input text is

"Are you OK? Dr. Pete is nearby.",

and the expected sentence tokenization result is

{"Are you OK?", "Dr. Pete is nearby."},

which is the same as how human understand the text. Sentence tokenization is not as trivial of a task as it may seem. For example, one may think that splitting a text by periods just works, however in the given example, splitting by periods ends up considering "Dr." as a sentence. One approach to solving this problem is to train a model to identify sentence boundaries [34].

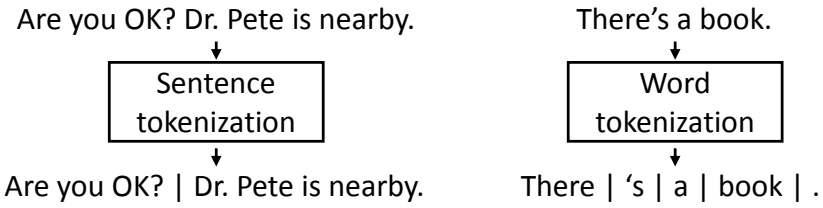

Figure 5.2: Exmaples of tokenization.

The right side of Fig. 5.2 shows an example of word tokenization, where the input text is

"There's a book.",

and the expected sentence tokenization result is

{"There, "'s", "a", "book", "."}.

Note that different tokenizers may treat punctuation differently, e.g. making individual punctuations as tokens. Although splitting a sentence by spaces works most of the time, special cases must be taken care of such as verb contractions (e.g. can't) and Saxon genitive (e.g. mother's).

5.2.1.2 POS tagging

Part-Of-Speech (POS) tagging is a process to set a POS label to each word in a sentence (which is word-tokenized), where a POS is a word category in which words possess similar grammatical properties. Some simplified examples of POS are noun, verb, adjective, etc. In reality, there are more specific categories. Fig. 5.3 shows an example of POS tagging where the input sentence is:

FOO is a read-only, one-hot register.

In the tagging results, NNP stands for proper noun, VBZ stands for present and singular verb, DT stands for determiner, JJ stands for ordinal adjective, and NN stands for common singular noun. Commonly used methods include rule-based models [35] and stochastic models [36].

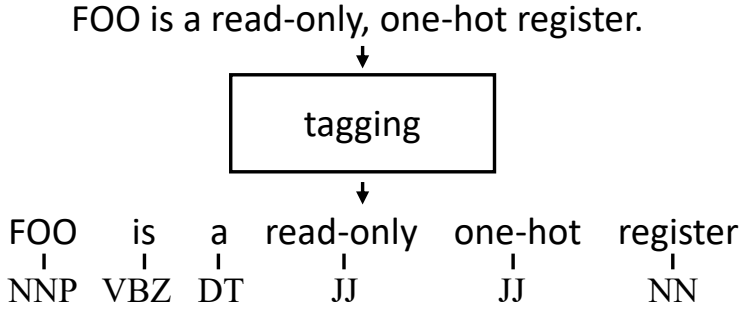

Figure 5.3: An example of POS tagging.

The POS tags provide grammatical information of each word in a sentence, which is useful for analyzing text. In our application, it is intuitive that the majority of signal

names will be nouns or proper nouns, and are highly unlikely to be labeled as tags such as adjective and verb.

5.2.1.3 Word selection

The objective of this step is to extract words relevant to the names of design signals. Typically it is customized and ad-hoc. We use a rule-based approach for word selection. First, we find all the words that are labeled as common noun or proper noun. Second, we design rules to select words of interest. Our rules are based on regular expressions and it is easy to add or remove a rule in our rule-based implementation. Example rules for hardware signal extractions include (1) words composed of only uppercase letters, and (2) words that have a underscore, "_". Lastly, a word is not selected if it is in our exclusion list, which contains known words that do not refer to design signals.

5.2.2 Mapping Text to Design Signals

This procedure maps the words extracted by text mining to the real design signals. Our approach is based on name matching and filtering. Fig. 5.4 depicts the flow of the proposed name mapping procedure.

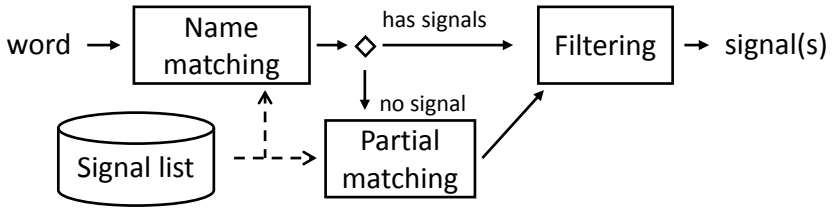

Figure 5.4: The flow of the proposed mapping procedure.

For each word, we search if there are matches from the list of all the design signals. The first step is to find the signals whose names exactly match the queried word. If there is no such signal found, a partial matching method is applied, which tries to find

if there are matches for certain substrings of the queried word. A typical substring to be matched can be chosen from segments of the queried word separated by underscores. Sometimes substrings that happen to be commonly used words (e.g., start, stop and etc) are also considered.

A filtering mechanism is required because normally there are lots of signals in the matching result. The filtering outputs signals with the highest score, where the score is calculated based on the length of matched string, the depth of the signal in the design hierarchy and whether the signal is in the module we are interested in.

5.3 Data Processing and Rule Learning

This section describes the data processing method and introduces the basic of decision tree based rule learning method. The goal of the data processing method is to convert simulation traces into a format ready for rule learning.

5.3.1 Data Processing

For each simulation trace, we first decide the required timestamps, and each sample correspond to a timestamp. Next, we define Boolean features that are used to represent each sample. Last, post-processing is used to remove duplicate samples.

5.3.1.1 Time discretization

Given a set of signals, for each test, we are interested in the timestamps whenever a value change of the signals occurs. Formally speaking, given a test, given a set of signals, and let $V(t)$ be the vector of the signal values at time t, we extract the set of timestamps $\{t_1, t_2, \ldots, t_n\}$ satisfying that $\forall t \in [t_i, t_{i+1})$, $V(t)$ holds the same and that $\forall i, V(t_i) \neq V(t_{i+1})$.

5.3.1.2 Binarization

We treat the data as categorical. Suppose during simulation, the set of observed values of Sig_A is $\{v_1, v_2, \ldots, v_n\}$, then n features are created: "Sig_A=v_1", "Sig_A=v_2", ..., "Sig_A=v_n". At time t, the value of "Sig_A=v_i" is 1 if the value of Sig_A is v_i, otherwise "Sig_A=v_i" is 0.

There are potential exponential explosion problems. To our experience, it happened only for data path signals. If a signal has more than 16 values, then we let the users decide whether they want to include this signal in the feature set. Alternatively, they can decide to create fewer bins for the signal on their own.

5.3.1.3 Including Value Transitions

For each signal, we create another set of features to indicate the value transitions. Our empirical study shows that without these features, the performance of rule learning is not acceptable because most assertion coverage points involve temporal properties.

The procedure is similar to binarization, but the created features are "Sig_A=v_i to v_j". At time t_k, the value of the feature is 1 if Sig_A is v_i at time $t_k - 1$ and Sig_A is v_j at time t_k, otherwise 0.

5.3.1.4 Time alignment

The purpose of time alignment is to deal with the asynchronous relation between assertion coverage and signals. We observed several cases where assertion coverage has its dedicated clock. Creating new timestamps doesn't work because we can have two timestamps, t_1 and t_2, such that $V(t_1) = V(t_2)$ but one hits the coverage point and the other does not.

For each assertion coverage point, we find the maximum timestamp that is not greater

than the time when the coverage point is hit. Then we use this timestamp as when the coverage point is hit.

5.3.2 Rule Learning Algorithms

Given a set of samples of different classes, rule learning algorithms aim at finding a rule that is able to distinguish samples from different classes. In our application, we have two classes of samples, i.e. positive samples that hit the target assertion coverage point and negative samples that do not. Our goal is to find a rule, composed of features processed from the previous procedure, that can explain why the target assertion coverage point can be hit. For example,

$$!(Sig_A{=}1) \wedge (Sig_B{=}0 \text{ to } 1) \Rightarrow \text{hit target coverage.}$$

There are many off-the-shelf rule learning algorithms. Examples include subgroup discovery [9], CN2 [10], and Classification and regression trees (CART) [12].

The proposed approach does not depend on a specific rule learning algorithm. In our experiment, we used CART.

CART belongs to a family of decision tree classifiers. Training decision tree classifier models is an iterative process. Given a set of samples, the training algorithm checks all the features and decides which one can best split the set into two subsets, where the best split is that where the two subsets are close to pure. A set is pure if it contains only samples of a single class. Commonly used metrics to measure the quality of splitting are Gini impurity and information gain. Then, the same procedure continues on the two subsets. This iterative process ends when a subset is pure, or there is no feature for further splitting.

Each path starting from the root of a decision tree model corresponds to a rule. The rule is the conjunction of the decisions along the path. Note that a path does not

necessarily end at a leaf. Fig. 5.5 shows a decision tree example. All the right branches are True branches, and all the left branches are False branches. The highlighted path corresponds to the rule

$$!(Sig_A{=}1) \wedge (Sig_B = 0 \text{ to } 1).$$

Depending on the training results, we have different methods to extract rules from a decision tree. (1) If there are nodes that are pure and contain only positive samples, then the extracted rule is the disjunction of rules that correspond to the paths to all the pure and positive nodes. All the pure and positive nodes are treated equally. (2) Otherwise, the extracted rule is the one corresponding to the path leading to the node with the highest ratio of positive samples.

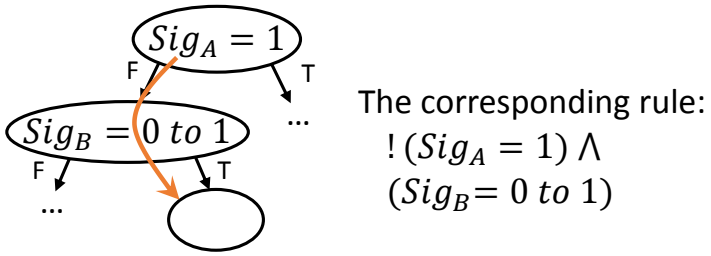

Figure 5.5: Rule extraction of a given node.

5.3.2.1 Dealing with Overfitting

Overfitting is a common machine learning problem whose root cause is either too many features, i.e. high dimensionality, or too few training data samples. To overcome this problem, we need to either remove irrelevant features or increase the number of training data samples. Increasing the number of training data samples does not work because of the extremely low hit rates of some assertion coverage points. Therefore, we resort to learning from smaller groups of features. This leads to the question of what features should be discarded and what features should be kept.

The idea is to learn from other assertion coverage points. Often, not all the assertion coverage points are independent. A group of features that is relevant to one assertion coverage point can be relevant to another assertion coverage point. Following this thought, for each assertion coverage point that has 100%-accurate rules, a group is created containing all the features used in its decision tree. After this step, multiple groups are created. Then, for each group, rule learning algorithms are applied to assertion coverage points that have no 100%-accurate rule. The rule with the highest accuracy is reported as the final result.

5.4 Experimental Results

The proposed approach was evaluated on a commercial dual-core microcontroller SoC targeting ultra-low power applications. The experiments focused on its system low-power control unit, which controls the system to enter and exit various power modes. This unit monitors triggering events of power mode transitions and then generates proper control sequences to clocks, power, and system memories to execute the transitions.

The in-house verification environment was a C-test based environment. The C stimuli were compiled into machine code and then executed on cores in RTL simulation. The correctness was insured both by self-checks in C-stimuli and the checkers in the testbench. Assertion coverage data was collected by a commercial coverage reporting tools.

We applied the proposed approach to 168 assertion coverage points created by other engineers during project development. Note that these assertion coverage points concern not only the system low-power control unit activities, but also other activities in the system to capture the overall system states of interest.

1000 tests were pre-run in this experiment, and they were partitioned into two equally sized sets for training and validation respectively. For each assertion coverage point, based

on the features extracted from the document, a rule was learned from the training tests. Then, each rule was validated on the other 500 tests. We show the effectiveness of our approach by comparing the hit rate of the training tests, which are randomly generated, to the hit rate of tests satisfying the rules of all the assertion coverage points.

Without feature selection processes, it is infeasible to apply rule learning algorithms. Before running into the theoretical overfitting problem, in practice, the cost of collecting and processing simulation data is prohibitive in terms of time and space. For instance, it required more than 2TB storage to save the whole chip simulation traces of 1000 tests.

5.4.1 Signal Extraction Based on Design Documents

The design document to start with is the design reference manual, which is a 49-page PDF file. There is no formal format of this document. In natural language, it describes design functionality, register usages, interface protocols, etc., with plenty of tables, diagrams, and waveforms.

We processed this document in Python2.7. The Python package pdfminer [37] was used to extract text from the PDF document. Next, the Python package nltk [38] was used to tokenize the text and tag the words. An exclusion list was created to ensure that we did not select words that were obviously irrelevant to design signals. After this procedure, 66 words were extracted from the document for the mapping procedure.

After executing the mapping procedure, there were 42 words that could be mapped to design signals. The 24 words that did not have mappings included the names of other hardware modules and special abbreviations. 46 design signals were obtained after the mapping procedure. We observed that two words may map to the same design signals, which is reasonable because the document is written in natural language and there is no strict format requirement to the document. Also, a word may map to multiple design

signals with different prefix or postfix in the signals names. These facts explain why the number of words having signal mapping is different from the number of signals of the mapping result. The signal extraction results is shown in Table 5.1.

Table 5.1: Text mining results

# words after text mining	66
# words having signal mapping	42
# signals of the mapping result	46

5.4.2 Rule Learning Based on the Extracted Features

We implemented our learning methods in Python using Pandas [39] and scikit-learn [40]. Pandas was used to process data, and scikit-learn provided us the implementation of CART.

There are 300 features in total after running the data processing procedure. The number of training samples for rule learning is 9216.

Fig. 5.6 shows the rule learning results based on all the signals extracted from documents. The y-axis is the hit rate, and the x-axis is the assertion coverage point sorted by its hit rate after learning. Each assertion coverage point has two bars. The orange bar is the hit rate after learning, and the blue bar is the hit rate of random test generation. The results show that for more than 60% of the assertion coverage points, 100%-accurate rules can be learned. The result implies that given an assertion coverage point, with chances over 50%, an engineer without much design knowledge can obtain accurate rules.

To overcome the overfitting problem, we ran the rule learning algorithms again with smaller groups of features. For each assertion coverage point that has 100%-accurate rules, a signal group is created comprising all the features in the corresponding decision tree. Then the rule learning algorithm is applied with each signal group.

Fig. 5.7 shows the learning results on the assertion coverage points that have no

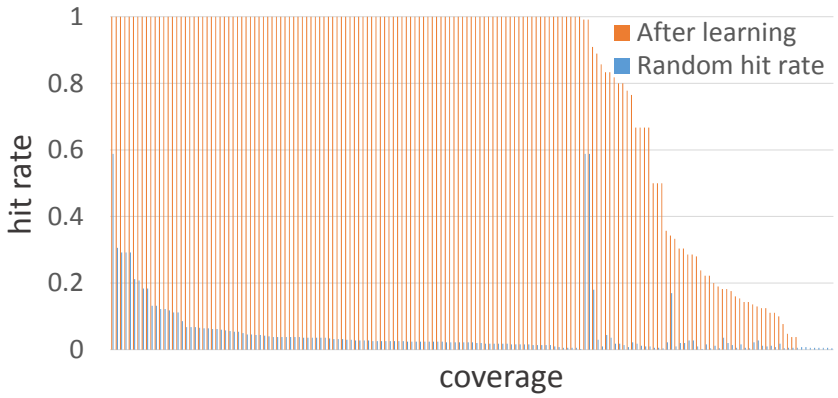

Figure 5.6: Hit rate improvement between random and rule learning results of all the assertion coverage points.

100%-accurate rule at the previous stage. 38 out of 58 assertion coverage points have hit rate improvement. In addition, 11 assertion coverage points have 100%-accurate rules.

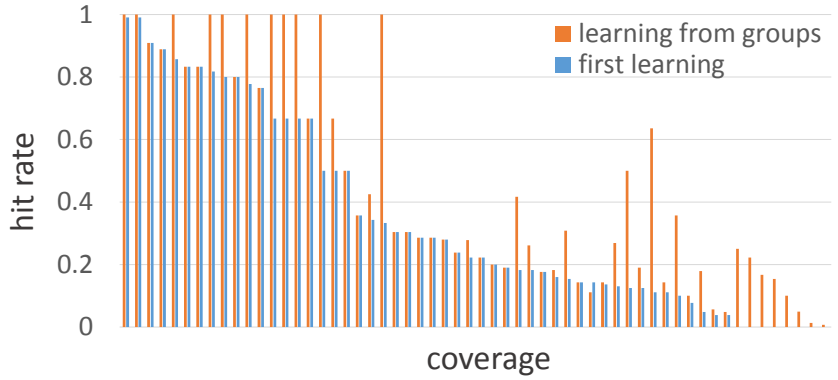

Figure 5.7: Hit rate improvement after learning from signal groups.

Table 5.2 shows the hit rate improvement of selected assertion coverage points with very low random hit rate. It clearly demonstrates the overfitting phenomena: when data is insufficient and the number of features is large, the learning result cannot be generalized, thus we have rules with 0 hit rate when learning with all the signals from documents. However, if we can exclude irrelevant features, the learning works with the same amount of data. This explains why we got hit rate improvement by learning from smaller groups.

Table 5.2: Hit rate improvement between learning from all the extracted signals and from signal groups

	random	all doc signals	signal groups
Assertion 1	1/500	0%	25%
Assertion 2	1/500	0%	22%
Assertion 3	1/500	0%	17%
Assertion 4	2/500	0%	15%

Table 5.3 summarizes the overall results of our experiments. With the signals extracted from documents, we obtained 100%-accurate rules for more than 70% of the assertion coverage.

Table 5.3: The percentage of the assertion coverage having 100%-accurate rules

random	all doc signals	signal groups
0.0%	64.9%	71.4%

The result suggests that the set of signals extracted from documents provides a good starting point for engineers who do not have deep knowledge of the design under verification. If the goal is to find accurate rules for all the assertion coverage points, the proposed approach ramps up from nothing to 70% without much effort. On the other hand, rules not 100%-accurate also provide information for engineers to analyze designs and tests, and assist in applications such as generating more tests and debugging.

5.5 Summary

Text mining methods are applied to obtain words likely to be the names of design signals. These words are then mapped to the real design signals to create the feature set. The effectiveness of these features is experimented by learning rules for improving assertion coverage. The experimental result shows that for more than 70% assertions, 100%-accurate rules can be obtained. Also, experimental result shows that using a smaller

set of signals can lead to a better rule learning result and suggests that a set of signals able to infer a coverage point may be able to infer another coverage point.

Chapter 6

Boolean Concept Learning

6.1 Introduction

The problem definition of Boolean concept learning is as follows. Suppose there is an unknown Boolean concept, i.e. a Boolean function, $h(x_1, x_2, \ldots, x_n) \rightarrow \{0, 1\}$, where n is the number of input variables. Given m samples and each sample is represented by $(\vec{x}, y)$, where $\vec{x} \in \{0, 1\}^n$ is an input assignment and $y = h(\vec{x})$. A sample is *positive* if its y value is 1. A sample is negative if is y value is 0. The goal of Boolean concept learning is to find the unknown function h by analyzing the given samples. This function h is also called a *rule* that can distinguish the positive samples and negative samples.

In the proposed verification system, Boolean concept learning is used in two tasks: (1) to learn the relationship between a coverage event and features, and (2) to learn the relation between features and test parameters, as depicted in Figure 6.1. For the first task, given a coverage event, the goal is to find a concept, which is composed of values of important signals, to infer the activation of the coverage event. In this task, a positive sample refers to an occurrence of the coverage event. For the second task, given a combination of signal values, the objective is to find a concept comprising test

parameters to infer the occurrence of the combination of signal values. In this task, a positive sample refers to an occurrence of the combination of signal values. Then, new tests can be generated by the test generator with the learned test parameters. Ideally, the generated tests can create the combination of signals values and hit the coverage event.

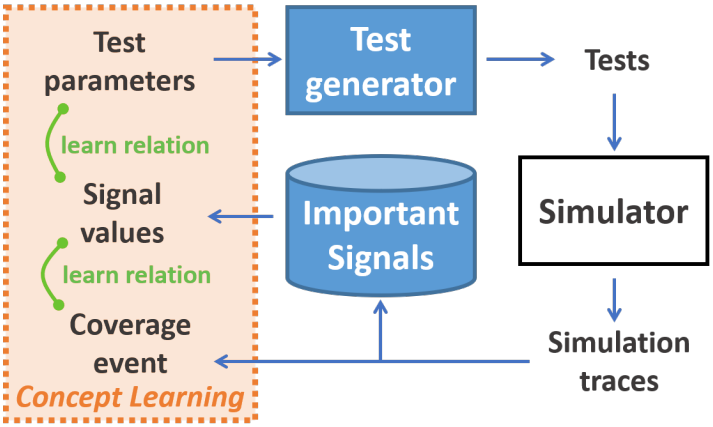

Figure 6.1: Concept learning in the proposed system

In the proposed verification system, it is assumed that an unknown concept is a k-term DNF with k less than a pre-defined number. The reason is that each term in a DNF is a cause of the positive samples and from our experience the number of causes is a small number.

For concept learning application in functional application, it is common that the data in the concept learning task is extremely imbalanced – only one or few positive samples and thousands of negative samples. Handling extremely imbalanced data is a challenge and traditional concept learning methods do not handle it well.

In this chapter, the problem of learning with extremely imbalanced data is described first. Then the occam's razor assumption and the uniqueness requirement is discussed. Last, based on the uniqueness requirement, a new concept learning tool, Version Space Cardinality based Concept Learning (VeSC-CoL), is proposed to deal with the extremely

imbalanced data.

6.2 Problems with Extremely Imbalanced Data

This section discusses two main problems of learning with extremely imbalanced data. First, there is no validation data and test data due to data limitation of positive samples. Hence, common machine learning techniques that uses validation data to access the validity of a learned model cannot be applied. Second, for decision tree based concept learning methods, it is impossible to learn some k-term DNF with extremely imbalanced data.

6.2.1 No Validation Data

Validation Data is commonly used to assess whether there is overfitting or not. Some new tests will be generated based on the learned concept. Then simulation is required to know whether the target coverage is covered or not. If it is not covered, which means the learned concept is incorrect, then another iteration of running rule learning method, generating tests and simulation is required.

There problem is that simulation takes time. Suppose there are 100 tests generated and each tests requires 5-minute cpu time for simulation, then each iteration takes 500-minute cpu time for simulation.

Also, in each iteration, it is not clear how to improve the learning result. Is the problem in the first learning task or second learning task? Should I obtaining more data or refining features. Typically the decision is heavily based on one's experience about the learning algorithm and the design under verification.

6.2.2 Inability to Learn k-term DNF

For a decision tree based concept learning method with limited positive samples, there are situations that it is impossible to learn a k-term DNF formula. Suppose the target concept is a 2-term DNF, say $h = ab + cd$, and there are only two positive samples available. Without loss of generality, suppose the method selects a to be the first feature for splitting. There are two cases: (1) The two positive samples are in the True branch and (2) One positive sample is in the true branch and the other positive sample is in the false branch. For the first case, the learned result is $h_1 = ag$ for some function g and clearly each term in h_1 contains a that cannot be simplified. However the term cd doesn't contains a and thus h_1 never equals h. For case two, The learned result is $h_2 = ag_1 + a'g_2$. To make $h_2 = h$, g_1 must be functionally equal to $b + cd$ and g_2 must be functionally equal to cd. However to obtain this result, at least three positive samples are required, two positive samples for g_1 and one positive samples for g_2. Thus h_2 is not equal to h. This example shows that with limited positive samples, decision tree is not capable of learning some k-term DNF.

6.3 The Uniqueness Requirement

Without validation data, we still need some performance measurement of the learning result to reduce the number of iterations in the concept learning process. Let's first discuss the advantages provided by validation data and what is the cause of such advantages. The main advantage of using validation data is to detect overfitting – when a learned model is too specific, it looses its generalization ability and performs badly on unseen data. However, is using validation data the only way to detect overfitting?

The Occam's razor principle states that if there are two models with the same performance on the training samples, then the simpler one is preferred. In machine learning

community, though there is no proof, it is usually assumed and accepted that a simpler model has better generalization ability [41]. Another result from Structural Risk Minimization [42] also states the same idea – simpler models are preferred. Follow these results, we prefer a concept learning algorithm that is guaranteed to find a simplest model.

However, there are cases that finding a simplest model is not enough. Consider a case that there are 1000 simplest concepts fit training samples. If the only requirement is to get a simplest model that fits the data, then any one of the 1000 concepts can be the learned concept. Intuitively, people are more confident to the result if there is an unique answer. If there are 1000 alternatives, it is an indicator of data insufficiency. This uniqueness requirement is discussed in [43].

However, in practice, simply checking the uniqueness is still not enough. It is possible that it is just by chance to have an simplest and unique concept that fits data. The idea of reducing the chance of getting false learning result is to test the learned concept with more negative samples (recall that positive samples are rare but there are a lot of negative samples). If a learned concept is not the unknown concept, the probability that the learned concept does not agree with a new randomly generated sample can be calculated, as shown in the next paragraph. Hence, an idea is to test a learned concept with new randomly generated samples.

Given an unknown target concept h, a concept g and m uniformly random-generated negative samples. Let P_{agree} be the probability that g agrees with the m negative samples. Then

$$P_{agree} = (1 - p_r)^m, \tag{6.1}$$

where p_r is the probability that g does not agree with a uniformly random-generated

negative sample. p_r can be calculated by

$$p_r = N_{minterms}(g \wedge \neg h)/N_{minterms}(\neg h), \tag{6.2}$$

where $N_{minterms}(g \wedge \neg h)$ is the number of minterms of $g \wedge \neg h$ and $N_{minterms}(\neg h)$ is the number of minterms of $\neg h$. The explanation of equation (6.2) is that (a) if a sample is negative, it must be a satisfiable assignment of $\neg h$ and (b) if g does not agree with the sample, it must be a satisfiable assignment of g. Note that p_r depends on both h and $g \wedge \neg h$ hence there is no closed form independent to g or h.

Follow the discussion above, we want a method to find a simplest concept and check its uniqueness, then test the concept with some number of negative samples.

6.4 VeSC-CoL Overview

In this work, VeSC-CoL adopts a particular learning strategy enabled by the capability to calculate version space cardinality. In VeSC-CoL, the complexity measurement of a hypothesis h is defined as the number of literals in h. The whole hypothesis space, i.e. k-term DNF with limited k, is then partitioned into hypothesis sub-spaces, where two hypotheses have the same complexity if and only if they are in the same hypothesis sub-space. VeSC-CoL then tries to find the simplest hypothesis. The search proceeds from the lowest-complexity sub-space to highest-complexity sub-space. The search process stops when it first finds a hypothesis that fits the data. The partition and search process is illustrated in Figure 6.2.

For example, suppose the hypothesis space is k-term DNF and $k \leq 3$. The simplest hypothesis sub-space has all hypotheses with 1 literal, e.g. a, b', c. The second simplest hypothesis sub-space has all hypotheses with 2 literals, e.g. $a'b$, $a + b$. The third

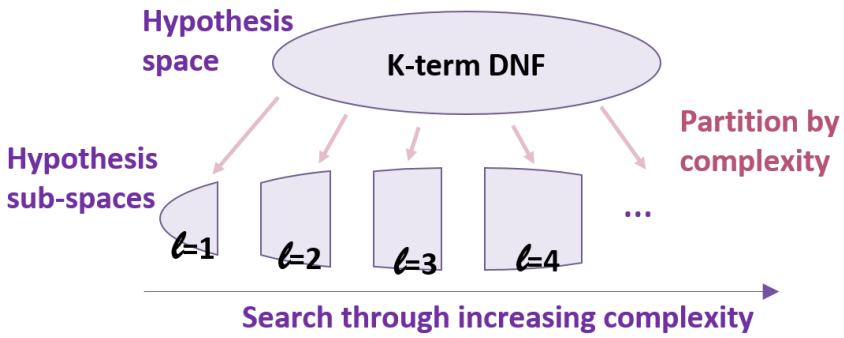

Figure 6.2: Hypothesis space partition by the number of literals

hypothesis sub-space has all hypotheses with 3 literals, e.g. abc, $ac + b'$, $a + b + c'$.

6.4.1 VeSC-CoL FLOW

Figure 6.3 depicts the flow of VeSC-CoL. It starts with calculating the version space cardinality of the simplest hypothesis sub-space. If there is no hypothesis consistent with the data, i.e. $|VS| = 0$, then VeSC-CoL moves onto the next hypothesis sub-space. The iteration stops when it first finds a non-empty version space. At this point, VeSC-CoL reports at most B hypotheses in version space as well as version space cardinality as a measure of learning quality, where B is an application-specific parameter and usually is set to the maximum number of hypotheses that a user can handle in model evaluation.

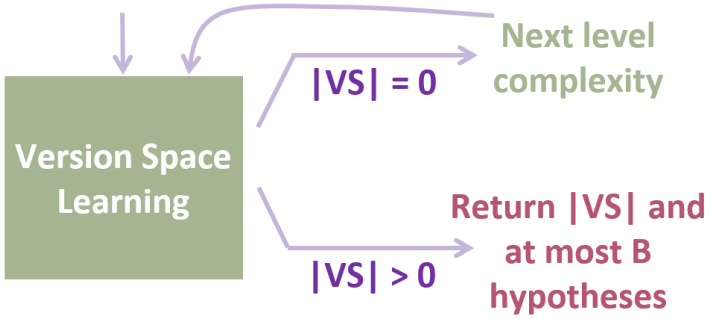

Figure 6.3: Illstration of VeSC-CoL flow

In theory, this problem is proved to be no easier than #P-complete [44]. Even deter-

mining whether a version space is empty or not for k-term DNF is NP-hard [45].

Though this problem is intractable in theory, in practice a useful tool can still be developed. In this paper, we propose two methods to calculate version space cardinality. The first method is based on Ordered Binary Decision Diagram (BDD) [17]. The version space cardinality can be obtained by calculating the number of minterms in BDD. The second method is based on Boolean Satisfiability (SAT). For the SAT implementation, VeSC-CoL does not calculate version space cardinality. Rather, VeSC-CoL tries to find at most $B + 1$ hypotheses in the version space.

6.5 Calculating Version Space

6.6 BDD-Based Implementation

Given the maximum number of terms k_{max}, to calculate the version space of a sub-space H_l that have all l-literal hypotheses, k_{max} BDD runs will be executed. The i_th run calculates the fitting hypotheses with exactly i terms and l literals. The result version space of cardinality the sub-space can then be obtained by the summation of the number of fitting hypotheses in each run.

6.6.1 Representation of A Subset

The idea of using BDD to calculate version space cardinality starts with using BDD to represent a subset of a given finite set. In our case, the finite set is the set of all hypotheses in a given hypothesis sub-space. A subset is the version space after seeing the input samples. This subset is revised as more input samples are provided. The following three points are the basic idea using BDD to represent a subset of a given finite set.

- Generally speaking, a BDD represents a Boolean function, where a Boolean function is defined as $f : \{0,1\}^n \rightarrow \{0,1\}$.

- Given a finite set S, let T be a subset of S. We use a Boolean function to represent T.

- The representation is the following. First, we use a function g to map elements of S to points in a Boolean space. Let $g : S \rightarrow \{0,1\}^n$ be an injection (because S is finite, with large n, g exists). Then, to represent T in the Boolean space, we use a function $f_T : \{0,1\}^n \rightarrow \{0,1\}$ where $f_T(x) = 1$ iff $g^{-1}(x)$ exists and $g^{-1}(x) \in S$.

The last point indicates that the cardinality of T is equal to the number of minterms in f_T.

In the implementation of VeSC-CoL, S is the set of all k-term DNF formulas with given k. An element in S is a hypothesis, i.e. a k-term DNF formula. Also, note that for a given hypothesis sub-space constrained by the number of literals l, this constraint needs to be represented by another BDD C. Then, the AND operation of the BDDs S and C will represent the desired hypothesis sub-space.

The mapping g is described in Section 6.6.3 and 6.6.4. Computing version space is done by performing intersection on several subsets of S, i.e. performing AND operation on several BDDs.

6.6.2 High-level Idea

The basic idea of the proposed BDD-based version space learning method is based on set intersection [46]. Figure 6.4 illustrates this idea.

The calculation starts with creating a set of hypotheses with a given complexity, which is the hypothesis sub-space. Then, each sample is converted into the set of hypotheses

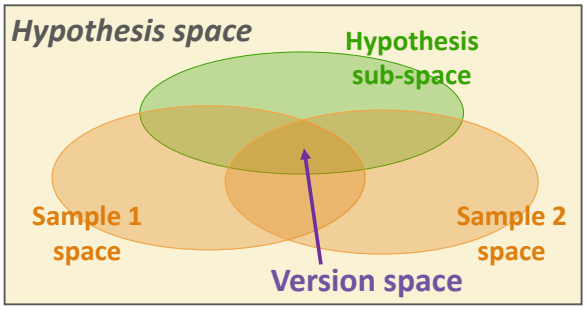

Figure 6.4: Version space learning by set intersection

that agree with the sample. The version space can be obtained by intersecting the hypothesis sub-space and all the sample spaces.

Set intersection can be performed via Boolean AND operation of BDDs. To determine the size of version space, we can simply calculate the number of minterms in the version space BDD. Note that if a hypothesis has multiple representations, special treatment is required and it is discussed in Section 6.6.8.1.

In the following sections, k is the number of terms and l is the number of literals.

6.6.3 Idea of Encoding

Given the number of features n and the number of terms k. Let x_i^j represent the status of the i-th feature in the j-th term, wherein $x_i^j \in \{\text{neg, pos, dcare}\}$, which denotes appearing in negative form, in positive form and don't care (not appearing). Since x_i^j is a three-value variable, we use two Boolean variables to represent it in BDD. An example is shown in Figure 6.5. In sum, there are $2nk$ variables in BDD.

6.6.4 Base Hypothesis Space Encoding

Algorithm 2 shows the method to create a BDD representing an n-feature and k-term DNF hypothesis space. This algorithm simply forces each x_i^j to be in its three possible

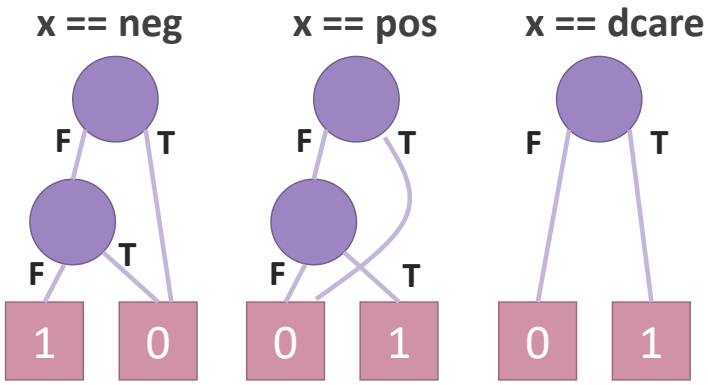

Figure 6.5: Using BDD to encode a three-value variable

values. Note that there is a bijection between a satisfiable assignment in the returned BDD and a k-term DNF representation.

Algorithm 6.1: Creating a BDD representing a n-variable, k-term DNF space

Input: Integers n, k
Output: BDD dd

```
1 dd ← BDD_One();
2 for j ∈ {1, 2, ..., k} do
3 |   for i ∈ {1, 2, ..., n} do
4 |   |   tmp_dd = BDD_Or(x_i^j == pos, x_i^j == neg, x_i^j == dcare);
5 |   |   dd ← BDD_And(dd, tmp_dd);
6 |   end
7 end
8 return dd;
```

6.6.5 Hypothesis Sub-space Encoding

Algorithm 3 describes the method to create a BDD representing an n-feature, k-term DNF, l-literal space. Essentially it creates a length constraint that exactly l literals are there in its DNF representation. Dynamic programming is used to reduce its space usage. The state variable lit_dd[w] is a function f, i.e a BDD, such that $f = 1$ if and only if there are at least w literals up to the current iteration.

Algorithm 6.2: Creating a BDD representing a n-variable, k-term, l-literal space

```
   Input: Integers n, k, l
   Output: BDD dd
 1 lit_dd[0] ← BDD_One();
 2 for w ∈ {1, 2, ..., l + 1} do
 3 |  lit_dd[w] ← BDD_Zero();
 4 end
 5 for j ∈ {1, 2, ..., k} do
 6 |  for i ∈ {1, 2, ..., n} do
 7 |  |  var_dd ← BDD_Or(x_i^j == pos, x_i^j == neg);
 8 |  |  for w ∈ {l + 1, l, ..., 1} do
 9 |  |  |  tmp_dd ← BDD_And(lit_dd[w − 1], var_dd);
10 |  |  |  lit_dd[w] ← BDD_Or(lit_dd[w], tmp_dd);
11 |  |  end
12 |  end
13 end
14 dd ← BDD_And(lit_dd[l], BDD_Not(lit_dd[l + 1]));
15 return dd;
```

Line 1 to Line 4 is the initialization process for dynamic programming. Line 5 to Line 13 updates the state variable, i.e. lit_dd in dynamic programming. At line 10, an assignment that makes lit_dd[w] == 1 can be mapped to a DNF formula with at least w literals in the processed x_i^j. The outer two loops iterate all x_i^j, then for each x_i^j, lit_dd is updated accordingly. Note that lit_dd is updated from the highest index to the lowest index because the current lit_dd[w] depends on the lit_dd[$w-1$] in the previous iteration. After processing all the x_i^j, line 14 creates the result BDD representing the l-literal sub-space.

Note that the returned BDD does not guarantee a bijection between a minterm and a DNF representation due to the NOT operation. To have this bijection, a minterm in the returned BDD must be in the base hypothesis space BDD as well. Hence the sub-space BDD can be obtained by the AND of the returned BDDs of Algorithm 2 and Algorithm 3. Also, with proper BDD variable ordering, it can be shown that the complexity of

Algorithm 3 is $\Theta(nkl)$.

6.6.6 Positive Sample Space Encoding

Algorithm 4 converts a positive sample to a BDD representing a set of consistent hypotheses. Its input parameters are n, the number of features, k, the number of terms, and $s[i] \in \{0, 1\}$, the value of the i-th feature. The key idea is in line 12, given a hypothesis, at least one term of the hypothesis must be evaluated as true so the hypothesis is evaluated as true.

Algorithm 6.3: Converting a positive sample to BDD

```
   Input: Integers n, k, and an n-dimensional Boolean vector s
   Output: BDD dd
 1 dd ← BDD_Zero();
 2 for j ∈ {1, 2, ..., k} do
 3 |   term_dd ← BDD_One();
 4 |   for i ∈ {1, 2, ..., n} do
 5 |   |   if s[i] == 0 then
 6 |   |   |   tmp_dd ← BDD_Or(x_i^j == neg, x_i^j == dcare);
 7 |   |   else
 8 |   |   |   tmp_dd ← BDD_Or(x_i^j == pos, x_i^j == dcare);
 9 |   |   end
10 |   |   term_dd ← BDD_And(term_dd, tmp_dd);
11 |   end
12 |   dd ← BDD_Or(dd, term_dd);
13 end
14 return dd;
```

Suppose $s = 101$. For a single term to be evaluated as true, feature 1 and feature 3 must not be negative literals and feature 2 must not be a positive literal in the term. Otherwise, this term is evaluated as false. The generalization of this idea shown in line 3 to line 11.

At line 10, each minterm in term_dd can be mapped to a DNF term that fits the positive sample. At line 14, each minterm in dd can be mapped to a k-term DNF formula

that fits the postive sample. With proper BDD variable ordering, the complexity of this algorithm 4 is $O(kn)$.

6.6.7 Negative Sample Space Encoding

The algorithm of converting a negative sample to its space BDD is similar to algorithm 4. The differences are (1) all the terms must be evaluated as false and (2) the conversion rule for a single term is negated. Algorithm 5 shows the conversion algorithm.

Algorithm 6.4: Converting a negative sample to BDD

Input: Integers n, k, and an n-dimensional Boolean vector s
Output: BDD dd

```
1  dd ← BDD_One();
2  for j ∈ {1, 2, ..., k} do
3  |   term_dd ← BDD_Zero();
4  |   for i ∈ {1, 2, ..., n} do
5  |   |   if s[i] == 0 then
6  |   |   |   tmp_dd ← (x_i^j == pos);
7  |   |   else
8  |   |   |   tmp_dd ← (x_i^j == neg);
9  |   |   end
10 |   |   term_dd ← BDD_Or(term_dd, tmp_dd);
11 |   end
12 |   dd ← BDD_And(dd, term_dd);
13 end
14 return dd;
```

Again, suppose $s = 101$. For a single term to be evaluated as false, one of the following conditions must hold: at least one of feature 1 and feature 3 appears as a negative literal, or feature 2 appears as a positive literal. The generalization of this idea shown in line 3 to line 11. At line 12, since all the terms must be evaluated as false, the result dd is the AND of all the term_dd. With proper BDD variable ordering, the complexity of this algorithm 5 is $O(kn)$.

6.6.8 Obtaining Version Space

Version space can be obtained by performing an AND of all the above BDDs. Recall that each BDD represents a set of hypotheses inside the hypothesis sub-space, which agree with a positive sample or a negative sample. The AND is equivalent to the set intersection operation so the resulting BDD represents the version space.

In actual implementation, the AND of a set of BDDs is accomplished by performing a sequence of AND operations on two BDDs. We observed that the ordering of AND operations on BDDs significantly influences the runtime. There can be two prferences: (1) Process the hypothesis sub-space BDD first and (2) If $k \leq 2$, process positive sample BDDs before negative sample BBDs; otherwise process negative sample BDDs before positive sample BDDs.

To illustrate the first preference, Figure 6.6 shows the number of BDD nodes in the version space BDD versus the number of processed samples. There are 100 features, 3 positive samples, and 800 negative samples. For the red line, the first AND operation is applied to the hypothesis sub-space BDD and a positive sample BDD. The next two ANDs involve the remaining two positive sample BDDs. The negative sample BBDs are processed afterward. For the green line, the first three positive sample BDDs are processed first, followed by processing negative sample BDDs. The hypothesis sub-space BDD is processed last.

The runtime is proportional to the number of BDD nodes. It can be clearly observed that the difference in runtime between the two cases is significant. The reason is that processing the hypothesis sub-space BDD first can more effectively trim the version space.

Table 6.1 shows the runtime comparison between processing positive sample BDDs first and processing negative sample BDDs first. The comparison is presented as a ratio between the two. In each case, there are 100 features, 250 positive samples, and 250

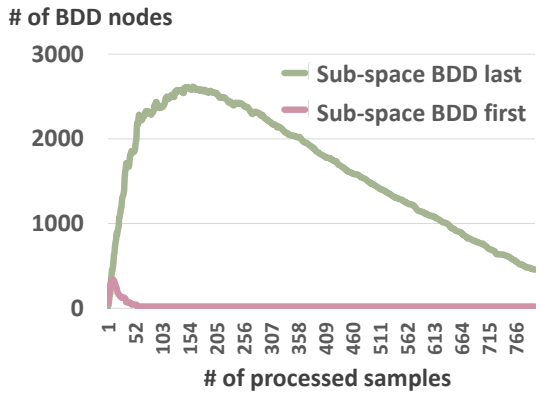

Figure 6.6: Example to illustrate that processing hypothesis sub-space BDD at the beginning is more efficient

negative negative samples. The number of literals l is randomly selected in each run and $l \leq 15$. In each case, there are 10 runs for the positive first and 10 runs for the negative first. A geometric mean of the 10 runtimes is calculated. Then, the ratio is calculated from the two geometric means. The reason to use the geometric mean is that the 10 runtimes can differ significantly based on the selection of l. Table 6.1 shows that for $k \leq 2$, processing positive sample BDDs before negative sample BDDs saves time, and vice versa.

Table 6.1: Runtime ratio of processing positive sample BDDs first over processing negative sample BDDs first

k	**pos-first/neg-first**
1	$1.89 * 10^{-6}$
2	$8.70 * 10^{-2}$
3	$1.76 * 10^{3}$
4	$2.38 * 10^{5}$

For $k = 1$, the problem is monomial learning. For monomial learning, it is well known that positive samples are far more important than negative samples, i.e. positive samples are far more effective to reduce the version space than negative samples. As a result, processing positive sample BDDs first is more effective. This property seems to somewhat carry over to the case $k = 2$. It is interesting that the situation reverses for

$k = 3$ and $k = 4$. The theoretical reason for this reverse is still unclear and should be investigated further in the future.

6.6.8.1 Handling Non-Canonicality

A hypothesis can be represented by different DNF formulas, e.g. $a+b = b+a$. Hence the size of version space cannot be obtained by counting the number of minterms in the version space BDD in general. Here we introduce another BDD that forces each term in a DNF representation to be in lexicographical order, which reduces the permutation among terms. Algorithm 6 shows the procedure to create a BDD having lexicographical order among two terms. In total $k-1$ such BDDs are required. Next, when the number of minterms in the version space BDD is in the same order as B, we convert each minterm to its DNF formula and then use a BDD to represent it. Since BDD is a canonical representation, we are able to obtain the size of version space.

Algorithm 6.5: BDD representing the k_1-th term is lexicographically smaller than the k_2-th term

Input: Integers n, k_1, k_2
Output: BDD dd
1 $dd \leftarrow$ BDD_Zero();
2 eq_dd $\leftarrow$ BDD_One();
3 **for** $i \in \{1, 2, \ldots, n\}$ **do**
4 cond1 $\leftarrow$ BDD_And($x_i^{k_1} == neg$, $x_i^{k_2} == pos$);
5 cond2 $\leftarrow$ BDD_And($x_i^{k_1} == neg$, $x_i^{k_2} == dcare$);
6 cond3 $\leftarrow$ BDD_And($x_i^{k_1} == pos$, $x_i^{k_2} == dcare$);
7 cond = BDD_Or(cond1, cond2, cond3);
8 tmp_dd $\leftarrow$ BDD_And(eq_dd, cond);
9 $dd \leftarrow$ BDD_Or(dd, tmp_dd);
10 eq_dd $\leftarrow$ BDD_And(eq_dd, $x_i^{k_1} == x_i^{k_2}$);
11 **end**
12 **return** dd;

6.7 SAT-Based Implementation

The idea of SAT-based version space learning is similar to BDD-based encoding. The basic components are the same: the hypothesis sub-space, the positive sample spaces, and the negative sample spaces. Let n be the number of features, l be the number of literals, k be the number of terms, m_p be the number of positive samples and m_n be the number of negative samples. The proposed encoding method results in a CNF formula with $\Theta(nkl + km_p)$ symbols and $\Theta(nkl + nkm_p + km_n)$ clauses.

6.7.1 Base Hypothesis Space Encoding

Same as BDD encoding, each feature can appear in positive, negative or does not appear in a term. Hence, three symbols are used to represent each case.

- $X^j_{i,1}$ is True iff the i-th feature in the j-th term appears in negative form
- $X^j_{i,2}$ is True iff the i-th feature in the j-th term appears in positive form
- $X^j_{i,3}$ is True iff the i-th feature in the j-th term does not appear

Since exactly one of the three cases is true, one-hot constraints are required to enforce the requirement:

$$\Pi^k_{j=1}\Pi^n_{i=1}(X^j_{i,1} + X^j_{i,2} + X^j_{i,3})(\neg X^j_{i,1} + \neg X^j_{i,2})$$
$$(\neg X^j_{i,1} + \neg X^j_{i,3})(\neg X^j_{i,2} + \neg X^j_{i,3}).$$

6.7.2 Hypothesis Sub-space Encoding

For a given (l, k), we need to constrain the space to contain only l literals. It is the cardinality constraint. The performance of different encoding methods for a cardinality constraint can be found in [47]. In our implementation, we choose the sequential counter

method [48] because its performance is comparable to other encoding methods and it has the unit propagation property [47]. The encoding formula shown in [48] cannot be used directly because it is for cardinality $\leq l$. A straightforward modification is used for cardinality $= l$, based on converting the sequential counter circuit to SAT clauses. The encoding for the cardinality constraint requires additional $l(nk-1)$ new symbols and $\Theta(nkl)$ clauses.

For this encoding, we use the same notation and symbol in [48], so it is easier to get the difference between the modification and the original encoding, wherein k is the number of symbols passing to the cardinality constraint and $S_{i,j}$ are additional symbols.

$$(x_1 + \neg S_{1,1})(\neg x_1 + S_{1,1}),$$
$$\Pi_{j=2}^{k}(\neg S_{1,j}),$$
$$\Pi_{i=2}^{n}(\neg x_i + \neg S_{i-1,k}),$$
$$\Pi_{i=2}^{n-1}(x_i + S_{i-1,1} + \neg S_{i,1})(\neg x_i + S_{i,1})(\neg S_{i-1,1} + S_{i,1}),$$
$$\Pi_{i=2}^{n-1}\Pi_{j=2}^{k}(x_i + S_{i-1,j} + \neg S_{i,j})(S_{i-1,j-1} + S_{i-1,j} + \neg S_{i,j})(\neg S_{i-1,j} + S_{i,j})(\neg x_i + \neg S_{i-1,j-1} + S_{i,j}),$$
$$(x_n + S_{n-1,k})(S_{n-1,k-1} + S_{n-1,k}).$$

6.7.3 Positive Sample Space Encoding

Again, given a positive sample $s = 101$. For a single term to be evaluated as true, feature 1 and feature 3 must not appear in negative form and feature 2 must not appear in positive form. Then, at least one term must be evaluated as true. A naive encoding leads to n^k clauses, which is not feasible. To overcome this challenge, additional k symbols, A^1, A^2, ..., A^k, are used such that A^j is true if and only if the j-th term is evaluated as true. With these additional symbols, the number of clauses reduces to $(n+1)k+1$. The requirement of at least one term is evaluated as true is encoded by a single clause:

$$(\Sigma_{j=1}^{k} A^j),$$

and for each j, the relation of A^j and $X^j_{i,\delta}$ is maintained by

$$\Pi_{i=1}^{n}(\neg X^j_{i,2-s[i]} + \neg A^j), \text{ and}$$
$$(\Sigma_{i=1}^{n} X^j_{i,2-s[i]} + A^j).$$

6.7.4 Negative Sample Space Encoding

Given a negative sample $s = 101$. For a single term to be evaluated as false, at least one of feature 1 and feature 3 must appear in negative form or feature 2 appear in positive form. Besides, all the terms must be evaluated as false. For each sample, k clauses are required and each clause encodes that a term is evaluated as false. The overall encoding is

$$\Pi_{j=1}^{k}(\Sigma_{i=1}^{n} X^j_{i,2-s[i]}).$$

6.7.5 Size of Version Space

Each satisfiable assignment in the above SAT problem can be mapped to a DNF formula. The size of version space can be obtained by counting the number of satisfiable assignments. A common approach is to add new clauses to remove previous satisfiable assignments and then call the SAT solver again. Removing a satisfiable assignment is a standard approach and omitted here. Note that we use the same approach to deal with the non-canonicality problem described in BDD-based learning, except the lexicographical order constraint is represented by SAT clauses.

6.8 Experimental Results

We use CUDD-3.0.0 [49] to implement the BDD-based learning and use Lingeling [50] for the SAT-based learning. The dynamic variable re-ordering option in CUDD is

disabled to facilitate the study of various aspects of the tool performance.

6.8.1 Effect of Uniqueness

In the experiment, we assume the number of features $n = 100$. We further assume the length of the true answer is 5 which can be a k-term DNF formula for $k = 1, 2, 3$. In each case, a k is randomly picked and the true answer is also randomly picked from all the k-term DNF hypotheses. The dataset contains exactly k positive samples which is randomly generated. Then, negative samples are also randomly generated.

Figure 6.7 shows a comparison result for VeSC-CoL and CART [12]. The experiment includes 100 cases. The x-axis shows the number of negative samples used in the learning up to that particular point. In the experiment, the k positive samples are always used first, before any negative samples are used. The maximum number of negative samples is 5000.

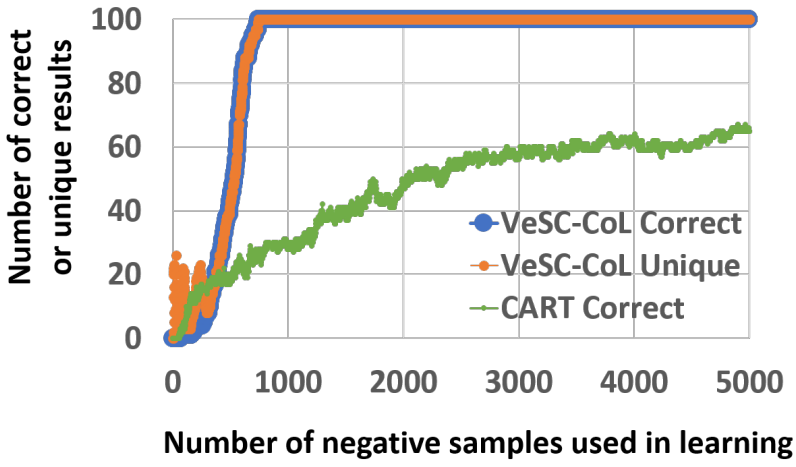

Figure 6.7: Correctness and uniqueness, comparing to CART

For CART, the figure shows the number of cases the CART tool correctly reports the true answer at each x value. For VeSC-CoL, the figure shows two numbers at each x value. The first is the number of cases where VeSC-CoL reports a unique hypothesis

as its answer. This is marked as a orange dot. The second is the number of cases where VeSC-CoL correctly finds the true answer. This is marked as a blue dot. Where these two numbers coincide, the figure shows an overlap of orange dots with blue background.

For CART, the best scenario is that it finds the true answer in 67 out of the 100 cases. This happens occasionally on some particular x values after 4899. For the range of the x values shown, CART has a trend that as more negative samples are provided it performs better, but its performance fluctuates quite frequently.

For VeSC-CoL, notice that a unique hypothesis found by the tool does not always guarantee it is the correct answer. However, this happens only when the x value is still relatively small. More interestingly unlike CART has performance fluctuation as x increases, VeSC-CoL Correct is monotonically increasing to x. After $x = 1000$, VeSC-CoL finds all the 100 true answers and the result does not change with more negative samples added.

Figure 6.8 presents the result of VeSC-CoL from a different perspective and focuses on x value up to 250 and randomly picked 20 cases from the previous experiment. For each of the y label from 1 to 20, the figure marks the x values where VeSC-CoL finds a unique hypothesis but it is not the true answer. We can call each range of such x values a mistake window. Figure 6.8 shows where such mistake windows occur as the number of samples increases.

In the figure, for each case there are at most 4 continuous windows because the length of unknown targets is 5. For some cases the number of continuous windows is less than 4 because a hypothesis space may go from not-unique to empty as a negative sample is added.

The largest mistake window in figure 6.8 is 48, which means if a unique but incorrect result is obtained, it will be removed within 48 new negative samples. More experiments were conducted for target $l \leq 6$, $k = 1, 2, 3$ and for each l there were 100 cases. The

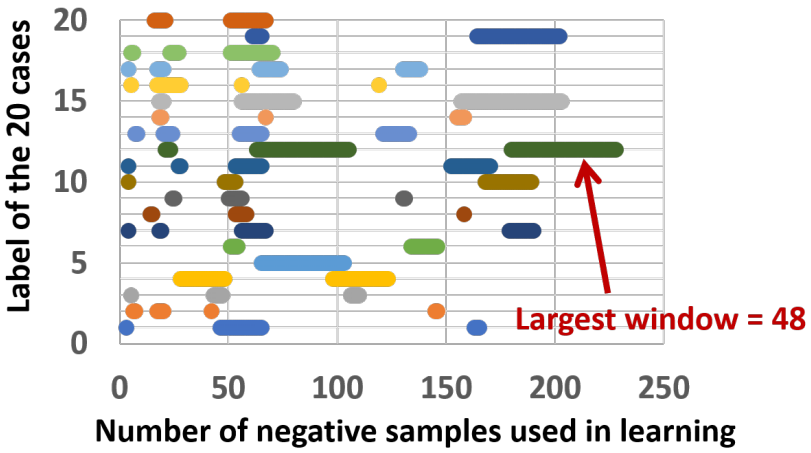

Figure 6.8: Window of unique and incorrect result

largest mistake window was less than 100.

6.8.2 Runtime Comparison between BDD and SAT

We observed different characteristics of runtime between the SAT-based method and the BDD-based method. For example, we use a simple experiment to illustrate their differences. In this experiment, the target concept is assumed to be a 5-literal monomial (i.e. $k = 1$ and $l = 5$). There are 1000 randomly generated negative samples. There can be 0, 1, and 2 positive samples. Figure 6.10 and Figure 6.9 show the runtime results. Each point is the average of runtimes over 10 runs. The size bound B is set to 1, so if the size of version space is large than 1, the SAT-based learning would stop after finding the second fitting hypothesis. Note that in this experiment, for all cases with $l < 5$ the calculated size of version space is always 0, for $l = 5$ the calculated size of version space is exactly 1, and for $l > 5$ the calculated size of version space is always larger than 1. This shows that both learning methods can identify the correct hypothesis sub-space and the correct hypothesis.

The results show that the runtime of the BDD-based method is exponential to l. On

the other hand, the SAT-based method has a peak runtime at $l = 5$, i.e. the size of version space is 1. Figure 6.11 shows another interesting property of BDD-based learning where the number of positive sample is 1. In each case, the positive sample BDD is processed first, followed by processing the negative sample BDDs. The figure shows the number of BDD nodes as a function of the number of processed samples. As it can be observed, the peak number of BDD nodes occurs earlier in the process than later, for example within the first 200 samples. This implies that the computational limitation occurs within the processing of the first 200 samples. As a result, it is not the case that a larger dataset implies a longer run time. As mentioned above, the deciding factor for the runtime is the length l.

Figure 6.12 shows similar runtime results as those shown in Figure 6.11. In this experiment, the target concept is a 2-term DNF with $l = 5$ where one term is of length 2 and the other item is of length 3. The number of positive samples is 5, the number of negative samples is 500, and the number of features is 100. Similarly, positive sample BDDs are processed first. Observe that the peak number of BDD nodes also occurs earlier in the process and the length l is the deciding factor for the runtime.

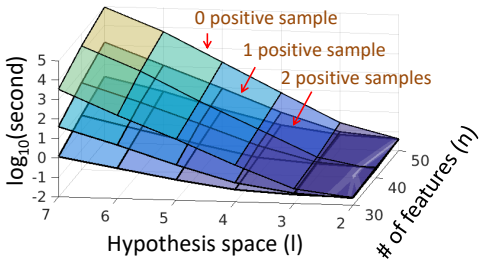

Figure 6.9: BDD runtime

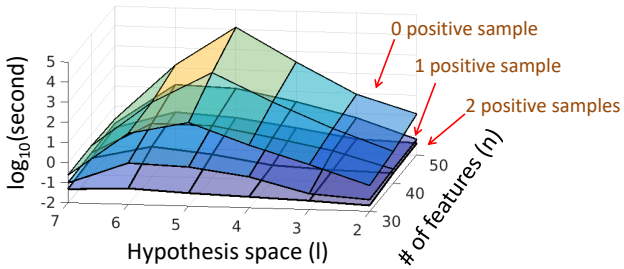

Figure 6.10: SAT runtime

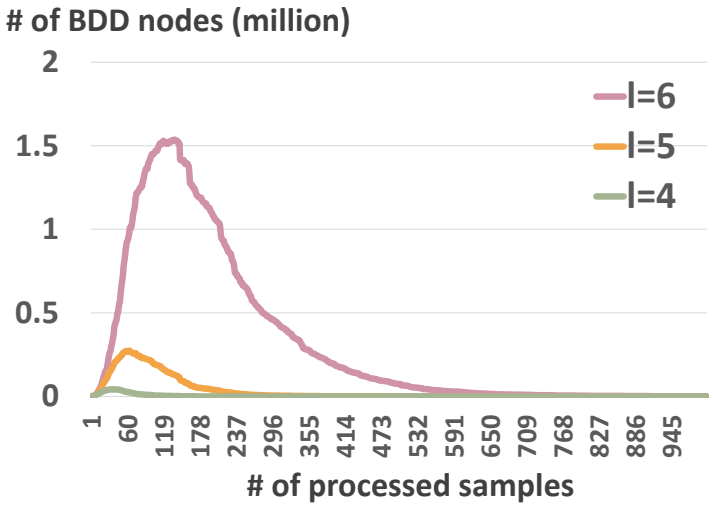

Figure 6.11: The peak number of nodes grows as l increases

6.8.3 Comparison with Other Methods

To compare VeSC-CoL with other concept learning methods such as CART and ID3 [11], we continue the experiment above where the target concept is a 5-literal monomial. There are 100 features, 2 positive samples, and 1000 negative samples. Table 6.2 shows the learning result. In each case, VeSC-CoL is able to correctly identify the target concept. On the other hand, the results from CART are less meaningful. For the first three tasks, each CART result has only 1 literal relevant to the target concept while providing 6 unrelated literals and missing 4 literals in the target. For the last task, the CART learning result is a 2-term DNF in which no feature is related to the target. The learning results from ID3 are dissimilar to the target concept as well.

Table 6.2: VeSC-CoL learns the target monomial, while CART and ID3 produce irrelevant results

VeSC-CoL	CART	ID3
$x_2x_{63}\overline{x_{75}}x_{78}\overline{x_{80}}$	$x_3x_4x_{28}x_{47}\overline{x_{53}}\overline{x_{55}}\overline{\boldsymbol{x_{80}}}$	$\boldsymbol{x_2}x_3x_4\overline{x_{30}}x_{47}\overline{x_{53}}\overline{x_{81}}$
$x_{39}\overline{x_{45}}x_{72}\overline{x_{74}}x_{95}$	$\overline{x_5}x_{16}x_{35}\overline{\boldsymbol{x_{45}}}\overline{x_{55}}\overline{x_{56}}x_{59}$	$x_8x_{40}\overline{\boldsymbol{x_{45}}}x_{64}\overline{\boldsymbol{x_{74}}}x_{87}$
$\overline{x_2}\overline{x_{14}}x_{52}\overline{x_{57}}x_{87}$	$x_{11}\overline{\boldsymbol{x_{14}}}\overline{x_{24}}x_{61}x_{64}x_{90}\overline{x_{92}}$	$\overline{x_5}\overline{x_6}x_{16}x_{35}\overline{x_{45}}\overline{x_{56}}x_{59}$
$x_{40}\overline{x_{45}}x_{64}\overline{x_{74}}x_{87}$	$\overline{x_4}x_8\overline{\boldsymbol{x_{45}}}\overline{x_{47}}\boldsymbol{x_{64}}\overline{\boldsymbol{x_{74}}}x_{89}$	$\overline{x_2}\overline{x_{14}}\overline{x_{24}}x_{61}\boldsymbol{x_{64}}x_{90}\overline{x_{92}}$
$\overline{x_{57}}x_{58}x_{77}\overline{x_{95}}x_{98}$	$\overline{x_5}x_{29}x_{38}\overline{x_{43}}\overline{x_{79}}x_{99} + \overline{x_3}\overline{x_5}\overline{x_{29}}x_{38}\overline{x_{43}}x_{49}\overline{x_{79}}x_{99}$	$\overline{x_5}x_6\overline{x_{11}}\overline{x_{14}}\overline{x_{18}}\overline{x_{34}}x_{45}$

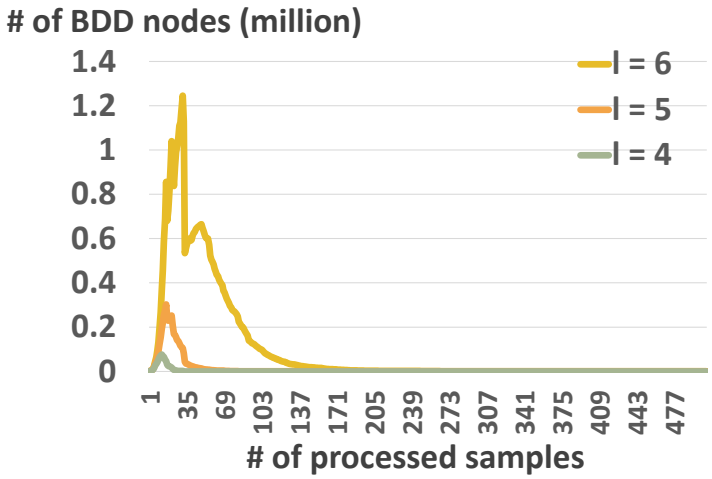

Figure 6.12: Similar result for 2-term DNF

If the target concept is a 2-term DNF, the results from CART and ID3 can be much less meaningful. We use a simple experiment to demonstrate the irrelevance of the target and the results from CART and ID3. In this experiment, the target concept is a 2-term DNF with 5 literals. There are 100 features, 6 positive samples and 100 negative samples. Table 6.3 shows the results, where VeSC-CoL successfully learn the target, while CART produced a 3-term DNF with 15 literals and ID3 produced a 3-term DNF with 19 literals.

Table 6.3: VeSC-CoL learns the target 2-term DNF, while CART and ID3 produce irrelevant results

VeSC-CoL	$x_2\overline{x_{80}} + \overline{x_{68}}\,\overline{x_{85}}\,\overline{x_{89}}$
CART	$x_2\overline{x_{51}}\,\overline{x_{80}} +$ $\overline{x_1}\,\overline{x_2}x_{30}x_{33}\overline{x_{51}}\,\overline{x_{80}} +$ $\overline{x_7}x_{28}\overline{x_{51}}x_{80}x_{81}x_{100}$
ID3	$x_2\overline{x_{51}}x_{59}x_{64}\overline{x_{80}} +$ $\overline{x_2}\,\overline{x_8}\,\overline{x_{35}}\,\overline{x_{51}}x_{59}x_{64}\overline{x_{98}} +$ $x_2x_{13}x_{37}\overline{x_{51}}x_{64}x_{59}x_{80}$

6.8.4 Complexity Ordering

As mentioned before, for two hypothesis sub-space H_i and H_j of k-term DNF, we consider the complexity of H_i is smaller than H_j if $l_i < l_j$ where l_i and l_j are the numbers of literals in the hypotheses in H_i and H_j, respectively. Recall that each hypothesis sub-space comprises hypotheses of equal length. Note that this complexity ordering is based on two main reasons: (1) As shown above, BDD-based learning is sensitive to the length l. Hence, the ordering ensures that the learning processes the computationally-easier hypothesis sub-spaces first. (2) In practice, a concept with a smaller length is easier to interpret than that with a larger length. Therefore, it is preferred to uncover a shorter concept if possible.

6.8.5 Accuracy of VeSC-CoL

In the experiments to compare VeSC-CoL with CART and ID3, we observe that VeSC-CoL can always uncover the correct answer. Note that it is possible to construct a dataset to fool the VeSC-CoL tool so that it reports an incorrect answer even with the cardinality bound $B = 1$. However, with randomly generated datasets, we observe that when the data is sufficiently large and $B = 1$, VeSC-CoL always finds the correct target concept assuming the concept is in the hypothesis space considered (e.g. 3-term DNF up to length 15). In particular, we observed in the following experiments that VeSC-CoL always find the correct answer:

- All 1-term DNF cases with up to 100 features and l up to 7. The number of positive samples can be 0 to 2 and the number of negative samples is 10000.
- All 1-term DNF cases with up to 500 features and l up to 8. The number of positive samples is larger than 5 and the number of negative samples is 10000.

- All 2-term DNF cases with up to 100 feature and l up to 8. For each term, there exists a positive sample that can be explained only by the term. The number of positive samples is larger than 5 and the number of negative samples is 10000.
- All 3-term DNF cases with up to 100 feature and l up to 9. For each term, there exists a positive sample that can be explained only by the term. The number of positive samples is larger than 10 and the number of negative samples is 10000.

6.9 Summary

We propose VeSC-CoL, a version space cardinality based concept learning tool, for learning extremely imbalanced datasets. We use experiment results to note several key properties of the tool. VeSC-CoL is applicable without cross-validation. The version space cardinality bound is used to control the quality of the learning result. In our study, we observed that VeSC-CoL can always identify the correct target concept assuming that the concept is included in one of the hypothesis sub-spaces to be analyzed. VeSC-CoL is supported by two implementations, one based on BDD and the other based on SAT. Their runtimes can be quite different. Therefore VeSC-CoL runs the two methods in parallel and stops when one of them completes.

For future work, one challenge is to generalize the encoding method. The current encoding method is closely related to the k-term DNF representation and complexity measurement. Suppose a hypothesis is represented in BDD and the complexity measure is the number of BDD nodes, the encoding will be different. Given a hypothesis representation and a complexity measure, finding an encoding method is a non-trivial task that needs further investigation.

Chapter 7

Conclusion

This work proposes a learning-based system for coverage-directed test generation. The main contribution of this work is to extend the scope of learning-based coverage-directed test generation from learning the relationship between coverage and test parameters to learning test generation and design knowledge. Three key components and challenges are identified in the proposed work, test generation, signals database for design signals and rule learning algorithms.

The proposed test generation method has the ability to learn from test examples to increase its capability. The proposed signals database can learn from the results of rule learning, from design documents and can be input directly by experienced engineers. The proposed rule learning method can output the version space explicitly to reduce the number of iterations in the rule learning and validation process. When the proposed learning system fails to handle a certain coverage point, because the proposed learning algorithm provides information about the size of version space and its hypothesis subspace, it is easy to identify that more data is required or another set of features are required. After this information to experienced engineers, they can provide examples in terms of new tests, to increase test generation capability, or a new set of features, to

increase the design knowledge.

7.1 Future Works

Ideally, the proposed verification can deal with all the coverage points as long as it has enough capability in test generation and signals database. In practice, it is still unknown that how much time and engineering efforts are required to achieve the closure of capability increasing. This problem needs to be further investigated in the future.

Another problem in functional verification in practice is that the design implementation can be updated during verification, i.e. the implementation can be time-varying. When there is an update in the design implementation, some knowledge stored in the test generation model and signals database may no longer be valid. Though dealing with time-varying design implementation is not discussed in this work, it is an important topic in practice.

Component-wisely, there are interesting topics worth investigation in each component. For constrained-process discovery, methods that can extract primitives from manually developed tests can be explored. Such methods can free human engineers from developing code in the representation of pre-defined primitives and make the process of providing test samples more natural. For signals database, more information can be stored to assist people in identifying signal importance and relevance. For example, when and where a set of features are from, the number of coverage points that can be inferred by a given set of features, etc. This information may help improve the efficiency of the system by not running rule learning algorithms for every set of signals. Also, this information can help human analyze a failure. For rule learning methods, it is unknown whether the proposed encoding and implementation methods are optimal. Beside, theoretically there can be different definitions of hypothesis complexity. Different complexity definitions

have different encoding schemes and may performs differently.

Bibliography

[1] E. M. Clarke, T. A. Henzinger, H. Veith, and R. P. Bloem, *Handbook of model checking*. Springer, 2016.

[2] R. E. Bryant, "Symbolic simulationtechniques and applications," in *Proceedings of the 27th ACM/IEEE Design Automation Conference*, pp. 517–521, ACM, 1991.

[3] H. Height, *A practical guide to adopting the universal verification methodology (UVM)*. Lulu. com, 2013.

[4] Y. Naveh, M. Rimon, I. Jaeger, Y. Katz, M. Vinov, E. s Marcu, and G. Shurek, "Constraint-based random stimuli generation for hardware verification," *AI magazine*, vol. 28, no. 3, p. 13, 2007.

[5] G. Moretti, "Accelleras support for esl verification and stimulus reuse," *IEEE Design & Test*, vol. 34, no. 4, pp. 69–75, 2017.

[6] A. Piziali, *Functional verification coverage measurement and analysis*. Springer Science & Business Media, 2007.

[7] J. Fürnkranz and T. Kliegr, "A brief overview of rule learning," in *Rule Technologies: Foundations, Tools, and Applications* (N. Bassiliades, G. Gottlob, F. Sadri, A. Paschke, and D. Roman, eds.), (Cham), pp. 54–69, Springer International Publishing, 2015.

[8] V. Svetnik, A. Liaw, C. Tong, J. C. Culberson, R. P. Sheridan, and B. P. Feuston, "Random forest: a classification and regression tool for compound classification and qsar modeling," *Journal of chemical information and computer sciences*, vol. 43, no. 6, pp. 1947–1958, 2003.

[9] N. Lavrač, B. Kavšek, P. Flach, and L. Todorovski, "Subgroup discovery with cn2-sd," *Journal of Machine Learning Research*, vol. 5, no. Feb, pp. 153–188, 2004.

[10] P. Clark and T. Niblett, "The cn2 induction algorithm," *Machine learning*, vol. 3, no. 4, pp. 261–283, 1989.

[11] J. R. Quinlan, "Induction of decision trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[12] L. Breiman, J. Friedman, R. Olshen, and C. Stone, *Classification and Regression Trees*. Monterey, CA: Wadsworth and Brooks, 1984.

[13] C. Roth Jr and L. Kinney, *Fundamentals of logic design*. Nelson Education, 2013.

[14] S. A. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*, pp. 151–158, ACM, 1971.

[15] M. Järvisalo, D. Le Berre, O. Roussel, and L. Simon, "The international sat solver competitions," *AI Magazine*, vol. 33, no. 1, pp. 89–92, 2012.

[16] M. Bjork, "Successful sat encoding techniques," *Journal on Satisfiability, Boolean Modeling and Computation*, 2009.

[17] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *Computers, IEEE Transactions on*, vol. 100, no. 8, pp. 677–691, 1986.

[18] K. S. Brace, R. L. Rudell, and R. E. Bryant, "Efficient implementation of a bdd package," in *Design Automation Conference, 1990. Proceedings., 27th ACM/IEEE*, pp. 40–45, IEEE, 1990.

[19] S. Fine and A. Ziv, "Coverage directed test generation for functional verification using bayesian networks," in *Proceedings of the 40th annual Design Automation Conference*, pp. 286–291, ACM, 2003.

[20] G. Squillero, "Microgpan evolutionary assembly program generator," *Genetic Programming and Evolvable Machines*, vol. 6, no. 3, pp. 247–263, 2005.

[21] I. Wagner, V. Bertacco, and T. Austin, "Stresstest: an automatic approach to test generation via activity monitors," in *Proceedings of the 42nd annual Design Automation Conference*, pp. 783–788, ACM, 2005.

[22] W. Chen, L.-C. Wang, J. Bhadra, and M. Abadir, "Simulation knowledge extraction and reuse in constrained random processor verification," in *Proceedings of the 50th Annual Design Automation Conference*, p. 120, ACM, 2013.

[23] C. De La Higuera, "A bibliographical study of grammatical inference," *Pattern recognition*, vol. 38, no. 9, pp. 1332–1348, 2005.

[24] W. M. Van der Aalst, V. Rubin, H. Verbeek, B. F. van Dongen, E. Kindler, and C. W. Günther, "Process mining: a two-step approach to balance between underfitting and overfitting," *Software & Systems Modeling*, vol. 9, no. 1, pp. 87–111, 2010.

[25] J. E. Hopcroft, *Introduction to Automata Theory, Languages and Computation: For VTU, 3/e*. Pearson Education India, 1979.

[26] E. M. Gold, "Language identification in the limit," *Information and control*, vol. 10, no. 5, pp. 447–474, 1967.

[27] J. Oncina and P. Garcia, "Inferring regular languages in polynomial updated time," in *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*, pp. 49–61, World Scientific, 1992.

[28] A. W. Biermann and J. A. Feldman, "On the synthesis of finite-state machines from samples of their behavior," *IEEE transactions on Computers*, vol. 100, no. 6, pp. 592–597, 1972.

[29] D. Angluin, "Learning regular sets from queries and counterexamples," *Information and Computation*, vol. 75, pp. 87–106, 1987.

[30] W. M. P. van der Aalst, *Process Mining: Discovery, Conformance and Enhancement of Business Processes*. Springer Publishing Company, 1st ed., 2011.

[31] P.-H. Chang and L.-C. Wang, "Automatic assertion extraction via sequential data mining of simulation traces," in *2010 15th Asia and South Pacific Design Automation Conference*, pp. 607–612, IEEE, 2010.

[32] W. Klieber and G. Kwon, "Efficient cnf encoding for selecting 1 from n objects," in *Proc. International Workshop on Constraints in Formal Verification*, 2007.

[33] Y. S. Mahajan, Z. Fu, and S. Malik, "Zchaff2004: An efficient sat solver," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 360–375, Springer, 2004.

[34] J. C. Reynar and A. Ratnaparkhi, "A maximum entropy approach to identifying sentence boundaries," in *Proceedings of the fifth conference on Applied natural language processing*, pp. 16–19, Association for Computational Linguistics, 1997.

[35] E. Brill, "A simple rule-based part of speech tagger," in *Proceedings of the workshop on Speech and Natural Language*, pp. 112–116, Association for Computational Linguistics, 1992.

[36] T. Brants, "Tnt: a statistical part-of-speech tagger," in *Proceedings of the sixth conference on Applied natural language processing*, pp. 224–231, Association for Computational Linguistics, 2000.

[37] Y. Shinyama, "Pdfminer: Python pdf parser and analyzer (2010)."

[38] S. Bird, "Nltk: the natural language toolkit," in *Proceedings of the COLING/ACL on Interactive presentation sessions*, pp. 69–72, Association for Computational Linguistics, 2006.

[39] "http://pandas.pydata.org/."

[40] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. Oct, pp. 2825–2830, 2011.

[41] S. Esmeir and S. Markovitch, "Occam's razor just got sharper," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, IJCAI'07, (San Francisco, CA, USA), pp. 768–773, Morgan Kaufmann Publishers Inc., 2007.

[42] V. Vapnik, "Principles of risk minimization for learning theory," in *Advances in neural information processing systems*, pp. 831–838, 1992.

[43] J. Pearl, "On the connection between the complexity and credibility of inferred models," *International Journal of General System*, vol. 4, no. 4, pp. 255–264, 1978.

[44] H. Hirsh, N. Mishra, and L. Pitt, "Version spaces and the consistency problem," *Artificial Intelligence*, vol. 156, no. 2, pp. 115–138, 2004.

[45] L. Pitt and L. G. Valiant, "Computational limitations on learning from examples," *Journal of the ACM (JACM)*, vol. 35, no. 4, pp. 965–984, 1988.

[46] H. Hirsh, "Generalizing version spaces," *Machine Learning*, vol. 17, no. 1, pp. 5–46, 1994.

[47] Y. Ben-Haim, A. Ivrii, O. Margalit, and A. Matsliah, "Perfect hashing and CNF encodings of cardinality constraints," in *International Conference on Theory and Applications of Satisfiability Testing*, pp. 397–409, Springer, 2012.

[48] C. Sinz, "Towards an optimal CNF encoding of boolean cardinality constraints," in *International Conference on Principles and Practice of Constraint Programming*, pp. 827–831, Springer, 2005.

[49] F. Somenzi, "CUDD: CU decision diagram package release 3.0.0. Public software, University of Colorado at Boulder," 2015.

[50] A. Biere, "Lingeling, plingeling and treengeling entering the sat competition 2013," *Proceedings of SAT Competition*, 2013.