Abstract

A Modular Synthesis Framework for Software Deployment, Design, and Implementation

Mark Santolucito

2020

Software development is an increasingly pervasive branch of modern industry and an increasingly dominant operating cost across industries and disciplines. There have been many attempts to automate aspects of software development, including decades of program synthesis research. Program synthesis is the process of automatically generating program code from high level specifications, such as input-output examples, logical formulas, and pseudocode. When users write specifications, we enable them to build systems by describing *what* the system does, rather than worrying about *how* it is done. Program synthesis has stood ready to revolutionize developer workflows, yet the impact on the ground has been limited to simple data cleaning and automation tasks.

This dissertation focuses on developing new program synthesis and analysis techniques to help automate the deployment, design, and implementation of software systems. By combining insights from logic and language-driven perspectives, as well as a code-as-data perspective, we introduce new techniques that enable program synthesis systems to scale to domains that have previously evaded automation. In particular, we look at domains such as configuration file management, reactive systems, and digital signal processing. From the design perspective, we introduce Temporal Stream Logic, a new temporal logic that leverages abstractions from Functional Reactive Programming to enable the synthesis of reactive systems that were previously out-of-scope for existing tools. We additionally introduce the problem of Digital Signal Processing Programming By Example, a domain where prior synthesis efforts do not support the large search space of digital signal processing programs. Complementing the design and implementation aspects of software, we adapt data-driven techniques, such as Association Rule Learning, to use synthesis in the context of software deployment challenges, such as analyzing configuration files. With these advances, we have successfully synthesized systems such as mobile apps, self-driving car controllers, and embedded systems. The dissertation concludes with reflections on how the future of program synthesis interfaces and the role synthesis plays in making access to computer science more equitable to students of diverse backgrounds.

A Modular Synthesis Framework for Software Deployment, Design, and Implementation

A Dissertation
Presented to the Faculty of the Graduate School
of
Yale University
in Candidacy for the Degree of
Doctor of Philosophy

by
Mark Santolucito

Dissertation Director: Ruzica Piskac

May, 2020

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Information technology has been praised as a labor saver and cursed as a destroyer of obsolete jobs. But the entire edifice of modern computing rests on a fundamental irony: the software that makes it all possible is, in a very real sense, handmade.

- Moshe Vardi

The software development process captures a wide range of activities necessitating a variety of expertise. In addition to the concrete implementation of programs, issues such as the high level design and architecture of a system, as well as the deployment strategies must be addressed. These are all labor intensive endeavors, requiring an increasingly large share of the workforce [1].

In order to help automate aspects of this development work, research in program synthesis has exploded in recent years [2–9]. New techniques have ranged from the use of type systems [2], SMT solvers [10], and machine learning [11]. Program synthesis has had impact on industry as well, for example with the FlashFill tool, shipped with Microsoft Excel in 2013, which allows users to automate data transformations in Excel spreadsheets. However, despite these advances, the majority of program synthesis research has focused on the "implementation" stage of the development of programs.

In this dissertation, we examine program synthesis from a "full-stack" perspective. Our goal is to create synthesis procedures that work modularly across the various stages of the software development process. We look at the problem of program synthesis beyond the scope of solving well-defined logical implementation issues, and look towards synthesis as an integral part of the

development cycle of software. We take a perspective on software development as involving three main stages.

We first examine what we call the *design stage*, where software is planned from a high level architecture perspective. This stage fixes key aspects of how the final program will be structure, but at this point the code is not complete enough to be executed. Next we examine the *implementation stage*, where the implementation details of high level design are filled in to create an executable program. Finally, we look at the *deployment stage*, where an implemented system is deployed on infrastructure such that it can be accessed by the intended users.

Similarly, reactive systems are a class of systems found across domains, from autonomous vehicles controllers to mobile applications. Reactive systems capture both complex temporal behavior and complex data transformations – by introducing a new logic that separates these two concerns [12], we have successfully applied reactive synthesis to previously intractable domains [13, 14]. Programming-by-example aims to make PBE synthesis more accessible and helpful to programmers by integrating inference-driven synthesis techniques with data-driven techniques, as well as redesigning the interface for PBE.

1.1 Synthesis for Design

In looking at the design of software, we focus on particularly on reactive systems. Reactive systems describe the class of systems that process continuous input and produce continuous output. The key contribution of this dissertation in the domain of reactive systems is to combine reactive synthesis with abstractions from programming languages. Reactive synthesis is a technique for synthesizing finite state automata generally applied to finite domains such as hardware. By incorporating abstractions from programming languages that can handle the arbitrary complexity of software domains into reactive synthesis, we are able to Specifically, we use Functional Reactive Programming (FRP), which is a powerful dataflow model of reactive computations embedded in a higher-order functional language. We use the type system of FRP to separate the programming tasks of managing control flow over time and the data manipulations that occur at each point in time. This allows us modularize synthesis and shift the focus of the classic reactive synthesis problem away from data manipulations and concentrate only on the temporal control flow. The motivation behind this approach is that

while developers are able to write small component functions to manipulate data, combining these functions to implement complex temporal behavior is a task best suited to a synthesis engine.

Our approach introduces a new logic, Temporal Stream Logic (TSL), in which formulas describe how function applications change over time, rather than how data changes over time. Whereas the largest reactive system previously synthesized was the AMBA Bus protocol, we have successfully used TSL to synthesize programs that were previously far out of scope for existing tools. Some of these applications include a fully functioning music player Android app from a temporal logic specification [**?**], and reverse engineering of production-level embedded systems [14]. Another exciting application of reactive program synthesis is in synthesis procedures for cyber physical systems, such as autonomous vehicle controllers [13]. Complex cyber-physical systems are not well described with temporal structures alone, but also require numerically sensitive constants, such as the maximal allowable steering angle - making TSL an ideal fit for this domain.

1.2 Synthesis for Implementation

After constructing software designs with TSL, the software development process must still proceed with the implementation stage. The implementation stage of software development is the area for which program synthesis has seen the most work [2–8]. In this dissertation we explore two new directions in this direction of synthesis. First, we extend programming by example to the domain of digital signal processing for computer music. In this work, we introduce the problem of digital signal processing programming-by-example (DSP-PBE), where users can provide input-output examples of audio files, and the system must synthesize the DSP program that transforms the input to the output [15]. Additionally, To this end, we introduce a methodology for using neural networks to accelerate synthesis for the syntax guided synthesis (SyGuS) style PBE [16]. Specificaly, we use neural networks as a preprocessing stage to prune the search space, and thereby help the SMT solver CVC4 [17] solve PBE problems more quickly.

1.3 Synthesis for Deployment

The development of software does not happen isolation from the systems on which it will be deployed. The deployment stage of software development is one that is ongoing throughout the lifetime of a software system. In creating opportunities for synthesis to assist with this process, we look specifically at issues of configurations. Configuration files are a ubiquitous software interface design that gives users access to critical knobs to adjust system behavior. While convenient, these files can easily be misconfigured in ways that have far-reaching, societal level impacts, for example taking down the 911 service for 12,600 callers [18].

A survey paper found that in storage systems, misconfigurations are twice as likely to cause system failures as program bugs [19]. More generally, configurations are a particularly problematic aspect of software development and deployment. Configuration files provide easy access for developers to adjust critical parameters of software behavior, but are largely unstructured (often text files of key-value pairs), making analysis and assistive technologies (such as synthesis) difficult to build [**?**]. This is problematic as traditional verification techniques rely on formal, logic specifications that define expected behavior of the language.

To address this issue we developed a tool, ConfigV, that synthesizes specifications for configuration files [20, 21]. This work required two core theoretical advances, the first was the introduction of probabilistic types, and the second was an extension to the data mining technique, Association Rule Learning. Since configuration files lack helpful semantic information to infer types, we use a probabilistic inference method to learn likely types for keywords based on their values from the training set. We combined this new type information with a generalization of Association Rule Learning that handles not just association rules, but arbitrary, typed relations.

We also look at configurations of continuous integration systems, such as TravisCI. In this setting, we have a history changes made to the code base (i.e. through version control), and each state of the code base is labeled as a passing or failing build. We combine formal reasoning techniques with decision tree learning to build a tool that allows us to learn rules that describe continuous integration configurations that are more or less likely to lead to build failures.

1.4 Synthesis for Impact

Finally, we look at the ways that synthesis has an impact on real-world. We ask what users need and expect when using program synthesis tools - a critical part of ensuring that program synthesis has the intended impact on the software development process. In particular, we look at the application of synthesis in the domain of scripting tasks. We find that, contrary to intuition, a synthesis tool decreasing the time to complete a task does not always positively correlate to users' perception of the helpfulness of that tool [22].

We additionally look at the role synthesis research plays in education. One of the most pressing issues facing the field of computer science today is the lack of diversity. We propose that synthesis, as a research area, is particularly well suited to involving and meaningfully engaging a diverse student body in computer science research. We reflect on a number of case studies of our own work with students from diverse backgrounds, and provide a mentoring framework to help future researchers in their efforts of involved students in program synthesis research.

Part I ❖ Synthesis for Design

The first part of this dissertation explores the role synthesis can play in designing software. In particular, we focus the synthesis of reactive programs. Reactive systems are a broad class of systems that are characterized by their continuous processing of input, and continuous production of input. Reactive systems are considered among the most difficult types of systems to design correctly [23]. Furthermore, even once a design has been fixed, writing the reactive program is still a highly error-prone process. As such, synthesizing a program, or parts of a program, could significantly improve the development process. Not only would synthesis streamline development, it would help to minimize errors, as one of the benefits of synthesized code is that it is correct-by-construction.

The synthesis of reactive systems has seen much work []. The standard approach of existing work uses specifications written in linear temporal logic (LTL), or variations thereof, a logic in which formulas describe how Boolean values change over time. However, one of the key limitations to reactive synthesis in a Boolean logic, like LTL, is that complex datatypes such as integers must be encoded with their Boolean expansion. Since the complexity of the LTL synthesis problem (finding a model/program that satisfies a given LTL formula) is doubly exponential [], Boolean expansion of any complex data is very expensive. This limits the ability of reactive synthesis to scale to the synthesis of programs and software systems. In our investigations, we noticed that prior approaches to reactive synthesis were all, as a result of the Boolean representations, implicitly trying to synthesize both control and data transformations at the same time. As a result, synthesis was not tractable for complex, real-world problems where the specification encodes the problem of temporal control, as well as data transformations.

To remedy this situation, we looked to recent work in the programming languages domain. Frameworks for building reactive programs provide the abstractions of data that we require for handling complex data in a reactive system. In our work, we focused on Functional Reactive Programming, or FRP, which is a declarative programming paradigm based on two fundamental abstractions: a continuous (functional) modeling of time-varying behaviors, and a discrete (reactive) calculus of user and process interaction. As a paradigm in a functional setting, FRP shared many similarities to logic and specification driven approaches to reactive synthesis. To make this connection

clear, and leverage the benefits of the reactive synthesis community as well as advances in FRP, we introduced a new logic called Temporal Stream Logic. In the subsequent section, we introduce Temporal Stream Logic and demonstrate its connection to FRP. We used Temporal Stream Logic to synthesize software for complex systems that were previously out-of-scope for existing tools, such as an autonomous vehicle controller that is formally guaranteed to correctly implement the specified driving behavior.

Chapter 2

Temporal Stream Logic

Work presented in this chapter was completed in collaboration with Felix Klein, Bernd Finkbeiner, and Ruzica Piskac. Sections of this work have been previously published [12–14], and are reproduced here, at times in their original form.

Reactive systems that operate in environments with complex data, such as mobile apps or embedded controllers with many sensors, are difficult to synthesize. Synthesis tools usually fail for such systems because the state space resulting from the discretization of the data is too large. We introduce TSL, a new temporal logic that separates control and data. We provide a CEGAR-based synthesis approach for the construction of implementations that are guaranteed to satisfy a TSL specification for all possible instantiations of the data processing functions. TSL provides an attractive trade-off for synthesis. On the one hand, synthesis from TSL, unlike synthesis from standard temporal logics, is undecidable in general. On the other hand, however, synthesis from TSL is scalable, because it is independent of the complexity of the handled data. Among other benchmarks, we have successfully synthesized a music player Android app and a controller for an autonomous vehicle in the Open Race Car Simulator (TORCS).

2.1 Motivation

Reactive programs implement a broad class of computer systems whose defining element is the continued interaction between the system and its environment. Their importance can be seen through

the wide range of applications, such as embedded devices [24], games [25], robotics [26], hardware circuits [27], GUIs [28], and interactive multimedia [29].

In reactive synthesis, we automatically translate a formal specification, typically given in a temporal logic, into a controller that is guaranteed to satisfy the specification. Over the past two decades there has been much progress on reactive synthesis, both in terms of algorithms, notably with techniques like GR(1)-synthesis [30] and bounded synthesis [31], and in terms of tools, as showcased, for example, in the annual SYNTCOMP competition [32].

In practice however, reactive synthesis has seen limited success. One of the largest published success stories [27] is the synthesis of the AMBA bus protocol. To push synthesis even further, automatically synthesizing a controller for an autonomous system has been recognized to be of critical importance [33]. Despite many years of experience with synthesis tools, our own attempts to synthesize such controllers with existing tools have been unsuccessful. The reason is that the tools are unable to handle the data complexity of the controllers. The controller only needs to switch between a small number of behaviors, like steering during a bend, or shifting gears on high rpm. The number of control states in a typical controller (cf. [34]) is thus not much different from the arbiter in the AMBA case study. However, in order to correctly initiate transitions between control states, the driving controller must continuously process data from more than 20 sensors.

If this data is included (even as a rough discretization) in the state space of the controller, then the synthesis problem is much too large to be handled by any available tools. It seems clear then, that a scalable synthesis approach must separate control and data. If we assume that the data processing is handled by some other approach (such as deductive synthesis [35] or manual programming), is it then possible to solve the remaining reactive synthesis problem?

In this paper, we show scalable reactive synthesis is indeed possible. Separating data and control has allowed us to synthesize reactive systems, including an autonomous driving controller and a music player app, that had been impossible to synthesize with previously available tools. However, the separation of data and control implies some fundamental changes to reactive synthesis, which we describe in the rest of the paper. The changes also imply that the reactive synthesis problem is no longer, in general, decidable. We thus trade theoretical decidability for practical scalability, which is, at least with regard to the goal of synthesizing realistic systems, an attractive trade-off.

We introduce Temporal Stream Logic (TSL), a new temporal logic that includes *updates*, such as $[\![\mathtt{y} \leftarrowtail \mathtt{f}\ \mathtt{x}]\!]$, and predicates over arbitrary function terms. The update $[\![\mathtt{y} \leftarrowtail \mathtt{f}\ \mathtt{x}]\!]$ indicates that the result of applying function `f` to variable `x` is assigned to `y`. The implementation of predicates and functions is not part of the synthesis problem. Instead, we look for a system that satisfies the TSL specification *for all possible interpretations of the functions and predicates*.

This implicit quantification over all possible interpretations provides a useful abstraction: it allows us to *independently* implement the data processing part. On the other hand, this quantification is also the reason for the undecidability of the synthesis problem. If a predicate is applied to the same term *twice*, it must (independently of the interpretation) return the *same* truth value. The synthesis must then implicitly maintain a (potentially infinite) set of terms to which the predicate has previously been applied. As we show later, this set of terms can be used to encode PCP [36] for a proof of undecidability.

We present a practical synthesis approach for TSL specifications, which is based on bounded synthesis [31] and counterexample-guided abstraction refinement (CEGAR) [37]. We use bounded synthesis to search for an implementation up to a (iteratively growing) bound on the number of states. This approach underapproximates the actual TSL synthesis problem by leaving the interpretation of the predicates to the environment. The underapproximation allows for inconsistent behaviors: the environment might assign different truth values to the same predicate when evaluated at different points in time, even if the predicate is applied to the same term. However, if we find an implementation in this underapproximation, then the CEGAR loop terminates and we have a correct implementation for the original TSL specification. If we do not find an implementation in the underapproximation, we compute a counter strategy for the environment. Because bounded synthesis reduces the synthesis problem to a safety game, the counter strategy is a reachability strategy that can be represented as a finite tree. We check whether the counter strategy is spurious by searching for a pair of positions in the strategy where some predicate results in different truth values when applied to the same term. If the counter strategy is not spurious, then no implementation exists for the considered bound, and we increase the bound. If the counter strategy is spurious, then we introduce a constraint into the specification that eliminates the incorrect interpretation of the predicate, and continue with the refined specification.

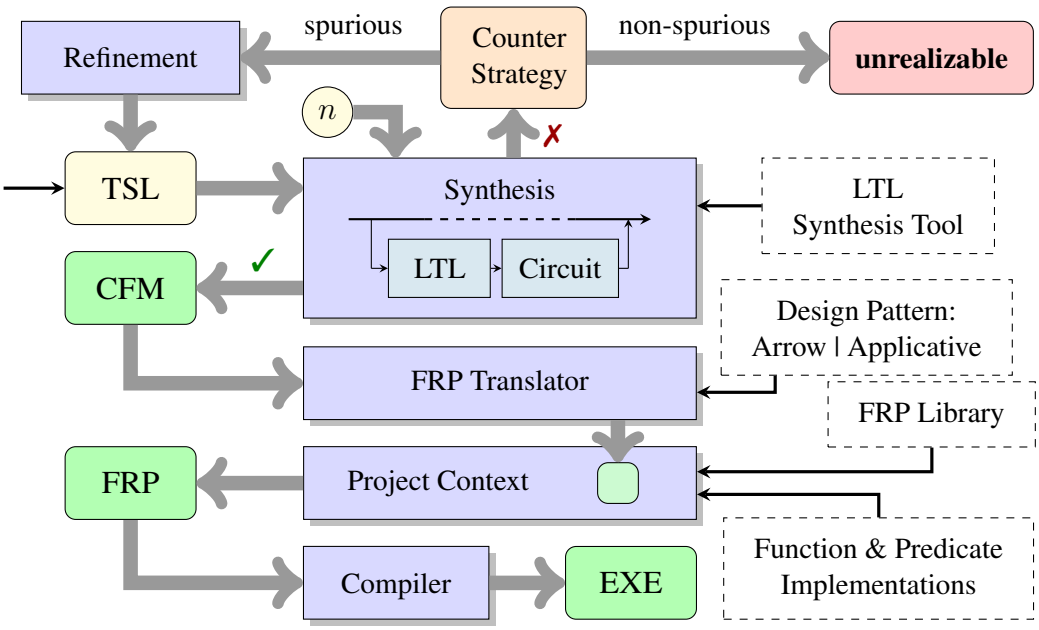


Figure 2.1: The TSL synthesis procedure uses a modular design. Each step takes input from the previous step as well as interchangeable modules (dashed boxes).

A general overview of this procedure is shown in Fig. 2.1. The top half of the figure depicts the bounded search for an implementation that realizes a TSL specification using the CEGAR loop to refine the specification. If the specification is realizable, we proceed in the bottom half of the process, where a synthesized implementation is converted to a control flow model (CFM) determining the control of the system. We then specialize the CFM to Functional Reactive Programming (FRP), which is a popular and expressive programming paradigm for building reactive programs using functional programming languages [38]. Our framework supports any FRP library using the *Arrow* or *Applicative* design patterns, which covers most of the existing FRP libraries (e.g. [39–42]). Finally, the synthesized control flow is embedded into a project context, where it is equipped with function and predicate implementations and then compiled to an executable program.

Our experience with synthesizing systems based on TSL specifications has been extremely positive. The synthesis works for a broad range of benchmarks, ranging from classic reactive synthesis problems (like escalator control), through programming exercises from functional reactive programming, to novel case studies like our music player app and the autonomous driving controller for a vehicle in the Open Race Car Simulator (TORCS).

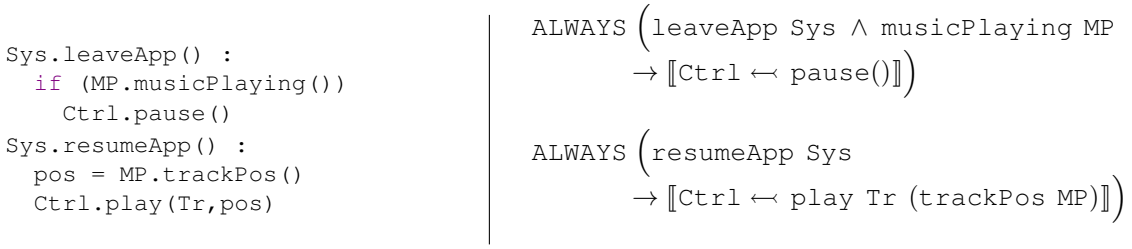


Figure 2.2: Sample code and specification for the music player app.

2.2 Motivating Example

To demonstrate the utility of our method, we synthesized a music player Android app[1] from a TSL specification. A major challenge in developing Android apps is the temporal behavior of an app through the *Android lifecycle* [43]. The Android lifecycle describes how an app should handle being paused, when moved to the background, coming back into focus, or being terminated. In particular, *resume and restart errors* are commonplace and difficult to detect and correct [43]. Our music player app demonstrates a situation in which a resume and restart error could be unwittingly introduced when programming by hand, but is avoided by providing a specification. We only highlight the key parts of the example here to give an intuition of TSL. The complete specification is presented in [44].

Our music player app utilizes the Android music player library (`MP`), as well as its control interface (`Ctrl`). It pauses any playing music when moved to the background (for instance if a call is received), and continues playing the currently selected track (`Tr`) at the last track position when the app is resumed. In the Android system (`Sys`), the `leaveApp` method is called whenever the app moves to the background, while the `resumeApp` method is called when the app is brought back to the foreground. To avoid confusion between pausing music and pausing the app, we use `leaveApp` and `resumeApp` in place of the Android methods onPause and onResume. A programmer might manually write code for this as shown on the left in Fig. 2.2.

The behavior of this can be directly described in TSL as shown on the right in Fig. 2.2. Even eliding a formal introduction of the notation for now, the specification closely matches the textual specification. First, when the user leaves the app and the music is playing, the music pauses. Likewise for the second part, when the user resumes the app, the music starts playing again.

However, assume we want to change the behavior so that the music only plays on resume when the music had been playing before leaving the app in the first place. In the manually written program,

1. `https://play.google.com/store/apps/details?id=com.mark.myapplication`.

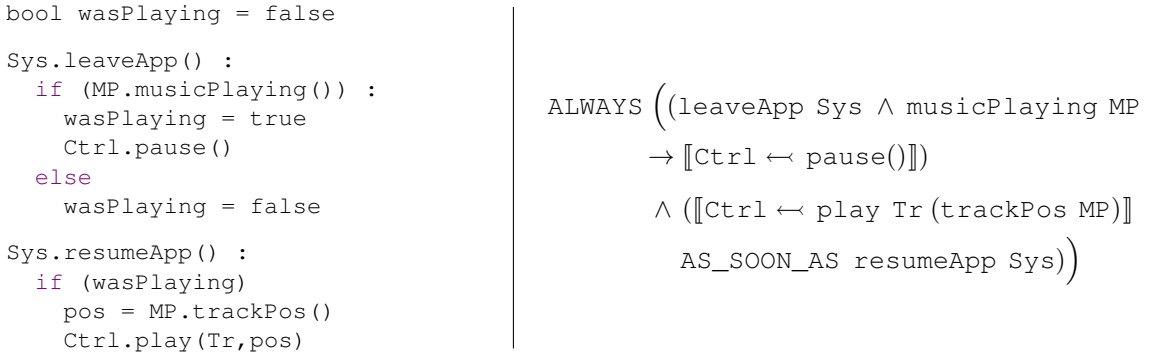


Figure 2.3: The effect of a minor change in functionality on code versus a specification.

this new functionality requires an additional variable `wasPlaying` to keep track of the music state. Managing the state requires multiple changes in the code as shown on the left in Fig. 2.3. The required code changes include: a conditional in the `resumeApp` method, setting `wasPlaying` appropriately in two places in `leaveApp`, and providing an initial value. Although a small example, it demonstrates how a minor change in functionality may require wide-reaching code changes. In addition, this change introduces a globally scoped variable, which then might accidentally be set or read elsewhere. In contrast, it is a simple matter to change the TSL specification to reflect this new functionality. Here, we only update one part of the specification to say that if the user leaves the app and the music is playing, the music has to play again as soon as the app resumes.

Synthesis allows us to specify a temporal behavior without worrying about the implementation details. In this example, writing the specification in TSL has eliminated the need of an additional state variable, similarly to a higher order `map` eliminating the need for an iteration variable. However, in more complex examples the benefits compound, as TSL provides a modular interface to specify behaviors, offloading the management of multiple interconnected temporal behaviors from the user to the synthesis engine.

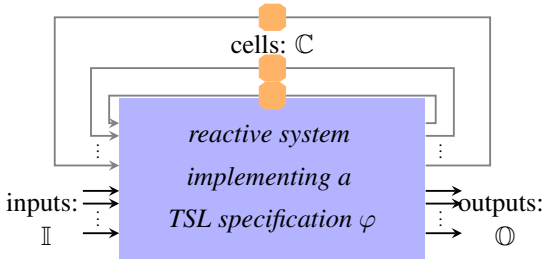
2.3 Reactive Synthesis

The synthesis of a reactive system concerns the process of automatically generating an implementation from a high-level specification. The reactive system acts as a deterministic controller, which reads inputs and produces outputs over an infinite amount of time. In contrast, a specification defines all input/output behavior pairs that are valid, i.e., allowed to be produced by the controller.

We assume time to be discrete and denote it by the set $\mathbb{N}$ of positive integers. A value is an arbitrary object of arbitrary type. $\mathcal{V}$ denotes the set of all values. The Boolean values are denoted by $\mathcal{B} \subseteq \mathcal{V}$. A stream $s\colon \mathbb{N} \to \mathcal{V}$ is a function fixing values at each point in time. An n-ary function $f\colon \mathcal{V}^n \to \mathcal{V}$ determines new values from n given values, where the set of all functions (of arbitrary arity) is given by $\mathcal{F}$. Constants are functions of arity 0. Every constant is a value, i.e., is an element of $\mathcal{F} \cap \mathcal{V}$. An n-ary predicate $p\colon \mathcal{V}^n \to \mathcal{B}$ checks a property over n values. The set of all predicates (of arbitrary arity) is given by $\mathcal{P}$, where $\mathcal{P} \subseteq \mathcal{F}$. We use $B^{[A]}$ to denote the set of all total functions with domain A and image B.

In the classical reactive synthesis setting, time is discrete and inputs and outputs are given as vectors of Boolean signals. The standard abstraction treats inputs and outputs as atomic propositions $\mathcal{I} \cup \mathcal{O}$, while their Boolean combinations form an alphabet $\Sigma = 2^{\mathcal{I} \cup \mathcal{O}}$ of alphabet symbols. At every time t signals appearing in the set $\sigma_t \in \Sigma$ are enabled (`true`), while signals not in σ_t are disabled (`false`). This fixes the behavior of the system to infinite sequences $\sigma = \sigma_0\sigma_1\sigma_2\ldots$ of alphabet symbols σ_t. The set of all such sequences is denoted by Σ^ω, where the ω-operator induces the infinite concatenation of alphabet symbols of Σ. A concrete reactive system is given as a total function mapping a stream of inputs to a stream of outputs $f\colon (2^{\mathcal{I}})^\omega \to (2^{\mathcal{O}})^\omega$. These functions often take the form of Mealy machines [45], Moore machines [46], or other transducer-like models. In the reactive synthesis setting, systems are specified directly, but rather by a formula in a temporal logic. A specification describes an arbitrary relation between input sequences $\alpha \in (2^{\mathcal{I}})^\omega$ and output sequences $\beta \in (2^{\mathcal{O}})^\omega$. The most popular such logic is Linear Temporal Logic(LTL) [47], which uses Boolean connectives to specify behavior at specific points in time, and temporal connectives, to relate sub-specifications over time. The realizability and synthesis problems for LTL are 2ExpTime-complete [48].

An implementation describes a realizing strategy, formalized via infinite trees. A Φ-labeled and Υ-branching tree is a function $\sigma\colon \Upsilon^* \to \Phi$, where Υ denotes the set of branching directions along a tree. Every node of the tree is given by a finite prefix $v \in \Upsilon^*$, which fixes the path to reach a node from the root. Every node is labeled by an element of Φ. For infinite paths $\nu \in \Upsilon^\omega$, the branch $\sigma\wr\nu$ denotes the sequence of labels that appear on ν, i.e., $\forall t \in \mathbb{N}.\ (\sigma\wr\nu)(t) = \sigma(\nu(0)\ldots\nu(t-1))$.



(a) Architecture

Function Term:

$$\tau_F := \mathtt{s_i} \mid \mathtt{f}\ \tau_F^0\ \tau_F^1\ \cdots\ \tau_F^{n-1}$$

Predicate Term:

$$\tau_P := \mathtt{p}\ \tau_F^0\ \tau_F^1\ \cdots\ \tau_F^{n-1}$$

Update:

$$[\![\mathtt{s_o} \leftarrowtail \tau_F]\!]$$

(b) Term Definitions

Figure 2.4: General architecture of reactive systems that are specified in TSL on the left, and the structure of function, predicate and updates on the right.

2.4 Temporal Stream Logic

We present a new logic: Temporal Stream Logic (TSL), which is especially designed for synthesis and allows for the manipulation of infinite streams of arbitrary (even non-enumerative, or higher order) type. It provides a straightforward notation to specify how outputs are computed from inputs, while using an intuitive interface to access time. The main focus of TSL is to describe temporal control flow, while abstracting away concrete implementation details. This not only keeps the logic intuitive and simple, but also allows a user to identify problems in the control flow even without a concrete implementation at hand. In this way, the use of TSL scales up to any required abstraction, such as API calls or complex algorithmic transformations.

2.4.1 Architecture

A TSL formula φ specifies a reactive system that in every time step processes a finite number of inputs $\mathbb{I}$ and produces a finite number of outputs $\mathbb{O}$. Furthermore, it uses cells $\mathbb{C}$ to store a value computed at time t, which can then be reused in the next time step $t + 1$. An overview of the architecture of such a system is given in Fig. 2.4a. In terms of behavior, the environment produces infinite streams of input data, while the system uses pure (side-effect free) functions to transform the values of these input streams in every time step. After their transformation, the data values are either passed to an output stream or are passed to a cell, which pipes the output value from one time step back to the corresponding input value of the next. The behaviour of the system is captured by its infinite execution over time.

2.4.2 Function Terms, Predicate Terms, and Updates

In TSL we differentiate between two elements: we use purely functional transformations, reflected by functions $f \in \mathcal{F}$ and their compositions, and predicates $p \in \mathcal{P}$, used to control how data flows inside the system. To argue about both elements we use a term based notation, where we distinguish between function terms τ_F and predicate terms τ_P, respectively. Function terms are either constructed from inputs or cells ($\mathtt{s_i} \in \mathbb{I} \cup \mathbb{C}$), or from functions, recursively applied to a set of function terms. Predicate terms are constructed similarly, by applying a predicate to a set of function terms.

Finally, an update takes the result of a function computation and passes it either to an output or a cell ($\mathtt{s_o} \in \mathbb{O} \cup \mathbb{C}$). An overview of the syntax of the different term notations is given in Fig. 2.4b. Note that we use curried argument notation similar to functional programming languages.

We denote sets of function and predicate terms, and updates by $\mathcal{T}_F$, $\mathcal{T}_P$ and $\mathcal{T}_U$, respectively, where $\mathcal{T}_P \subseteq \mathcal{T}_F$. We use $\mathbb{F}$ to denote the set of function literals and $\mathbb{P} \subseteq \mathbb{F}$ to denote the set of predicate literals, where the literals $\mathtt{s_i}$, $\mathtt{s_o}$, $\mathtt{f}$ and $\mathtt{p}$ are symbolic representations of inputs and cells, outputs and cells, functions, and predicates, respectively. Literals are used to construct terms as shown in Fig. 2.4b. Since we use a symbolic representation, functions and predicates are not tied to a specific implementation. However, we still classify them according to their arity, i.e., the number of function terms they are applied to, as well as by their type: input, output, cell, function or predicate. Furthermore, terms can be compared syntactically using the equivalence relation $\equiv$. To assign a semantic interpretation to functions, we use an assignment function $\langle\cdot\rangle : \mathbb{F} \to \mathcal{F}$.

2.4.3 Inputs, Outputs, and Computations

We consider momentary inputs $i \in \mathcal{V}^{[\mathbb{I}]}$, which are assignments of inputs $\mathtt{i} \in \mathbb{I}$ to values $v \in \mathcal{V}$. For the sake of readability let $\mathcal{I} = \mathcal{V}^{[\mathbb{I}]}$. Input streams are infinite sequences $\iota \in \mathcal{I}^\omega$ consisting of infinitely many momentary inputs.

Similarly, a momentary output $o \in \mathcal{V}^{[\mathbb{O}]}$ is an assignment of outputs $\mathtt{o} \in \mathbb{O}$ to values $v \in \mathcal{V}$, where we also use $\mathcal{O} = \mathcal{V}^{[\mathbb{O}]}$. Output streams are infinite sequences $\varrho \in \mathcal{O}^\omega$. To capture the behavior of a cell, we introduce the notion of a computation ς. A computation fixes the function terms that are used to compute outputs and cell updates, without fixing semantics of function literals. Intuitively, a

computation only determines which function terms are used to compute an output, but abstracts from actually computing it.

The basic element of a computation is a computation step $c \in \mathcal{T}_F^{[\mathbb{O} \cup \mathbb{C}]}$, which is an assignment of outputs and cells $\mathtt{s}_\circ \in \mathbb{O} \cup \mathbb{C}$ to function terms $\tau_F \in \mathcal{T}_F$. For the sake of readability let $\mathcal{C} = \mathcal{T}_F^{[\mathbb{O} \cup \mathbb{C}]}$. A computation step fixes the control flow behaviour at a single point in time. A computation $\varsigma \in \mathcal{C}^\omega$ is an infinite sequence of computation steps.

As soon as input streams, and function and predicate implementations are known, computations can be turned into output streams. To this end, let $\langle\cdot\rangle \colon \mathbb{F} \to \mathcal{F}$ be some function assignment. Furthermore, assume that there are predefined constants $\mathit{init}_\mathtt{c} \in \mathcal{F} \cap \mathcal{V}$ for every cell $\mathtt{c} \in \mathbb{C}$, which provide an initial value for each stream at the initial point in time. To receive an output stream from a computation $\varsigma \in \mathcal{C}^\omega$ under the input stream ι, we use an evaluation function $\eta_{\langle\cdot\rangle} \colon \mathcal{C}^\omega \times \mathcal{I}^\omega \times \mathbb{N} \times \mathcal{T}_F \to \mathcal{V}$:

$$\begin{aligned}
\eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{s}_\mathtt{i}) &= \begin{cases} \iota(t)(\mathtt{s}_\mathtt{i}) & \text{if } \mathtt{s}_\mathtt{i} \in \mathbb{I} \\ \mathit{init}_{\mathtt{s}_\mathtt{i}} & \text{if } \mathtt{s}_\mathtt{i} \in \mathbb{C} \ \wedge \ t = 0 \\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t-1, \varsigma(t-1)(\mathtt{s}_\mathtt{i})) & \text{if } \mathtt{s}_\mathtt{i} \in \mathbb{C} \ \wedge \ t > 0 \end{cases} \\
\eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{f}\ \tau_0\ \cdots\ \tau_{m-1}) &= \langle \mathtt{f} \rangle\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \tau_0)\ \cdots\ \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \tau_{m-1})
\end{aligned}$$

Then $\varrho_{\langle\cdot\rangle,\varsigma,\iota} \in \mathcal{O}^\omega$ is defined via $\varrho_{\langle\cdot\rangle,\varsigma,\iota}(t)(\circ) = \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \circ)$ for all $t \in \mathbb{N}$, $\circ \in \mathbb{O}$.

Syntax Every TSL formula φ is built according to the following grammar:

$$\varphi \ := \ \tau \in \mathcal{T}_P \cup \mathcal{T}_U \ \mid \ \neg\varphi \ \mid \ \varphi \wedge \varphi \ \mid \ \bigcirc\varphi \ \mid \ \varphi\,\mathcal{U}\,\varphi$$

An atomic proposition τ consists either of a predicate term, serving as a Boolean interface to the inputs, or of an update, enforcing a respective flow at the current point in time. Next, we have the Boolean operations via negation and conjunction, that allow us to express arbitrary Boolean combinations of predicate evaluations and updates. Finally, we have the temporal operator next: $\bigcirc\psi$, to specify the behavior at the next point in time and the temporal operator until: $\vartheta\,\mathcal{U}\,\psi$, which enforces a property ϑ to hold until the property ψ holds, where ψ must hold at some point in the future eventually.

2.4.4 Semantics

Formally, this leads to the following semantics. Let $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$, $\iota \in \mathcal{I}^\omega$, and $\varsigma \in \mathcal{C}^\omega$ be given, then the validity of a TSL formula φ with respect to ς and ι is defined inductively over $t \in \mathbb{N}$ via:

$$
\begin{aligned}
\varsigma, \iota, t \;&\vDash_{\langle\cdot\rangle}\; \mathtt{p}\; \tau_0\; \cdots\; \tau_{m-1} && :\Leftrightarrow && \eta_{\langle\cdot\rangle}(\varsigma, \iota, t, \mathtt{p}\; \tau_0\; \cdots\; \tau_{m-1}) \\
\varsigma, \iota, t \;&\vDash_{\langle\cdot\rangle}\; [\![\mathtt{s} \leftarrowtail \tau]\!] && :\Leftrightarrow && \varsigma(t)(\mathtt{s}) \equiv \tau \\
\varsigma, \iota, t \;&\vDash_{\langle\cdot\rangle}\; \neg\psi && :\Leftrightarrow && \varsigma, \iota, t \;\nvDash_{\langle\cdot\rangle}\; \psi \\
\varsigma, \iota, t \;&\vDash_{\langle\cdot\rangle}\; \vartheta \wedge \psi && :\Leftrightarrow && \varsigma, \iota, t \;\vDash_{\langle\cdot\rangle}\; \vartheta \;\wedge\; \varsigma, \iota, t \;\vDash_{\langle\cdot\rangle}\; \psi \\
\varsigma, \iota, t \;&\vDash_{\langle\cdot\rangle}\; \bigcirc\psi && :\Leftrightarrow && \varsigma, \iota, t+1 \;\vDash_{\langle\cdot\rangle}\; \psi \\
\varsigma, \iota, t \;&\vDash_{\langle\cdot\rangle}\; \vartheta\,\mathcal{U}\,\psi && :\Leftrightarrow && \exists t'' \geq t.\; \forall t \leq t' < t''.\; \varsigma, \iota, t' \;\vDash_{\langle\cdot\rangle}\; \vartheta \;\wedge\; \varsigma, \iota, t'' \;\vDash_{\langle\cdot\rangle}\; \psi
\end{aligned}
$$

Consider that the satisfaction of a predicate depends on the current computation step and the steps of the past, while for updates it only depends on the current computation step. Furthermore, updates are only checked syntactically, while the satisfaction of predicates depends on the given assignment $\langle\cdot\rangle$ and the input stream ι. We say that ς and ι satisfy φ, denoted by $\varsigma, \iota \;\vDash_{\langle\cdot\rangle}\; \varphi$, if $\varsigma, \iota, 0 \;\vDash_{\langle\cdot\rangle}\; \varphi$.

Beside the basic operators, we have the standard derived Boolean operators, as well as the derived temporal operators: *release* $\varphi\,\mathcal{R}\,\psi \equiv \neg((\neg\psi)\,\mathcal{U}(\neg\varphi))$, *finally* $\diamondsuit\varphi \equiv \mathit{true}\,\mathcal{U}\,\varphi$, *always* $\square\varphi \equiv \mathit{false}\,\mathcal{R}\,\varphi$, the *weak* version of *until* $\varphi\,\mathcal{W}\,\psi \equiv (\varphi\,\mathcal{U}\,\psi) \vee (\square\varphi)$, and *as soon as* $\varphi\,\mathcal{A}\,\psi \equiv \neg\psi\,\mathcal{W}(\psi \wedge \varphi)$.

2.4.5 Realizability

We are interested in the following realizability problem: given a TSL formula φ, is there a strategy $\sigma \in \mathcal{C}^{[\mathcal{I}^+]}$ such that for every input $\iota \in \mathcal{I}^\omega$ and function implementation $\langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}$, the branch $\sigma \wr \iota$ satisfies φ, i.e.,

$$\exists \sigma \in \mathcal{C}^{[\mathcal{I}^+]}.\; \forall \iota \in \mathcal{I}^\omega.\; \forall \langle\cdot\rangle\colon \mathbb{F} \to \mathcal{F}.\; \sigma \wr \iota, \iota \;\vDash_{\langle\cdot\rangle}\; \varphi$$

If such a strategy σ exists, we say σ realizes φ. If we additionally ask for a concrete instantiation of σ, we consider the synthesis problem of TSL.

2.5 TSL Properties

In order to synthesize programs from TSL specifications, we give an overview of the first part of our synthesis process, as shown in Fig. 2.1. First we show how to approximate the semantics of TSL through a reduction to LTL. However, due to the approximation, finding a realizable strategy immediately may fail. Our solution is a CEGAR loop that improves the approximation. This CEGAR loop is necessary, because the realizability problem of TSL is undecidable in general.

2.5.1 Approximating TSL with LTL

We approximate TSL formulas with weaker LTL formulas. The approximation reinterprets the syntactic elements, $\mathcal{T}_P$ and $\mathcal{T}_U$, as atomic propositions for LTL. This strips away the semantic meaning of the function application and assignment in TSL, which we reconstruct by later adding assumptions lazily to the LTL formula.

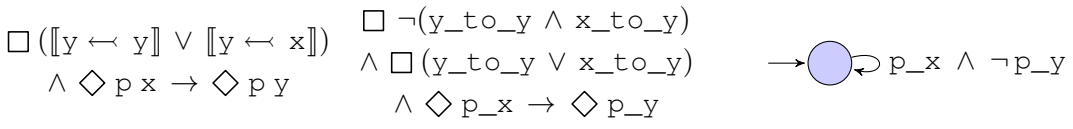
Formally, let $\mathcal{T}_P$ and $\mathcal{T}_U$ be the finite sets of predicate terms and updates, which appear in φ_{TSL}, respectively. For every assigned signal, we partition $\mathcal{T}_U$ into $\biguplus_{\mathtt{s}_\circ \in \mathbb{O} \cup \mathbb{C}} \mathcal{T}_U^{\mathtt{s}_\circ}$. For every $\mathtt{c} \in \mathbb{C}$ let $\mathcal{T}_{U/\mathtt{id}}^{\mathtt{c}} = \mathcal{T}_U^{\mathtt{c}} \cup \{[\![\mathtt{c} \leftarrowtail \mathtt{c}]\!]\}$, for $\mathtt{o} \in \mathbb{O}$ let $\mathcal{T}_{U/\mathtt{id}}^{\mathtt{o}} = \mathcal{T}_U^{\mathtt{o}}$, and let $\mathcal{T}_{U/\mathtt{id}} = \bigcup_{\mathtt{s}_\circ \in \mathbb{O} \cup \mathbb{C}} \mathcal{T}_{U/\mathtt{id}}^{\mathtt{s}_\circ}$. We construct the LTL formula φ_{LTL} over the input propositions $\mathcal{T}_P$ and output propositions $\mathcal{T}_{U/\mathtt{id}}$ as follows:

$$\varphi_{LTL} = \square\Big(\bigwedge_{\mathtt{s}_\circ \in \mathbb{O} \cup \mathbb{C}} \bigvee_{\tau \in \mathcal{T}_{U/\mathtt{id}}^{\mathtt{s}_\circ}} \big(\tau \wedge \bigwedge_{\tau' \in \mathcal{T}_{U/\mathtt{id}}^{\mathtt{s}_\circ} \setminus \{\tau\}} \neg \tau'\big)\Big) \wedge \textsc{SyntacticConversion}(\varphi_{TSL})$$

Intuitively, the first part of the equation partially reconstructs the semantic meaning of updates by ensuring that a signal is not updated with multiple values at a time. The second part extracts the reactive constraints of the TSL formula without the semantic meaning of functions and updates.

Theorem 1 ([44])**.** If φ_{LTL} is realizable, then φ_{TSL} is realizable.

Note that unrealizability of φ_{LTL} does not imply that φ_{TSL} is unrealizable. It may be that we have not added sufficiently many environment assumptions to the approximation in order for the system to produce a realizing strategy.



(a) TSL specification (b) initial approximation (c) spurious counter-strategy

Figure 2.5: A TSL specification (a) with input x and cell y that is realizable. A winning strategy is to save x to y as soon as p(x) is satisfied. However, the initial approximation (b), that is passed to an LTL synthesis solver, is unrealizable, as proven through the counter-strategy (c) returned by the LTL solver.

2.5.2 Example

As an example, we present a simple TSL specification in Fig. 2.5a. The specification asserts that the environment provides an input x for which the predicate p x will be satisfied eventually. The system must guarantee that eventually p y holds. According to the semantics of TSL the formula is realizable. The system can take the value of x when p x is true and save it to y, thus guaranteeing that p y is satisfied eventually. This is in contrast to LTL, which has no semantics for pure functions - taking the evaluation of p y as an environmentally controlled value that does not need to obey the consistency of a pure function.

2.5.3 Refining the LTL Approximation

It is possible that the LTL solver returns a counter-strategy for the environment although the original TSL specification is realizable. We call such a counter-strategy *spurious* as it exploits the additional freedom of LTL to violate the purity of predicates as made possible by the underapproximation. Formally, a counter-strategy is an infinite tree $\pi \colon \mathcal{C}^* \to 2^{\mathcal{T}_P}$, which provides predicate evaluations in response to possible update assignments of function terms $\tau_F \in \mathcal{T}_F$ to outputs $\circ \in \mathbb{O}$. W.l.o.g. we can assume that $\mathbb{O}$, $\mathcal{T}_F$ and $\mathcal{T}_P$ are finite, as they can always be restricted to the outputs and terms that appear in the formula. A counter-strategy is spurious, iff there is a branch $\pi \wr \varsigma$ for some computation $\varsigma \in \mathcal{C}^\omega$, for which the strategy chooses an inconsistent evaluation of two equal predicate terms at different points in time, i.e.,

$$\begin{aligned}
&\exists \varsigma \in \mathcal{C}^\omega.\ \exists t, t' \in \mathbb{N}.\ \exists \tau_P \in \mathcal{T}_P. \\
&\quad \tau_P \in \pi(\varsigma(0)\varsigma(1)\dots\varsigma(t-1)) \ \wedge\ \tau_P \notin \pi(\varsigma(0)\varsigma(1)\dots\varsigma(t'-1)) \ \wedge \\
&\quad \forall \langle\cdot\rangle \colon \mathbb{F} \to \mathcal{F}.\ \eta_{\langle\cdot\rangle}(\varsigma, \pi \wr \varsigma, t, \tau_P) \ =\ \eta_{\langle\cdot\rangle}(\varsigma, \pi \wr \varsigma, t', \tau_P).
\end{aligned}$$

Note that a non-spurious strategy can be inconsistent along multiple branches of the executions, as long as the branch is consistent with itself. That is to say, the environment may choose different function and predicate assignments for each system strategy it is trying to invalidate (from the game perspective).

Along a single branch however, the purity of predicates in TSL forces the environment to always return the same value for predicate evaluations on equal input values. This semantic property cannot be enforced implicitly in LTL. Effectively, purity is encoding an unbounded assumption on the input values (predicate evaluations) in LTL - which we cannot express with a finite LTL specification.

To resolve this issue we use the returned counter-strategy to identify spurious behavior in order to strengthen the LTL underapproximation with additional environment assumptions. As an example of this process, reconsider the spurious counter-strategy of Fig. 2.5c. Already after the first system response $[\![\mathtt{y} \leftarrowtail \mathtt{x}]\!]$, the environment produces an inconsistency by evaluating $\mathtt{p}\ \mathtt{x}$ and $\mathtt{p}\ \mathtt{y}$ differently. This is inconsistent, as the cell $\mathtt{y}$ holds the same value at time $t = 1$ as the input $\mathtt{x}$ at time $t = 0$. The user may then provide a new assumption $\square([\![\mathtt{y} \leftarrowtail \mathtt{x}]\!] \rightarrow (\mathtt{p}\ \mathtt{x} \leftrightarrow \bigcirc \mathtt{p}\ \mathtt{y}))$. After adding this assumption, which tightens the underapproximation from TSL, the LTL synthesizer returns a realizability result. The work of Finkbeiner et. al [12] presents details on an automated approach to adding these assumptions.

2.5.4 Undecidability

Although we can approximate the semantics of TSL with LTL, there are TSL formulas that cannot be expressed as LTL formulas of finite size. The intuition is that TSL is encoding the semantics of a "pure" function directly into the semantics of the logic, and therefore capturing more information than we have in the semantics of the Boolean based logic of LTL. In particular, a pure function always returns the same value on the same input. In order for us to model the environment of a TSL specification in LTL, we must ensure all predicates behave in a manner consistent with purity (a la the refinements presented in Fig. 2.5. For an incoming stream of predicate evaluations to match the semantics of purity, it then must remember which output of the predicate was chosen for every environmental input over time. Since we have an infinitely long stream of values, we may need to remember an evaluation of a predicate infinitely far into the past - then taking infinite memory

(infinitely large LTL specification), whence undecidability. A more formal proof of undecidability can be obtains through a reduction to an instance of the Post Correspondence Problem (PCP) [36].

Theorem 2 ([12])**.** The realizability problem of TSL is undecidable.

2.6 TSL Synthesis

Our synthesis framework provides a modular refinement process to synthesize executables from TSL specifications, as depicted in Fig. 2.1. The user initially provides a TSL specification over predicate and function terms. At the end of the procedure, the user receives an executable to control a reactive system.

The first step of our method answers the synthesis question of TSL: if the specification is realizable, then a control flow model is returned. To this end, an intermediate translation to LTL is used, utilizing an LTL synthesis solver that produces circuits in the AIGER format. If the specification is realizable, the resulting control flow model is turned into Haskell code, which is implemented as an independent Haskell module. The user has the choice between two different targets: a module built on Arrows, which is compatible with any Arrowized FRP library, or a module built on Applicative, which supports Applicative FRP libraries. Our procedure generates a single Haskell module per TSL specification. This makes naturally decomposing a project according to individual tasks possible. Each module provides a single component, which is parameterized by their initial state and the pure function and predicate transformations. As soon as these are provided as part of the surrounding project context, a final executable can be generated by compiling the Haskell code.

An important feature of our synthesis approach is that implementations for the terms used in the specification are only required after synthesis. This allows the user to explore several possible specifications before deciding on any term implementations.

2.6.1 Control Flow Model

The first step of our approach is the synthesis of a *Control Flow Model* $\mathcal{M}$ (CFM) from the given TSL specification φ, which provides us with a uniform representation of the control flow structure of our final program.

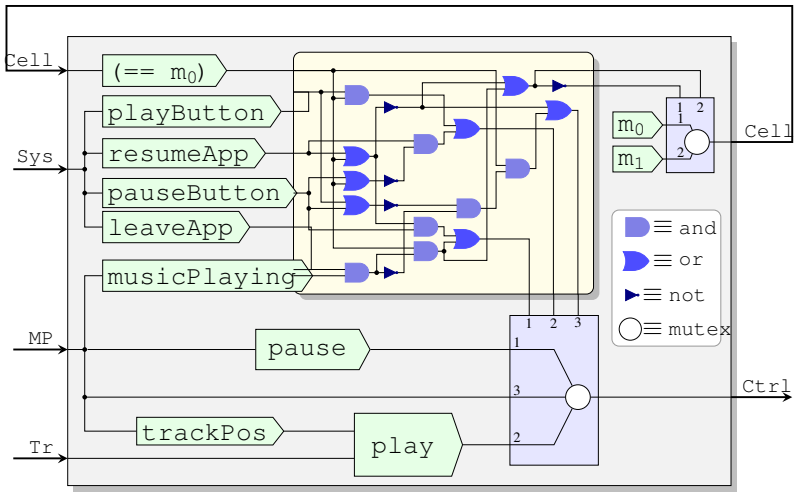


Figure 2.6: Example CFM of the music player generated from a TSL specification.

Formally, a CFM $\mathcal{M}$ is a tuple $\mathcal{M} = (\mathbb{I}, \mathbb{O}, \mathbb{C}, V, \ell, \delta)$, where $\mathbb{I}$ is a finite set of inputs, $\mathbb{O}$ is a finite set of outputs, $\mathbb{C}$ is a finite set of cells, V is a finite set of vertices, $\ell \colon V \to \mathbb{F}$ assigns a vertex a function $\mathtt{f} \in \mathbb{F}$ or a predicate $\mathtt{p} \in \mathbb{P}$, and

$$\delta \colon (\mathbb{O} \cup \mathbb{C} \cup V) \times \mathbb{N} \to (\mathbb{I} \cup \mathbb{C} \cup V \cup \{\bot\})$$

is a dependency relation that relates every output, cell, and vertex of the CFM with $n \in \mathbb{N}$ arguments, which are either inputs, cells, or vertices. Outputs and cells $\mathtt{s} \in \mathbb{O} \cup \mathbb{C}$ always have only a single argument, i.e., $\delta(\mathtt{s}, 0) \not\equiv \bot$ and $\forall m > 0.\, \delta(\mathtt{s}, m) \equiv \bot$, while for vertices $x \in V$ the number of arguments $n \in \mathbb{N}$ align with the arity of the assigned function or predicate $\ell(x)$, i.e., $\forall m \in \mathbb{N}.\, \delta(x, m) \equiv \bot \leftrightarrow m > n$. A CFM is valid if it does not contain circular dependencies, i.e., on every cycle induced by δ there must lie at least a single cell. We only consider valid CFMs.

An example CFM for our music player of Sec. 7.2 is depicted in Fig. 2.6. Inputs $\mathbb{I}$ come from the left and outputs $\mathbb{O}$ leave on the right. The example contains a single cell $\mathtt{c} \in \mathbb{C}$, which holds the stateful memory `Cell`, introduced during synthesis for the module. The green, arrow shaped boxes depict vertices V, which are labeled with functions and predicates names, according to ℓ. For the Boolean decisions that define δ, we use circuit symbols for conjunction, disjunction, and negation. Boolean decisions are piped to a multiplexer gate that selects the respective update streams. This allows each update stream to be passed to an output stream if and only if the respective Boolean trigger evaluates positively, while our construction ensures mutual exclusion on the Boolean triggers. For code generation, the logic gates are implemented using the corresponding dedicated Boolean

functions. After building a control structure, we assign semantics to functions and predicates by providing implementations. To this end, we use Functional Reactive Programming (FRP). Prior work has established Causal Commutative Arrows (CCA) as an FRP language pattern equivalent to a CFM [49–51]. CCAs are an abstraction subsumed by other functional reactive programming abstractions, such as Monads, Applicative and Arrows [49, 52]. There are many FRP libraries using Monads [28, 38, 53], Applicative [39, 40, 54, 55], or Arrows [41, 42, 56, 57], and since every Monad is also an Applicative and Applicative/Arrows both are universal design patterns, we can give uniform translations to all of these libraries using translations to just Applicative and Arrows. Both translations are possible due to the flexible notion of a CFM.

In the last step, the synthesized FRP program is compiled into an executable, using the provided function and predicate implementations. This step is not fixed to a single compiler implementation, but in fact can use any FRP compiler (or library) that supports a language abstraction at least as expressive as CCA. For example, instead of creating an Android music player app, we could target an FRP web interface [55] to create an online music player, or an embedded FRP library [54] to instantiate the player on a computationally more restricted device. By using the strong core of CCA, we even can directly implement the player in hardware, which is for example possible with the CλaSH compiler [40]. Note that we still need separate implementations for functions and predicates for each target. However, the specification and synthesized CFM always stay the same.

2.7 Experimental Results

To evaluate our synthesis procedure we implemented a tool that follows the structure of Fig. 2.1. It first encodes the given TSL specification in LTL and then refines it until an LTL solver either produces a realizability result or returns a non-spurious counter-strategy. For LTL synthesis we use the bounded synthesis tool BoSy [58]. As soon as we get a realizing strategy it is translated to a corresponding CFM. Then, we generate the FRP program structure. Finally, after providing function implementations the result is compiled into an executable.

To demonstrate the effectiveness of synthesizing TSL, we applied our tool to a collection of benchmarks from different application domains, listed in Table 2.1. Every benchmark class consists of multiple specifications, addressing different features of TSL. We created all specifications from

Table 2.1: Number of cells $|\mathbb{C}_{\mathcal{M}}|$ and vertices $|V_{\mathcal{M}}|$ of the resulting CFM $\mathcal{M}$ and synthesis times for a collection of TSL specifications φ. A * indicates that the benchmark additionally has an initial condition as part of the specification.

BENCHMARK (φ)	$\lvert\varphi\rvert$	$\lvert\mathbb{I}\rvert$	$\lvert\mathbb{O}\rvert$	$\lvert\mathbb{P}\rvert$	$\lvert\mathbb{F}\rvert$	$\lvert\mathbb{C}_{\mathcal{M}}\rvert$	$\lvert V_{\mathcal{M}}\rvert$	SYNTHESIS TIME (S)
Button								
default	7	1	2	1	3	3	8	0.364
Music App								
simple	91	3	1	4	7	2	25	0.77
system feedback	103	3	1	5	8	2	31	0.572
motivating example	87	3	1	5	8	2	70	1.783
FRPZoo								
scenario_0	54	1	3	2	8	4	36	1.876
scenario_5	50	1	3	2	7	4	32	1.196
scenario_{10}	48	1	3	2	7	4	32	1.161
Escalator								
non-reactive	8	0	1	0	1	2	4	0.370
non-counting	15	2	1	2	4	2	19	0.304
counting	34	2	2	3	7	3	23	0.527
counting*	43	2	2	3	8	4	43	0.621
bidirectional	111	2	2	5	10	3	214	4.555
bidirectional*	124	2	2	5	11	4	287	16.213
smart	45	2	1	2	4	4	159	24.016
Slider								
default	50	1	1	2	4	2	15	0.664
scored	67	1	3	4	8	4	62	3.965
delayed	71	1	3	4	8	5	159	7.194
Haskell-TORCS								
simple	40	5	3	2	16	4	37	0.680
advanced								
gearing	23	4	1	1	3	2	7	0.403
accelerating	15	2	2	2	6	3	11	0.391
steering								
simple	45	2	1	4	6	2	31	0.459
improved	100	2	2	4	10	3	26	1.347
smart	76	3	2	4	8	5	227	3.375

Table 2.2: Synthesis times and solution sizes for an equivalent LTL specification of the button example, synthesized by an LTL synthesis tool. The benchmarks increase in expressivity by increasing the maximum number of counter bits.

COUNTER BITS	1	2	3	4
TSL SYNTHESIS TIME (SEC)	0.080	0.080	0.080	0.080
LTL SYNTHESIS TIME (SEC)	0.031	0.0212	12.492	60+ timeout
SOLUTION STATES	2	4	8	16

scratch, where we took care that they either relate to existing textual specifications, or real world scenarios. A short description of each benchmark class is given in [44].

For every benchmark, we report the synthesis time and the size of the synthesized CFM, split into the number of cells ($|\mathbb{C}_{\mathcal{M}}|$) and vertices ($|V_{\mathcal{M}}|$) used. The synthesized CFM may use more cells than the original TSL specification if synthesis requires more memory in order to realize a correct control flow. The synthesis was executed on a quad-core Intel Xeon processor (E3-1271 v3, 3.6GHz, 32 GB RAM, PC1600, ECC), running Ubuntu 64bit LTS 16.04.

The experiments of Table 2.1 show that TSL successfully lifts the applicability of synthesis from the Boolean domain to arbitrary data domains, allowing for new applications that utilize every level of required abstraction. For all benchmarks we always found a realizable system within a reasonable amount of time, where the results often required synthesized cells to realize the control flow behavior.

For the sake of demonstration, we also synthesized the button example using an equivalent LTL specification. Since LTL can only express Boolean data streams, we used an explicit encoding of the counter stream, where the number of counter bits is bounded to a finite value. The results are presented in Table 2.2 and confirm the expected exponential blowup due to the additional overhead of handling the explicit counter representation.

2.8 Related Work

Our approach builds on the rich body of work on reactive synthesis, see [59] for a survey. The classic reactive synthesis problem is the construction of a finite-state machine that satisfies a specification in a temporal logic like LTL. Our approach differs from the classic problem in its connection to an actual programming paradigm, namely FRP, and its separation of control and data.

The synthesis of *reactive programs*, rather than finite-state machines, has previously been studied for standard temporal logic [60, 61]. Because there is no separation of control and data, these approaches do not directly scale to realistic applications. With regard to FRP, a *Curry-Howard correspondence* between LTL and FRP in a dependently typed language was discovered [62, 63] and used to prove properties of FRP programs [64, 65]. However, our paper is the first, to the best of our knowledge, to study the synthesis of FRP programs from temporal specifications.

The idea to separate control and data has appeared, on a smaller scale, in the synthesis with *identifiers*, where identifiers, such as the number of a client in a mutual exclusion protocol, are treated symbolically [66]. *Uninterpreted functions* have been used to abstract data-related computational details in the synthesis of synchronization primitives for complex programs [67]. Another connection to other synthesis approaches is our CEGAR loop. Similar *refinement loops* also appear in other synthesis appraches, however with a different purpose, such as the refinement of environment assumptions [68].

So far, there is no immediate connection between our approach and the substantial work on *deductive* and *inductive synthesis*, which is specifically concerned with the data-transformation aspects of programs [69–74]. Typically, these approaches are focussed on non-reactive sequential programs. An integration of deductive and inductive techniques into our approach for reactive systems is a very promising direction for future work. Abstraction-based synthesis [75–78] may potentially provide a link between the approaches.

Chapter 3

Synthesizing Functional Reactive Programs

Work presented in this chapter was completed in collaboration with Felix Klein, Bernd Finkbeiner, and Ruzica Piskac. Sections of this work have been previously published [12–14], and are reproduced here, at times in their original form.

Functional Reactive Programming (FRP) is a paradigm that has simplified the construction of reactive programs. There are many libraries that implement incarnations of FRP, using abstractions such as Applicative, Monads, and Arrows. However, finding a good control flow, that correctly manages state and switches behaviors at the right times, still poses a major challenge to developers.

An attractive alternative is specifying the behavior instead of programming it, as made possible by the recently developed logic: Temporal Stream Logic (TSL). However, it has not been explored so far how Control Flow Models (CFMs), resulting from TSL synthesis, are turned into executable code that is compatible with libraries building on FRP. We bridge this gap, by showing that CFMs are a suitable formalism to be turned into Applicative, Monadic, and Arrowized FRP.

We demonstrate the effectiveness of our translations on a real-world kitchen timer application, which we translate to a desktop application using the Arrowized FRP library `Yampa`, a web application using the Monadic `Threepenny-GUI` library, and to hardware using the Applicative hardware description language `ClaSH`.

```
yampaButton :: SF (Event MouseClick) Picture
yampaButton = proc click → do
  rec
    count    <- init 0 -< newCount
    newCount <- arr f1 -< (click, count)
    pic      <- arr f2 -< count
  returnA -< pic

f1 :: (Event MouseClick, Int) → Int
f1 (click, count)
  | isEvent click = count + 1
  | otherwise     = count

f2 :: Int → Picture
f2 = text . show
```

Figure 3.1: A button written with the FRP library `Yampa`.

3.1 Motivation

Building a reactive program is a complex process, of which the most difficult part is finding a good and correct high-level design [23]. Furthermore, even once this design has been fixed, implementing the system still remains a highly error-prone process [43]. While FRP helps with the latter problem by embedding the concept of time into the type system, it still leaves the challenge of switching between behaviors and managing state efficiently open.

A solution for solving the design challenge has been proposed with Temporal Stream Logic [44], a specification logic to specify the temporal control flow behavior of a program. The logic enforces a clean separation between control and data transformations, which also can be leveraged in FRP [38]. To demonstrate this, we give an example for a TSL specification:

$$\square\big((\texttt{event click} \leftrightarrow [\![\texttt{count} \leftarrowtail \texttt{increment count}]\!]) \\ \wedge [\![\texttt{screen} \leftarrowtail \texttt{display count}]\!]\big)$$

which states that the counter must always be incremented if and only if there is a click event, and that the value of the counter is displayed on a screen.

An implementation that satisfies the specified behavior, built using the FRP library `Yampa` [41], is shown in Fig. 3.1. The program implements a button in a GUI which shows a counter value that increments with every click. The `Yampa` FRP library uses an abstraction called arrows [79], where the arrows define the structure and connection between functions [50]. As mentioned before, they can be used to cleanly separate data transformations into pure functions, creating a visually clear

separation between the control flow and the data level. In the example program of Fig. 3.1, this separation is clearly visible. The `yampaButton` is the "arrow" part of the code, which defines a control flow. The functions `f1` and `f2` are the "functional" part, describing pure data transformations.

In the TSL specification, function applications, like `click`, `increment` or `display`, are not tied to a particular implementation. Instead, the synthesis engine ensures that the specification is satisfied for all possible implementations that may be bound to these placeholders, similar to an unknown polymorphic function that is used as an argument in a functional program. Thus, the implementation of Fig. 3.1 indeed satisfies the given specification by implementing `event` with `isEvent` of the `yampa` library, `increment` with `(+1)`, and `display` with `text` of the `gloss` library and `show`.

The immediate advantage of synthesis over manual programming is that, if the synthesis succeeds, then there is a guarantee that the constructed program satisfies the specification. Sometimes, the synthesis does not succeed, and this also leads to interesting results. An example is given by the FRPZoo [80] study, which consists of implementations for the same program for 16 different FRP libraries. The program consists of two buttons that can be clicked on by the user: a `counter` button, which keeps track of the number of clicks, and a `toggle` button, which turns the counter button on and off. To our surprise, after translating the written-English specification from the FRPZoo website into a formal TSL specification, the synthesis procedure was not able to synthesize a satisfying program. By inspecting the output of the synthesis tool, we noticed that the specification is actually unrealizable. The problem is that the specification requires the counter to be incremented whenever the `counter` button is clicked. At the same time, the counter must be reset to zero whenever the `toggle` button is clicked. This creates a conflict when both buttons are clicked together. To obtain a solution, we had to add the assumption that both buttons are never pressed simultaneously.

In chapter **??**, we discuss the synthesis process for creating the CFM in detail. In this chapter we explore this process of generating an FRP implementation of that CFM. In particular, we show how Causal Commutative Arrows (CCA) form a foundational abstraction for FRP in the context of program synthesis. From this connection between CCA and a CFM, we build a system to generate library-independent FRP across a range of FRP abstractions.

There is no single style of FRP which is generally accepted as canonical. Instead, FRP is realized through a number of libraries, which are based on fundamentally different abstractions, such as

Monadic FRP [53, 55, 81], Arrowized FRP [41, 57], and Applicative FRP [39, 40]. We show that our system is flexible enough to handle all of these abstractions, by demonstrating translations from a CFM to `Threepenny-GUI` [81], `Yampa` [41], and `ClaSH` [40].

We do not envision FRP synthesis as a replacement for FRP libraries, but rather as a complement. Through synthesis and code generation, users automatically construct FRP programs in these libraries that provide critical interfaces to input/output domains. Furthermore, we show that TSL synthesis generates code as expressive as CCA. While this power is sufficient for many applications, the FRP libraries still provide an interface to more powerful language abstraction features, in case the user chooses to use them.

In this chapter, we describe the process of automatically generating library-independent FRP control code from TSL specifications. We also examine the relation between CFMs and CCA, and compare the differences between various FRP abstractions during the translation process.

3.2 Functional Reactive Programming

One of the most common solutions for the construction of reactive systems in an imperative setting are call-back frameworks, embedded into a loop. The call-backs are either used to query the state of variables, or to change them. This imperative approach is well suited for rapid prototyping of small systems. However, tracing behaviors over time quickly becomes unmanageably complex for larger systems.

Functional Reactive Programming instead introduces an abstraction of time that allows the programmer to safely manipulate time-varying values. The core abstraction in FRP is that of a *signal*

$$Signal\ a = \mathbb{N} \to a$$

which produces values of some arbitrary type a over time. The type a can be an input from the world, such as the current position of the mouse, or an output type, such as some text that should be rendered to the screen. Signals are also used internally to manipulate values over time, for example if the position of the mouse should be rendered to the screen.

There are many incarnations of FRP, which use various abstraction to manipulate signals over time. One popular abstraction for FRP is a Monad, but a weaker abstraction, called Arrows, is also

Listing 3.1: Basic Arrowized FRP syntax

```
myDriver :: SF Image Steer
myDriver = proc image → do
  basicSteer    <-     turn  -< image
  adjustedSteer <- arr avoid -< (image, basicSteer)
  returnA -< adjustedSteer
```

used in many modern libraries [42, 56]. The Arrow abstraction describes a computation connecting inputs and outputs in a single type [79]. Hence, an Arrow type also allows us to describe reactive programs that process inputs and produce outputs over time.

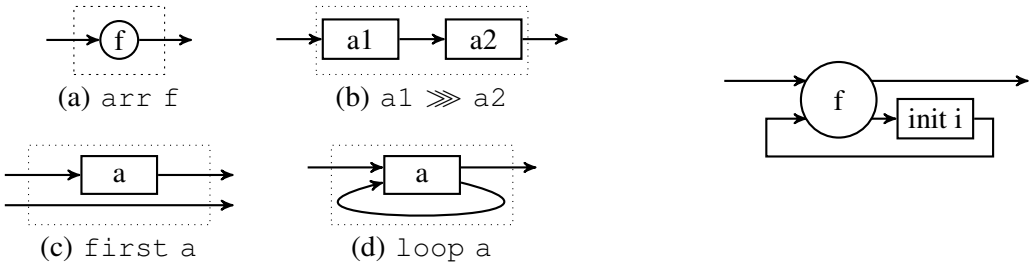
This abstraction of time can be realized through many types abstractions from type theory. For example monads allow for complex manipulation of signal flows [82]. We instead focus on the arrow abstraction, or so called Arrowized FRP [83]. Arrows generally run faster and with little need for manual optimization [51], but are fundamentally less expressive than a monadic FRP [52]. This more restrictive language is in fact a benefit, as it makes it harder for the programmer to introduce errors. At the same time, the syntax is clear and accessible enough to make for an easy introduction to the FRP paradigm, as well as a convenient target for synthesis.

Arrowized FRP was introduced to plug a space leak in the original FRP work [38, 50]. By using the Arrow abstraction introduced in [79], which describes in a single type inputs and outputs, we can also describe reactive programs that process inputs and produce outputs over time. At the top level, an Arrowized FRP program will have the form

$$SF\ Input\ Output = Signal\ Input \rightarrow Signal\ Output$$

which is a signal function type, parametrized by the type of input from the world and the type of output to the world. This is a transformer from one signal to another over time. As an example, imagine a type for an FRP function that controllers a steering wheel of an autonomous vehicle, which operates based on a video stream, such as `turn :: SF Image Steer`. This function processes video and uses it to decide how to steer. We omit an implementation, as the details of the data transformation are not relevant to the structure of the FRP code.

Haskell provides special syntax for Arrowized FRP, which mimics the structure of control flow charts. The syntax provides a composition environment, in which the programmer just manages the



(a) The core Arrow operators. Others, like `second`, are built from these.

(b) `loopD i f`: a special loop from CCA that is initialized with a user provided value `i`.

Figure 3.2: Common operators used to manipulate signals in Arrowized FRP.

composition of arrow functions. Inputs are read in from the right hand side, and piped to the left hand side (`output <– function –< input`).

A demonstration of this syntax for Arrowized FRP is given in Listing 3.1. The example introduces `avoid :: (Image, Steer) → Steer`, a pure function that adjusts the basic steering plan based on the image to avoid any obstacles. In Listing 3.1, this `avoid` function is lifted to the signal level using `arr :: (a → b) → SF a b`. The core Arrow operators in addition to `arr`, shown in Fig. 3.2a, are used to compose multiple arrows into larger programs. The function `turn` is already on the signal level (has an SF type). Hence, we do not need to lift it to the arrow level.

Stateful FRP To avoid obstacles on the road, we might write an `avoid2` function as shown in Listing 3.2, which requires two images to calculate the adjusted steering command. For this, we need a mechanism to maintain state between each processing step. Two images would be necessary to filter noise in the image, or calculates the velocity of an approaching obstacle. To implement it, we use an abstraction called `ArrowLoop` to save the previous state of the image for the next processing step. The syntax is presented in Listing 3.2. Intuitively, `ArrowLoop` gives us a recursive computation, as also indicated by the `rec` keyword. We elide the technical details for the purposes of this presentation and refer the interested reader to [84].

The predefined function `iPre` takes an initial state, in our case an empty image, and saves images for one time step, each time it is processed. This way, we create a feedback loop that is then used in the updated `avoid` function. At the same time, the `rec` keyword is used to denote a section

Listing 3.2: Using ArrowLoop to send feedback

```
myDriver :: SF Image Steer
myDriver = proc image → do
  rec
    oldI          <- iPre null  -< image
    basicSteer    <-      turn  -< image
    adjustedSteer <- arr avoid2 -< (image, oldI, basicSteer)
  returnA -< adjustedSteer
```

of arrow code with mutual dependencies. Without the keyword, there is an unresolvable dependency loop.

CCA We target a restricted set of arrows called Causal Commutative Arrows (CCA) [49, 51]. Specifically, CCA adds additional laws to arrows that constrain their behavior and the type of state they may retain. Of particular interest is that CCA also introduces a special initialization operator, `init`. This `init` operator allows for `loopD`, a loop that includes initialization, as shown in Fig. 3.2b.

We use CCA as a minimal language for synthesis. Our synthesis is able to support any FRP library which is at least as powerful as CCA. Because CCA is again restricted in its interface, there are more libraries that can simulate CCA FRP than Arrowized FRP in general. This makes our synthesis procedure possible for an even wider range of application scenarios. We revisit the implications of CCA as our choice of computation abstraction in Sec. 3.3.1.

3.2.1 Connections between FRP and Reactive Systems

A first inspection reveals that FRP fits into the definition of a reactive system, as given in Sec. 2.3: an FRP program reads an infinite stream of input signals and finally produces a corresponding infinite output stream. Nevertheless, FRP does not fit into the classical setting used for reactive systems, as the input and output streams in FRP are allowed to have arbitrary types.

To solve this problem, one could restrict FRP to just streams of enumerative types, which then are reduced to a Boolean representation. However, this would drop the necessity of almost all interesting features of FRP and it is questionable whether this restricted notion of FRP would give any benefits against Mealy/Moore automata or circuits, which are already used for reactive systems. Additionally, it just creates an exponential blowup and does not provide any insights into the core problem.

```
control =
  rec
-- gather values from the previous cell value
    ∀c ∈ ℂ. c ← δ(c, 0)
-- gather applications of 𝔽 and ℙ
    ∀v ∈ (V ∩ (𝔽 ∪ ℙ)). v ← (δ(v, 0), ..., δ(v, n))
-- compute control signals from cs, vs, and ctrls
    ∀ctrl ∈ (V ∩ 𝕃). ctrl ← (δ(ctrl, 0), ..., δ(ctrl, n))
-- use control signals to select from 𝔽 and ℙ applications
    ∀m ∈ (V ∩ 𝕌). m ← (δ(m, 0), ..., δ(m, n))
-- output signals take the signals from either the cs, vs, or ms as specified by
    δ
  return ∀o ∈ 𝕆. o ← δ(o, 0)
```

Figure 3.3: The general code template for the control block of the synthesized FRP program. The exact syntax for `rec`, assignment, and outputting signals varies across different FRP abstractions.

Hence, it is more reasonable and interesting to ask whether it is possible to natively handle streams of arbitrary type within reactive systems. Recall that FRP includes functional behavior, defined using standard functional paradigms, but also a control structure, defined via arrows and loops. We will target synthesis of the control structure, leaving the functional level synthesis to tools such as MYTH [70, 71] or SYNQUID [85].

In order to have a more clear connection between notions of FRP and TSL during the subsequent sections, we will introduce some common notation. We assume time to be discrete and denote it by the set $\mathbb{N}$ of positive integers. A value is an arbitrary object of arbitrary type, where we use $\mathcal{V}$ to denote the set of all values. We consider the Boolean values $\mathcal{B} \subseteq \mathcal{V}$ as a special subset, which are either $\texttt{true} \in \mathcal{B}$ or $\texttt{false} \in \mathcal{B}$.

A signal $s \colon \mathbb{N} \to \mathcal{V}$ is a function fixing a value at each point in time. The set of all signals is denoted by $\mathcal{S}$, usually partitioned into input signals $\mathcal{I}$ and output signals $\mathcal{O}$.

An n-ary function $f \colon \mathcal{V}^n \to \mathcal{V}$ determines a new value from n given values. We denote the set of all functions (of arbitrary arity) by $\mathcal{F}$. Constants are functions of arity zero, while every constant is also a value, i.e, an element of $\mathcal{F} \cap \mathcal{V}$. An n-ary predicate $p \colon \mathcal{V}^n \to \mathcal{B}$ checks a truth statement on n given values. The set of all predicates (of arbitrary arity) is denoted by $\mathcal{P}$.

3.3 Code Generation

We present a system for FRP program generation from synthesized CFMs. The user initially provides a CFM that was synthesized from a TSL specification over a set of predicate and function terms. The user specifies a target FRP abstraction, and receives an executable Haskell FRP program in the library of their choosing.

Our approach takes a multi-stage approach, whereby the TSL specification is used to generate a control flow model (CFM). The CFM is an abstract representation of the temporal changes that the FRP program must implement in order to satisfy the TSL specification. In particular, a CFM maps the input signals through various function and predicate terms to the output signals. We only consider valid CFMs, where for every cycle created, by mapping an output back to an input, there is at least one *cell*. A cell is a memory unit with delay, *i.e.* , at one moment a value may be stored, and at the next that value is retrieved. The concept of cells is analogous to ArrowLoop [86], or registers in hardware.

The synthesis from TSL to CFM is the most computationally expensive and may result in unrealizability, in which case synthesis terminates with no solution. We omit a detailed description of the generation of the CFM from a TSL specification, and instead direct the reader to [44]. Here we focus on how the generality of the CFM is utilized to generate framework-independent FRP code. If a satisfying CFM is found during the first stage, a user specifies a target FRP abstraction (Applicative, Monad, Arrow) that is used to generate the FRP program code from the CFM.

Given a CFM that satisfies the TSL specification, we convert it into a template for our FRP program. The code implementing the CFM is given in Fig. 3.3. The CFM is transformed via a syntactic transformation into an FRP program in the abstraction of the user's choice, as a function that is parameterized over the named function and predicate terms, as shown in Fig. 3.4. The user then provides implementations of the function and predicate terms that complete the construction of the FRP program based on the generated template.

The CFM transformation is modularized to fit any FRP library that is at least as powerful as CCA [42, 53, 56, 80, 87]. The key insight is that first-order control along with a loop describes the expressive power of both CCA and the CFM model generated from the techniques of [44]. Although many FRP libraries support more powerful operations than CCA, *e.g.* , `switch` in `Yampa`, we do

```
control
  :: _ signal  -- FRP abstraction
  ⇒ _          -- cell implementation
  → (_ → _)    -- functions and predicates
  → _          -- initial values
  → signal _   -- input signals
  → signal _   -- output signal
```

(a) The general template of the type signature for the control block of the synthesized FRP program.

```
control
  :: Applicative signal
  ⇒ (∀ p. p → signal p → signal p)
  → (a → Bool)   -- press
  → (b → c)      -- display
  → (b → b)      -- increment
  → b            -- initial value: count
  → c            -- initial value: screen
  → signal a     -- button (input)
  → ( signal b   -- count (output)
    , signal c   -- screen (output)
    )
```

(b) An example instantiation of the type signature for the control block of the button (as described in the introduction) as it has been specialized for Applicative FRP.

Figure 3.4: The control block follows a general type signature template across FRP abstractions.

not need to utilize these in the synthesis procedure, and thus can generalize synthesis to target any FRP library that is at least as expressive as CCA.

Recall that in TSL, output signals can be written at the current time t, and be read from at time $t + 1$. To implement this in the FRP program, we use the concept of a cell in the CFM. In the translation, we allow a space for the user to provide an implementation of the cell that is specialized to their FRP library of choice (as shown in Fig. 3.4). In the case of CCA, this is the `loopD` combinator. The `loopD` combinator pipes the output values back to the input to allow them to be read at time $t + 1$. Since a system may require output values at $t = 0$, the user must also provide initial values to $\mathbb{O}$.

3.3.1 Properties

In the translation of the CFM, we use Casual Commutative Arrows (CCA) as the target conceptual model. Understanding the implications of using CCA as an underlying model to connect TSL to FRP allows us to gain insight into the expressive power and limitations of using TSL synthesis to construct FRP programs. One interesting note about this is that CCA does not allow the `arrowApply`

function, enforcing a static structure on the generated program. The `arrowApply` (also called `switch`) function is a higher-order arrow that allows for dynamically replacing an arrow with a new one that arrives on an input wire. While `switch` is a very expressive operation, it also comes with drawbacks. First, dynamically evolving networks cannot provide run-time guarantees for memory requirements in general, while static networks do. Second, the behavior of a dynamically evolving network is hard to grasp in general, which especially makes them unamenable for verification. Third, the use of dynamic networks is largely impractical for FRP applications with restricted resources, as for example applications that are executed on embedded devices [88] or are implemented directly in hardware [40]. An insight provided by prior work on CCA [49] was that, in general, the expressive power of higher-order arrows makes automatic optimization more difficult. Furthermore, for most FRP programs, first-order switch is more than enough [89].

The lack of support for `arrowApply` in CCA is mirrored as a fundamental restriction in the synthesis of systems in TSL. This is because in TSL updates are fixed by the specification, so too must the arrow structure be fixed in synthesis. With TSL, every update term $[\![x \leftarrowtail y]\!]$ is directly lifted to an arrow that updates x with y over time. Note that having a fixed arrow structure disallows higher-order arrows, but higher-order functions can still be passed along wires. As an example, we may have a function term `app :: (a → b) → a → b` and signals `f :: a → b` and `x :: a`. A simple specification making use of higher order functions then could state that the system should always apply the incoming higher-order function to the incoming value: $\square[\![\texttt{x} \leftarrowtail \texttt{app(f, x)}]\!]$.

Additionally, a key difference between arrows and circuits is that arrows are able to carry state that tracks the application of each arrow block. By using CCA, a user may write TSL specifications about stateful arrows that are still handled correctly by the synthesis procedure. To this end, we only synthesize programs that obey the *commutativity* law [49, 51] restated below that ensures that arrows cannot carry state influencing the result of composed computations.

$$first\ f \ggg second\ g = second\ g \ggg first\ f$$

Imagine an arrow with a global counter to track data of a buffer. Since addition is commutative, this arrow respects the commutativity law. However, non-commutative state is possible as well. For example, when building GUIs with Arrowized FRP [57], the position of each new UI element depends on the order of the previously laid out elements. Due to the commutativity of the Boolean

operators, the commutativity of CCA is a necessary precondition for synthesis of a TSL specification. Specifically, the commutativity of logical conjunction allows the solution to update signals in any arbitrary order. Thus, the correctness of the TSL synthesis relies on commutativity of composition, which is naturally modeled with CCA's commutativity law.

3.4 Related Work

There are various lines of work that are related to our approach. While we draw inspiration from these research directions, each one, on its own, addresses a different type of problem.

3.4.1 Temporal Types for FRP

FRP is a programming paradigm for computations over time, and, hence, a natural extension is to investigate type systems to be able to reason about time. A correspondence between LTL and FRP in a dependently typed language was discovered simultaneously by [62, 63]. In this formulation, FRP programs are proofs and LTL propositions are reactive, time-varying types that describe temporal properties of these programs. In establishing the connection between logic and FRP, these LTL types are also used to ensure causality and loop-freeness on the type level.

Dependent LTL types are a useful extension to FRP that provides insight into the underlying model of FRP, but does not lend itself to control flow synthesis. In the work of Jeffery and Jeltsch, the types describe the input/output change over time for each arrow. Using these LTL types, only arrows adhering to sensible temporal orderings (*e.g.* computations only depend on past values) will be well typed. However, as with any other FRP system, the temporal control flow of function applications in the program is fixed by the code. A similar approach was used by [65] to make a temporal type system that ensures there are no space-time leaks in a well typed FRP program. Work on reasoning about FRP using temporal logics also includes [90], although this setting considered dynamic network structure, as allowed by higher-order arrows. While the above works apply temporal types for functional correctness, type extensions have also been used to encode fairness properties [64], to ensure that any well-typed system will eventually perform actual work.

In contrast, we use the logic TSL, for a fine-grained description of function application behavior which cannot be expressed within pure LTL. The synthesis procedure of TSL determines a temporal

control flow of functions, where the TSL specifications determines the transformations to be applied at each point in time. In addition to the logical specifications, the synthesis is also constrained by the types of functions appearing in the specification. Since the types of all functions are fixed at all times, the type system can be lifted to the specification. If the specification is well typed, synthesis is guaranteed to yield a well typed program.

One connection to our work however is the implications of the fact that the Curry-Howard correspondence extends to FRP and LTL. In the aforementioned work, LTL propositions are types for FRP programs. If a proof of a TSL proposition can be interpreted as a program, one might expect that there is some corresponding type system to TSL. We leave such explorations to future work.

3.4.2 Synthesis of Reactive Programs

A distinguishing feature of our approach is the connection to an actual programming paradigm, namely FRP. Most reactive synthesis methods instead target transition systems or related formalisms such as finite state machines. The idea to synthesize programs rather than transition systems was introduced in [60]. In his work, an automaton is constructed that works on the syntax tree of the program, which makes it possible to obtain concise representations of the implementations, and to determine how many program variables are needed to realize a particular specification. Unlike our FRP programs, Madhusudan's programs only support variables on a finite range of instances.

Another related approach is the synthesis of synchronization primitives introduced in [30] for the purpose of allowing sequential programs to be executed in parallel. Similar to our synthesis approach, uninterpreted functions are used to abstract from implementation details. However, both the specification mechanism (the existing program itself is the specification) and the type of programs considered are completely different from TSL and FRP.

3.4.3 Logics for Reactive Programs

Many logics have been proposed to specify properties of reactive programs. Synthesis from Signal Temporal Logic [91] focuses on modeling physical phenomena on the value level, introducing continuous time and resolving to a system of equations. The approach allows for different notion of

data embedded into the equations. While more focused on the data level, the handling of continuous time might provide inspiration for future extensions to explicitly handle continuous time.

Another logic that has been proposed, Ground Temporal Logic [92], is a fragment of First Order Logic equipped with temporal operators, where it is not allowed to use quantification. Satisfiability and validity problems are studied, with the result that only a fragment is decidable. However, specifications expressed in Ground Temporal Logic, as well as their motivations, are completely different from our goals.

3.4.4 Reasoning-based Program Synthesis

Reasoning-based synthesis [69–72] is a major line of work that has been mostly, but not entirely, orthogonal to reactive synthesis. While reactive synthesis has focused on the complex control aspects of reactive systems, deductive and inductive synthesis has been concerned with the data transformation aspects in non-reactive, sequential programs. Our work is most related to Sketching [72]. In Sketching the user provides the control structure and synthesizes the transformations while, in TSL we synthesize the control and leave the transformations to the user.

The advantage of deductive synthesis is that it can handle systems with complex data. Its limitation is that it cannot handle the continuous interaction between the system and its environment, which is typical for many applications, such as for cyber-physical systems. This type of interaction can be handled by reactive synthesis, which is, however, typically limited to finite states and can therefore not be used in applications with complex data types. Abstraction-based approaches can be seen as a link between deductive and reactive synthesis [75, 76].

Along the lines of standard reactive synthesis, our work is focused on synthesizing control structures. We extend the classic approach by also allowing the user to separately provide implementations of data transformations. This is useful in the case where the value manipulations are unknown or beyond the capability of the synthesis tool. For example, a user may want to synthesize an FRP program that uses closed source libraries, which may not be amenable to deductive synthesis. In this case, the user can only specify that certain functions from that API should be called under certain conditions, but cannot and may not want to reason about their output.

3.5 Limitations

In this work we have presented a detailed account of how to transform Control Flow Models into framework-independent FRP code. With this transformation, we utilize TSL synthesis as presented in [44] to build a complete toolchain for synthesizing Functional Reactive Programs. Using TSL specifications prior to manually programming improves designing the underlying control flow. The developer is immediately notified about conflicts in the current design and supported by the feedback returned from synthesis for resolving them.

So far, we have used a discrete time model in our formalization, however, the behavior of the kitchen timer is in fact sampling rate independent (Continuous Time FRP). Sampling rate independence is guaranteed in TSL as long as the next operator is not used. However, the relation between TSL with the next operator and Continuous Time FRP still needs to be explored.

In another direction, the usual way in FRP to distinguish between continuous and discrete behaviors is to use signals and events. So far we have embedded data into signals. It is open to future work how to utilize events natively. Future directions for improvements to usability include integrating FRP synthesis more tightly with programming, *e.g.* , by allowing specifications to be used inline with QuasiQuoters [93].

Chapter 4

Case Study: Synthesizing Autonomous Vehicle Controllers

Work presented in this chapter was completed in collaboration with Felix Klein, Bernd Finkbeiner, and Ruzica Piskac. Sections of this work have been previously published [12–14], and are reproduced here, at times in their original form.

Functional languages have provided major benefits to the verification community. Although features such as purity, a strong type system, and computational abstractions can help guide programmers away from costly errors, these can present challenges when used in a reactive system. Functional Reactive Programming is a paradigm that allows users the benefits of functional languages and an easy interface to a reactive environment. We present a tool for building autonomous vehicle controllers in FRP using Haskell.

4.1 Motivation

Autonomous vehicles are considered to be one of the most challenging types of reactive systems currently under development [94–96]. They need to interact reliably with a highly reactive environment and crashes cannot be tolerated. Life critical decisions have to be made instantaneously and need to be executed at the right point in time.

The development of autonomous vehicles and other cyberphysical systems is supported by a wide spectrum of programming and modeling methodologies, including synchronous programming

languages like Lustre [97] and Esterelle [98], hardware-oriented versions of imperative programming languages like SystemC [99], and visual languages like MSCs and Stateflow-charts [100, 101]. The question of which programming paradigm is best-suited to write easy-to-understand, bug-free code is still largely unresolved.

In the development of other forms of critical software, outside the embedded domain, developers increasingly turn to functional programming (cf. [102]). The strong type system in functional languages largely eliminates runtime errors [103]. Higher-order functions like `map` often eliminate the need for explicit index counters, and, hence, the risk of "index out of bounds" errors. Functional purity reduces the possibility of malformed state that can cause unexpected behavior.

While mathematical models of embedded and cyberphysical systems often rely on functional notions such as stream-processing functions [104, 105], the application of functional programming in the practical development of such systems has, so far, been limited. One of the most advanced programming language in this direction is Ivory, which was used in the development of autonomous vehicles [106]. Ivory is a restricted version of the C programming language, embedded in Haskell. It provides access to the low level operations necessary for embedded system programming, but still enforces good programming practice, such as disallowing pointer arithmetic, with a rich type system.

Ivory does not, however, have an explicit notion of time. It cannot deal directly with the integration of continuous and discrete time, which is fundamental for the development of a cyberphysical system. For example, in a car, continuous signals, such as the velocity or acceleration, mix with the discrete steps of the digital controller.

In this paper, we investigate the use of functional programming in a domain where the interaction between continuous and discrete signals is of fundamental importance. We build a vehicle controller capable of both autonomous vehicle control and multi-vehicle communication, such as the coordination in platooning situations.

We have built a library, Haskell-TORCS, to use FRP to control a vehicle inside a simulation. The library interfaces Haskell FRP programs to TORCS, The Open Racing Car Simulator, an open-source vehicle simulator [107]. TORCS has been used in the Simulated Car Racing Championship competition [108], as well as other autonomous vehicle research projects [109–112]. Through Haskell-TORCS, the Haskell program has access to the sensors and actuators of the car, as well as

Figure 4.1: A screenshot of Haskell controlling the autonomous vehicle in the TORCS simulator.

communication channels between different vehicles. Such a simulator is a critical component of modern autonomous vehicle research, especially towards the goal of safe platooning algorithms [113].

Notably, TORCS has even been used for formal verification of platoons [109, 113]. None of these works have used FRP as the language for the controller. With the assistance of FRP, we build vehicle controllers in a principled way that allows users to manipulate sensor data in a transparent and well structured environment.

To the best of our knowledge this is the first FRP-based vehicle simulator. Although there are many bindings to various vehicle simulators, these tend to use imperative languages. For instance, TORCS allows users to directly edit the source code and add a new car in C++. There are also TORCS bindings for Python, Java, and Matlab, which have been used in the SCRC competition [108].

FRP specifically has been proposed as a tool for vehicle control [114, 115], where FRP was extended to prioritize functions for timing constraints. However, due to the lack of a compatible simulator, the vehicle simulation never was implemented. FRP has also been used for embedded systems [54] and networking [116]. The FRP networking library took advantage of Haskell's multicore support and significantly outperformed competing tools written in C++ and Java.

Here, we first introduce our library for TORCS, Haskell-TORCS, and demonstrate its use through manual implementation of controller for a solo car, and another for multi-vehicle platooning using a communication channel between the cars. The ability to run full simulations for solo and platooning vehicles is a critical piece to advancing the state of the art in using FRP for autonomous vehicle control. We then show how we use TSL synthesis from chapter **??** to synthesis a controller in Haskell-TORCS.

4.2 Haskell-TORCS

TORCS, The Open Racing Car Simulator, is an existing open source vehicle simulator [107] that has bindings for various languages [108]. We provide the first bindings for Haskell, and further extend this into a full library for multi-vehicle simulations. The library is an open source library, called Haskell-TORCS, and publicly available at `https://hackage.haskell.org/package/TORCS`. We now explain the functionality provided by our library, and highlight the ability of FRP to create modular and flexible controllers with clean code for autonomous vehicles.

4.2.1 Basics

To interface with Haskell-TORCS, a user must implement a controller that will process the `CarState`, which contains all the data available from the sensors. The controller should then output a `DriveState`, which contains all the data for controlling the vehicle. This transformation is succinctly described as the now familiar *signal function*. The core functionality of Haskell-TORCS is captured in the function `startDriver`, which launches a controller in the simulator. This function automatically connects a `Driver` to TORCS, which results in continuous `IO()` actions, the output type of this function.

```
type Driver = SF CarState DriveState
startDriver :: Driver → IO ()
```

The sensor and output data structures contain all the typical data available in an autonomously controlled vehicle. `CarState` includes fields like `rpm` to monitor the engine, or `track` to simulate an array of LiDAR sensors oriented to the front of the vehicle. `DriveState` includes fields like `accel` to control the gas pedal, or `steering` to control the angle of the steering wheel. A full description of the interface is available in the Simulated Car Racing Competition Manual [117].

4.2.2 Case Study : Driving

As a demonstration of the Haskell-TORCS library in use, we implemented a simple controller, shown in Listing 4.1. The code is complete and immediately executable as-is together with an installation of TORCS. Our controller successfully navigates, with some speed and finesse, a vehicle on track, as

shown in Fig. 4.1 along with a video demonstration[1]. The controller uses `ArrowLoop` to keep track of the current gear of the car. Although the gear is available as sensor data, it is illustrative to keep track locally of this state. In general, the `ArrowLoop` can be used to maintain any state that may be of interest in a future processing step. Additionally, notice all of the data manipulation functions are pure, and lifted via the predefined function `arr`.

One major advantage of FRP is this separation of dependency flow and data level manipulation. This abstraction makes it possible to easily reason about each of the components without worrying about confounding factors from the other. For example, if a programmer wants to verify that the steering control is correct, it is semantically guaranteed that the only function that must be checked is `steering`. Because of Haskell's purity, this is the only place where the steering value is changed. This significantly reduces the complexity of verification or bug tracking in case of an error.

4.2.3 Case Study : Communication for Platoons

Thanks to functional languages' exceptional support for parallelism, controlling multiple vehicles in a multi-threaded environment is exceedingly simple. In our library API, the user simply uses `startDrivers` rather than `startDriver`, and passes a list of `Driver` signal functions "driving" together. In this way, we easily let various implementations race against each other, or build a vehicle platooning controller. In the latter, the user can even extend the implementation to simulate communication between the vehicles.

Our library already provides a simple interface for simulating communication between vehicles. In order to broadcast a message to the other vehicles in the simulation, the controller simply writes a message to the `broadcast` field of `DriveState`. That message is then sent to all other vehicles as soon as possible, and received in the `communication` field of the input `CarState`.

A fragment of communication code is given in Listing 4.2, to pass messages between vehicles. In this fragment, a vehicle checks if a collision is imminent, and can request for the other cars in the platoon to go faster and move out of the way. Every vehicle also checks if any other car has requested for the platoon to speed up, and will adjust its own speed accordingly. These functions can be added to a controller, like the one in Listing 4.1, with little effort.

1. `http://www.marksantolucito.com/torcsdemo`

Listing 4.1: A complete basic controller for an autonomous vechile in Yampa using Haskell-TORCS.

```
{-# LANGUAGE Arrows, MultiWayIf, RecordWildCards #-}
module TORCS.Example where
import TORCS.Connect
import TORCS.Types

main = startDriver myDriver

myDriver :: Driver
myDriver = proc CarState{..}  → do
  rec
    oldG <- iPre 0 -< g
    g <- arr shifting -< (rpm, oldG)
    s <- arr steering -< (angle, trackPos)
    a <- arr gas -< (speedX, s)
  returnA -< defaultDriveState
    { accel = a, gear = g, steer = s }

shifting :: (Double, Int) → Int
shifting (rpm, g) = if
  | rpm > 6000 → min 6 (g + 1)
  | rpm < 3000 → max 1 (g - 1)
  | otherwise  → g

steering :: (Double, Double) → Double
steering (spd, trackPos) = let
  turns = spd * 14 / pi
  centering = turns - (trackPos * 0.1)
  clip x = max (-1) (min x 1)
 in
  clip centering

gas :: (Double, Double) → Double
gas (speed, steer) =
  if speed < (100 - (steer * 50)) then 1 else 0
```

Listing 4.2: Communicating between controllers

```
request :: Double → Message
request dist =
  if dist < 3 then "faster" else ""

adjustSpeed :: (Communications, Double) → Double
adjustSpeed (comms, oldSpeed) =
  if any (map (== "faster") comms) then s + 10 else s
```

We allow all vehicles in the simulation to communicate irrespective of distance and with zero packet loss. However, users are free to implement and simulate unreliable communications, or distance constraints.

4.2.4 Implementation

Haskell-TORCS uses Yampa [41] as the core FRP library, though its structure can easily be adapted to any other Haskell FRP library.

TORCS uses a specialized physics engine for vehicle simulations, that includes levels of detail as fine grained as tire temperatures effect on traction. When TORCS is used in the Simulated Car Racing Championship competition [108], each car is controlled via a socket that sends the sensor data from the vehicle and receives and processes the driving commands. So too, Haskell-TORCS communicates over these sockets to control vehicles inside the TORCS simulations.

In addition to the core controller functionality, we have also augmented Haskell-TORCS with the ability to test vehicle platooning algorithms that utilize cross-vehicle communication. The communication channels are realized via a hash map, using the `Data.Hashmap` interface, from vehicle identifiers to messages. Each vehicle is given write permissions to their unique channel, where all other vehicles have read-only permissions. The access is mutually exclusive, which is ensured by Haskell's `MVar` implementation, a threadsafe shared memory library. This ensures that there will never be packet loss in the communication.

4.3 Synthesis with Haskell-TORCS

Our specification for the autonomous vehicle controller builds upon the previously introduced Haskell-TORCS bindings. The bindings are also used to run the synthesized implementations within

the simulator. Autonomous vehicles use limited sensor data about the environment (*e.g.* the distance to nearest obstacle) to control actuators in the car (*e.g.* the steering wheel). The Haskell-TORCS set of benchmarks synthesize a controller from TSL specifications where the sensors and actuators are the input and output signals respectively. The functions used in the TSL specifications for the Haskell-TORCS benchmarks, for example "slowDown" or "turnLeft", are implemented after the controller synthesis process. In this way, we obtain a guarantee on the larger behavior of the system, while still allowing numerically sensitive, data level manipulations, to be optimized as required by the application.

The first "simple" Haskell-TORCS controller combines simple functions without states. The "advanced" controllers included more detailed planning behavior when approaching a turn. The specifications are also modular, in the sense that control of the steering wheel and control of the gears are given separate specifications, and combined into a single FRP program after synthesis.

Part II ❖ Synthesis for Implementation

Previously, we demonstrate how abstracting away from data transformation details allows us to refocus the efforts of synthesis on high-level structural design. In this part we explore new techniques for synthesis on the implementation level, filling the gap left by TSL synthesis. In particular, we introduce the concept of Digital Signal Processing Programming by Example (DSP-PBE). We introduce the problem of digital signal processing programming by example (DSP-PBE), where users specify input and output wave files, and a tool automatically synthesizes a program that transforms the input to the output. This program can then be applied to new wave files, giving users a new way to interact with music and program code. We formally define the problem of DSP-PBE, and provide a first implementation of a solution that can handle synthesis over commutative filters.

As audio data is an information rich format (a one second stereo audio clip of 64-bit audio at 96kHz is roughly 1.5 MB), typical examples provided by users are much larger than more traditional PBE domains. In turn the program space we much explore is also much larger. In additional to the space of distinct abstract syntax trees, the syntax trees are parameterized by values in a continuous domain.

While PBE is promising interaction paradigm, synthesis is still too slow for realtime interaction and more widespread adoption. Existing approaches to PBE synthesis in traditional data manipulation domains have used automated reasoning tools, such as SMT solvers, as well as works applying machine learning techniques. At its core, the automated reasoning approach relies on highly domain specific knowledge of programming languages. On the other hand, the machine learning approaches utilize the fact that when working with program code, it is possible to generate arbitrarily large training datasets. We propose a system for combining the symbolic approach of automated reasoning with the subsymbolic techniques of machine learning to solve Syntax Guided Synthesis (SyGuS) style PBE problems. By preprocessing SyGuS PBE problems with a neural network, we reduce the size of the search space, allowing automated reasoning-based solvers to more quickly find a solution analytically. Our system is able to run atop existing SyGuS PBE synthesis tools, decreasing the runtime of the winner of the 2019 SyGuS Competition for the PBE Strings track by 47.65% to outperform all of the competing tools.

Chapter 5

Digital Signal Processing Programming-by-Example

Work presented in this chapter was completed in collaboration with Aedan Lombardo, Kate Rogers, and Ruzica Piskac. Sections of this work have been previously published [15], and are reproduced here, at times in their original form.

5.1 Introduction

The great proliferation of computer music programming languages points to the difficulty of building a natural interface for users that want to computationally interact with musical data. Programming applications in the domain of computer music, and specifically digital signal processing (DSP), requires that users not only grasp fundamental programming techniques, but also have a large domain specific knowledge of time and signal manipulations. The amount of prerequisite skill and effort to overcome these barriers is often higher than many users are able to commit.

Furthermore, the difficulty of programming DSP applications is often not commensurate with the scope of the creative intentions. A simple creative choice may require a disproportionate technical effort. As a motivating example, imagine a user hears a sample in a piece of music, and again hears the same sample later in the piece with some added effects. In order to reuse this effect in the user's own musical composition, the user must now reconstruct the filter that was used to transform an audio clip. In this case the user has the original audio file, and the transformed audio file, but does

not know exactly how this transformation happened. In the standard approach, a user would need to be a domain expert and listen to the two files, and aurally estimate which kinds of filters were used to achieve the transformation. Once the user has some suspicion as to the appropriate filter types that will be needed, the user must write a program in some language (SuperCollider [118], CSound [119], PureData [120], etc) to implement the DSP filter the user has in mind. Further still, the user will then need to spend time tweaking the filter parameters to find the best fit.

To simplify this process, we introduce *DSP programming by example* (DSP-PBE). With DSP-PBE, the user simply provides our tool with the original audio (input), and the transformed audio (output), and the tool will automatically construct a DSP filter that approximates the transformation.

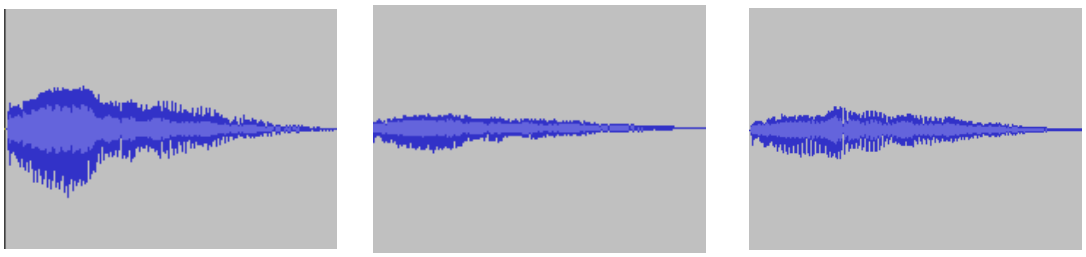
We formally define the problem of DSP programming by example as follows: Given an input waveform I and an output waveform O, construct a DSP filter $\mathcal{F}$, to minimize the aural distance $dist$ between O and $\mathcal{F}(I)$. In a single line,

$$\text{Find } \mathcal{F}, \text{ such that } dist(O, \mathcal{F}(I)) = 0$$

In the sequel we describe our approach to the two key components of this statement; the definition of distance, and a search technique to find $\mathcal{F}$. A distance metric that is faithful to the psycho-acoustics of the human ear is critical for a useful DSP-PBE tool. As an example, taking a trivial distance function that returns the difference in length of the two audio samples will allow a delay filter to satisfy any example pair of samples. Additionally, an efficient search algorithm is critical, as the space of possible DSP filters is very large. Not only do we need to consider a wide variety of filters, we need to consider the space of parameters for each filter, as well as the different ways of combining multiple filters.

5.2 Motivating Example

As a motivating example, imagine a user was to reconstruct the filter that was used to transform an audio clip, as shown in Figure 5.1. In this example, a user provided a clip of a `cartoon-spring.wav` in Figure 5.1a, and the same sound as it had been transformed with a low-pass filter at 800 Hz, $lpf(800)$, as shown in Figure 5.1b. However the nature of the transformation is unknown to the



(a) Input example (b) Output example (c) Generated

Figure 5.1: The waveforms (a) and (b) are provided as examples, and DSP-PBE synthesizes a filter that produces (c).

user and they wish to discover the filter needed. Our DSP-PBE tool is able to synthesize a filter $lpf(1989)$, that when applied to the original sound, produces the waveform shown in Figure 5.1c. While the solution is not exact, the difference is not significantly noticeable to an untrained ear.

5.3 Background

To give context for DSP-PBE, we give a definition of the traditional concept of programming by example [121–123]. Programming by example (PBE) is a synthesis technique that automatically generates programs that coincide with given examples. An example is specified as a tuple of input and output values. Given a set $S = \{(i_1, o_1), \ldots, (i_n, o_n)\}$ of input/output examples, the goal is to automatically derive a program P such that for every j, $P(i_j) = o_j$. Recent work in PBE has focused on manipulating fundamental data types such as strings [124, 125] and lists [126, 127].

The core difference between traditional PBE and DSP-PBE is in the application domain of Digital Signal Processing. Digital Signal Processing (DSP) programming languages provide users with an interface to build signal processing programs in domain specific languages. Some of these languages provide their own implementations of signal processing primatives, such as SuperCollider [118], CSound [119], and PureData [120]. Other DSP languages provide alternative front-ends to these languages, such as Vivid [128], which provides Haskell bindings to Supercollider.

Although many DSP languages are full featured enough to write general purpose programs, in this work we focus on the construction of DSP filters. A DSP filter is, broadly speaking, any program that transforms a digital signal from one form to another. An example of a DSP filter is a low-pass

filter, which takes an input signal and generates an output signal that keeps frequencies below some frequency threshold, but removes frequencies above that threshold.

One of the most closely related works in audio signal processing is a technique called resynthesis [129]. Resynthesis is the process of decomposing a sound into its spectrogram, and then building a synthesizer to recreate a similar sound. The limitation here is that resynthesis builds a generative synthesizer, which does not take into account any information about the components used to create the original sound. This limitation means that resynthesis cannot be applied in a new context, whereas DSP-PBE allows us to construct a DSP program that can be used with various new input samples to create novel sounds. For example, DSP-PBE could be given a sample of a trumpet and a trombone, and the generated DSP program could be applied to a violin to hear what a violin sounds like if it was a trumpet that had been turned into a trombone. In this case we can discover the analogy *trumpet:trombone :: violin:?*. Further work has utilized genetic programming approaches citegarcia2001automatic to construct synthesizers, with more recent work utilizing neural networks [130].

From a machine learning perspective, the above example use case is closely related to work on learning analogies [131], where the goal is to discover relations such as *man:king :: woman:queen*. To do this, words are embedded in a vector space, so that the transformation from *man* to *king*, can be directly applied to *woman*. There are two keys differences between this approach and DSP-PBE. The first is that DSP-PBE should produce a human readable transformation. We would like to generate DSP programs that can be used verbatim, but also inspected and modified by the user. While program code provides this readability, vector transformations are not comprehensible in the same way. Second, word embeddings require that the semantics of an object can be embedded into a vector space. As we will see in Sec 5.4, a semantic representation of an audio file (what a human perceives) is not immediately recoverable from its direct representation.

5.4 Aural Distance

As a distance metric, we used as a starting point the literature on acoustic fingerprinting [133]. Acoustic fingerprinting is the concept of creating a condensed, distinct summary of an audio file that can be used later to identify that audio file or to look it up in a database. Acoustic fingerprints

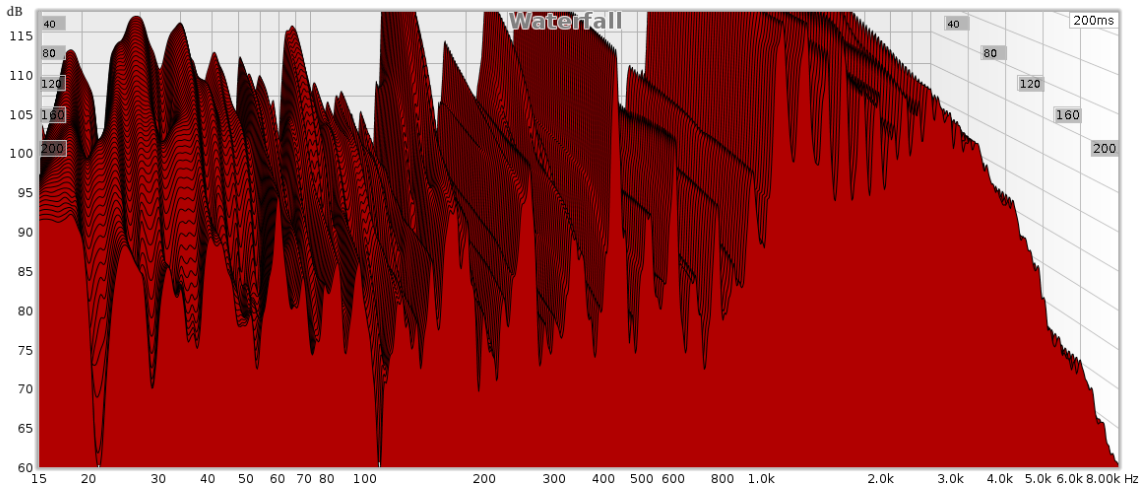


Figure 5.2: A waterfall plot of the `cartoon-spring.wav` from 0-200 ms between frequencies of 20-8000 Hz, where the height indicates the amplitude of each frequency. This plot was created with the REW tool [132].

turn an audio file into a represent of how the file will sound to the human ear regardless of how it is represented in a digital format [133]. There are numerous ways to develop acoustic fingerprints and companies like Shazam and Sound-Hound have developed complex algorithms to create accurate fingerprints even from low quality files recorded on a cellphone mic. For this work, we used the work of Shazam [134] as an inspiration for our distance metric calculation.

As an intuition, the psycho-acoustic identity of a sound file (how humans distinguish between one sound and the next) can be captured by taking every "moment" of audio, and listing the predominate frequencies for that slice of time. This intuition can be represented with a waterfall plot, as shown in Figure 5.2, which plots how the frequencies change over time. A waterfall plot uses the Fast Fourier Transform (FFT) to calculate many discrete Fourier transforms over small times slices. In this way, a waterfall plot is a representation of an audio file as a list of spectrograms plotted over time. In the Shazam method, the peaks are selected from each time slice of audio and used to create a "constellation" of peaks over time. This constellation is then used to build a hash that acts as a fingerprint to uniquely identify the audio sample. We use a similar strategy by first performing a real Fast Fourier Transform on the audio file and then picking out the frequency peaks in each time frame. However, the key difference in DSP-PBE is that we do not use the constellation as a hash for lookup in a database (as Shazam and SoundHound do), but instead, we need a distance metric between two constellations to provide a measure of how close we are to synthesizing the correct

DSP filter. Distance metrics are common in music synthesis tasks, for example, in the generation of jazz improvisations, where the improvisation should stay close by some measure to the original melody [135].

Fast Fourier Transforms (FFT) are the key to a good acoustic fingerprint. The FFT, however, cannot be taken as a blackbox in our application. The two factors we need to consider are 1) the window-size for how many samples will be used to calculate the FFT, and 2) the bin size which roughly speaking, defines the resolution of the FFT.

Each return element is a frequency *bin*, and depending on the scale of your return array the size of these bins varies. In order for each bin to correspond to 1 Hz the size of the return vector must be equal to the sampling frequency (44,100 Hz). If each bin is not 1 Hz, the effects of spectral leakage will be seen. This occurs when the bins do not correspond to the exact frequency peaks of the sound. The amplitude from the peaks that fall in between bins will *leak* over into the closest bin and create a distorted spectrogram. For this reason we had to adjust the size of the FFT return arrays to be 44,100, as 44,100 Hz is a common format for audio. Although this slows down the process of FFT, it provides the most accurate representation of the sound and for our purposes frequency accuracy is paramount.

With our constellations created from the waterfall plot, we constructed a `dist` function that measures the aural distance and is faithful to the psycho-acoustics of the human ear. Our implementation takes the Euclidean distance of the peaks in a time slice on the frequency-amplitude axis. In order to define this distance more formally, we introduce the notation $c@t$ to indicate selecting time slice t from constellation c. We also use the function $peak :: Int \rightarrow Constellation \rightarrow Peak$ to select a peak from a constellation, where the peaks are in sorted order based on frequency. Then, for an audio clip x and an audio clip y, and a function $toC :: Audio \rightarrow Constellation$ to transform the audio clip into a constellation with ts time slices and p peaks in each time slice:

$$\sum_{t=0}^{ts} \sum_{i=0}^{p} euclid\Big(\ peak(i, toC(x)@t), peak(i, toC(y)@t) \ \Big)$$

Note that this definition requires the audio clips to be temporally aligned, which is not always a fair assumption in the real world. We leave the exploration of a temporal offset between two example audio samples to future work.

As a sanity check that this distance metric matches the psycho-acoustic definition of distance, we used the test cases listed in Table 5.1.

Table 5.1: Test cases to evaluate distance metric. The exact values are only important in relationship to the others. The two instruments used were a recording of a Piano (P-) and a Horn (H-).

Test Name & Expected Result	Value 1	Value 2
Identity Value 1 = 0	(P-C, P-C) = 0	NA
Commutativity Value 1 = Value 2	(P-C, P-CSharp) = 5.635	(P-CSharp, P-C) = 5.635
Commutativity Value 1 = Value 2	(P-C, H-CSharp) = 20.500	(H-CSharp, P-C) = 20.500
Filter less than pitch Value 1 $<$ Value 2	(P-C, P-FilterC) = 3.749	(P-C, P-CSharp) = 5.635
Filter less than pitch+instrument Value 1 $<$ Value 2	(P-C, P-FilterC) = 3.749	(P-C, H-CSharp) = 20.500
Pitch less than pitch+instrument Value 1 $<$ Value 2	(P-C, P-CSharp) = 5.635	(P-C, H-CSharp) = 20.500

The goal in the synthesis procedure is to find a DSP filter program, F, such that $\texttt{dist}(O, F(I)) = 0$. However, in practice the DSP-PBE Synthesizer can only get us so close to this metric and we instead just minimize this distance $\texttt{dist}(O, F(I)) < \epsilon$. To do this, the user specifies a default threshold distance for the aural distance. The threshold distance defines how close is acceptably close, and can be changed by the user depending on their needs or requirements.

5.5 Search

As the search space of possible DSP program is extremely large, our search procedures must be exceptionally efficient. As a first foray into DSP-PBE, we restrict ourselves to only synthesizing low-pass and high-pass filters, and global volume adjustment. These two filters have the key property that they are quasi-commutative – when the thresholds of these filters do not overlap, applying a low-pass and then a high-pass is the same as applying a high-pass and then a low-pass. Although our approach has no theoretical basis for being applicable to non-commutative filters (for example, delay lines or ring filters), we do attempt to use our approach on such filters in Sec 5.8. We leave a more thorough exploration of non-commutative filters to future work.

5.5.1 Gradient Descent

Gradient descent is a technique commonly used in modelling and machine learning. Given a cost function, which represents the disagreement between a proposed model and the actual data, gradient descent can be used efficiently to minimize the cost and generate the model of best fit. Gradient descent is only guaranteed to terminate with the globally minimal cost if the cost function being optimized is convex – this is because gradient descent will "descend" along the surface of the cost function, in each step following the steepest gradient. While we were not able to design our aural distance function from Section 5.4 to be convex, our cost function does demonstrate some properties of convexity that allow gradient descent to produce useful results, even if the result is not guaranteed to be the global minimum. We will describe here some properties of our distance metric that were helpful in minimizing the cost of the synthesized filter, as well as the shortcomings of our design, and how we try to overcome them by adjusting our implementation of gradient descent.

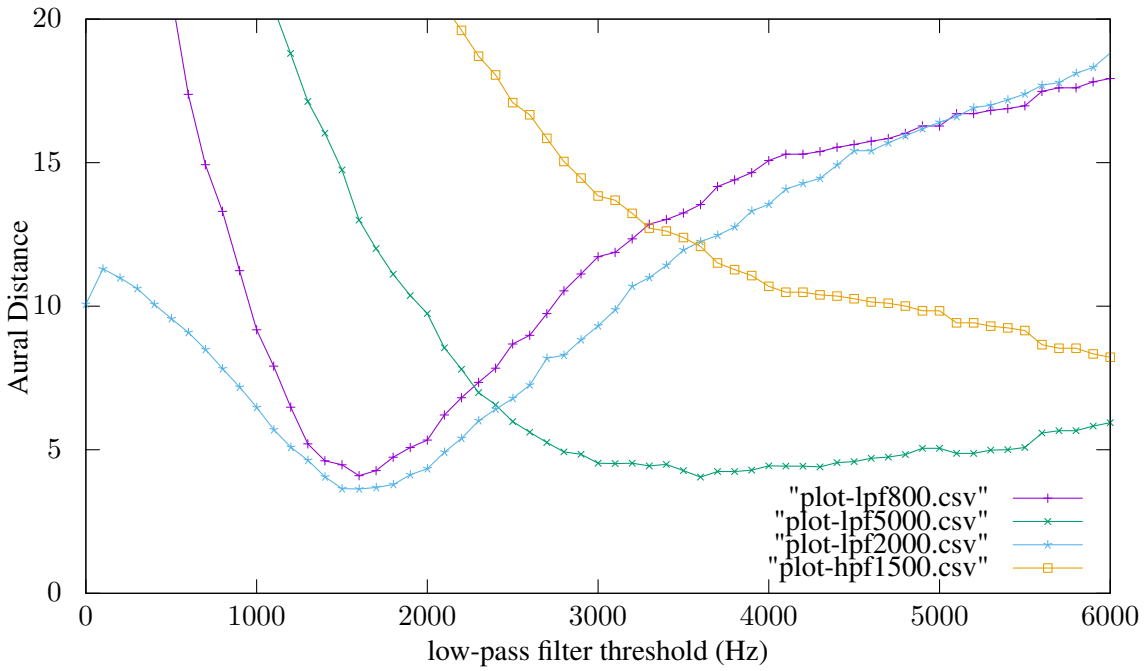


Figure 5.3: The distance curves showing the convex-like shape of the aural distance function. Each curve is the distance between an input file, and a filter applied to that file - $dist(I, \mathcal{F}(I))$.

In order to visualize the rough shape of our distance metric, we plot the distance between pairs of examples, and various possible DSP filters in Figure 5.3. Here we only visualize the distance

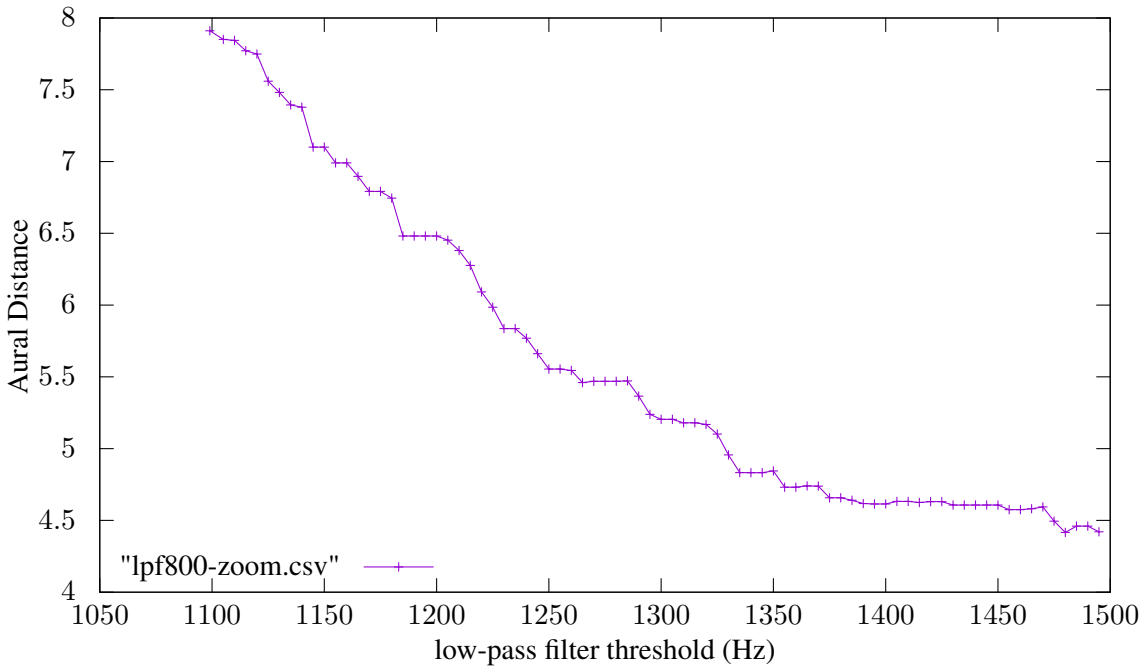


Figure 5.4: Zooming in (1000 to 1500 Hz) on a portion of a curve from Figure 5.3, we see the aural distance function is not perfectly convex on the micro scale.

curves in the dimension of the low-pass filter. Notice that the curves exhibit a clear "saddle", which represents the minimum cost. In the ideal case, gradient descent will find these points. Note that we do not have these graphs available during synthesis – producing the entire graph as in Figure 5.3 is prohibitively expensive.

In Figure 5.3, the last curve we plot is the distance between `cartoon-spring.wav` and `cartoon-spring-hpf1500.wav`, the same file with a high pass filter applied with a threshold of 1500 Hz. Notice that as the threshold of the low-pass filter applied to the input example (`cartoon-spring.wav` increases, the distance to the output example decreases. This is because as a low-pass filter's threshold increases, it allows more and more frequencies to pass into the output – thereby having less of an effect. Whereas in the case of the `cartoon-spring-hpf1500.wav`, the true filter is a high-pass filter, so the less we apply a low-pass filter, the closer we get to the correct filter.

Although Fig. 5.3 depicts on one dimension of the search space (low-pass filter threshold), the actual space we need to search has many more dimensions. In our implementation, we only explore a space of two DSP filters and volume adjustment, but this already results in 5 dimensional space (each

filter requires both a threshold value and an amplitude value for how much of the filter to apply). In general, this space becomes even larger for DSP-PBE as more DSP primitives (ring filter, white noise, delay etc) are added. To speed up gradient descent, we use stochastic gradient descent, so that in each step, we only move in $d < 5$ number of dimensions.

We note that, as we treat the DSP components as black-boxes, our functions are not differentiable, and we cannot use gradient descent in its classic form. Instead we compute gradients by sampling points small values away from our current point to estimate the gradient. This is somewhat similar to a technique called coordinate descent [136]. Recent work has explored the use of differentiable DSP components [130] to allow for more powerful machine learning methods to be used.

5.5.2 Dealing with Non-convexity

There are a number challenges with working with gradient descent in the aural DSP domain because our distance metric is not convex. On the micro scale, the distance function is susceptible to noise and not entirely smooth, as shown in Figure 5.4. In order to handle the micro scale variations, we use a periodic restart of the gradient descent. This means that every n rounds, as defined by the user, the gradient descent will backtrack to the best solution it has found so far. Intuitively, the choice of n represents how far gradient descent is allowed to explore a path of optimization before it is forced to give-up on that direction if it has not found any benefit to this direction. The best value for n then must be determined based on the trade-off of potential time wasted on poor choices, and the potential benefit of these choices. In our implementation we use $n = 4$ after a holistic evaluation of the convexity of the aural distance function. The stochastic gradient descent will then continue, selecting dimensions to explore in each round using a new random seed.

On the macro scale, we face the challenge that the distance function is again not convex – there are many local minima and long plateaus, as shown in Figure 5.5. In order to overcome this, we must carefully pick the initial value for gradient descent. If we pick a value in the middle of a plateau, the gradient descent algorithm will not find any significant gradient, and conclude we have reached the convergence condition. In our current implementation, we iterate at large intervals (1000 Hz) of possible threshold values for both low and high pass filters. We choose possible DSP programs that use only low pass, only high pass, and both low and high pass filters. After evaluating these, we take the lowest cost initial DSP program, and start gradient descent from that point.

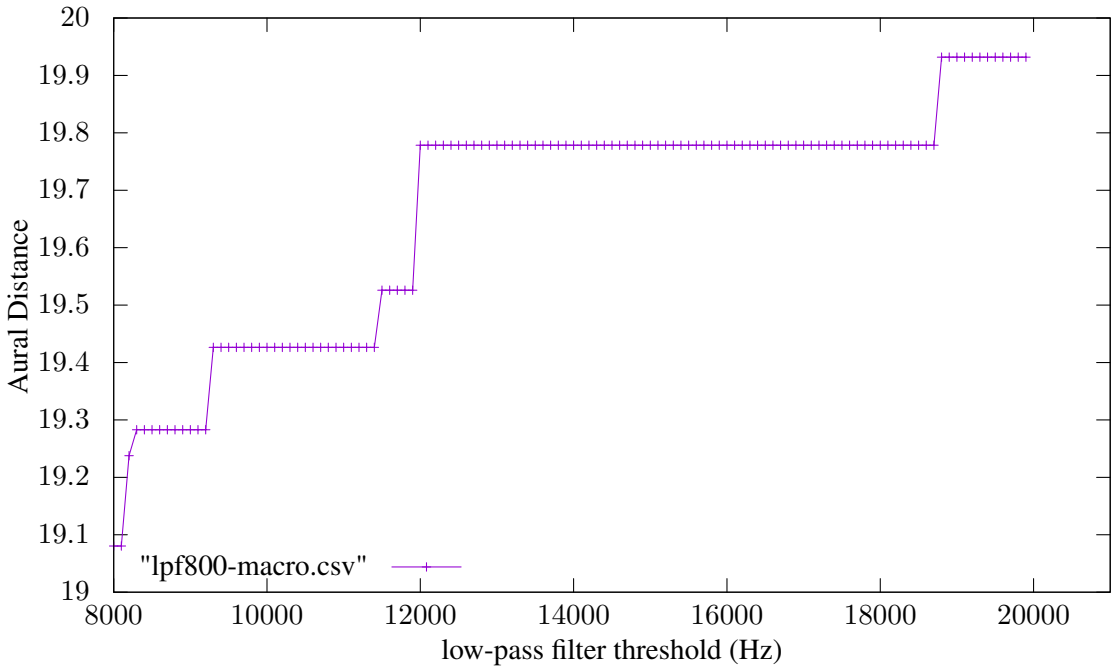


Figure 5.5: Looking at the portion of a curve from Figure 5.3 between 8k Hz and 20k Hz, we see the aural distance function is not perfectly convex on the macro scale. In this case, that is because the sample has very few frequencies above the 8k Hz range.

Finally, one of the key parts of a good application of gradient descent is the choice of the parameters such as the learning rate and the convergence goal. These parameters must be adjusted based on the values observed from the cost (in our case, distance) function. While the details of tuning gradient descent are outside the scope of this paper, it suffices to note that any change in the distance metric will likely also require a readjustment of these parameters.

5.6 Evaluation

Table 5.2: Time to converged on a solution DSP program for various benchmarks. The programs may not match the known DSP program, but may still be psycho-acoustically equivalent depending on the expertise of the listener.

Description	True DSP	Synth'ed DSP	Time (sec)
Cartoon Spring	$lpf(800)$	$lpf(1989)$	56.195
Cartoon Spring	$lpf(5000)$	$lpf(4000) \ggg hpf(7000)$	54.004
Cartoon Spring	$hpf(1500)$	$lpf(1000) \ggg hpf(1000)$	53.964
BTS DNA (Kpop)	$lpf(2000)$	$lpf(1996)$	56.874
Holst Mars	$hpf(3500)$	$lpf(10000) \ggg hpf(1000)$	55.444

We implemented a DSP-PBE tool based on the approach described in Section 5.4 and Section 5.5. Our tool is available open-source at `www.github.com/santolucito/DSP-PBE`[1]. Our tool is mostly written in Haskell and uses the Vivid library [128] for bindings to SuperCollider [118]. Haskell allows easy access to type information and metaprogram construction tools that are useful for program synthesis, however the programs themselves are easily translated back to SuperCollider "synth defs", which are DSP filter programs. We use the scipy python module for calling the FFT since the library is quite mature and provides a simplified interface specifically for calling FFT on audio.

One key implementation point is that we use a separate representation of a DSP for running gradient descent, and for actually processing the audio. Gradient descent works best when all parameters are in the same scale, so we map the frequencies [0,20k] Hz to a [-1,1] scale. Likewise, we map the application levels for each filter (how much of the filtered output should be included in the final mix) on a [-1,1] scale.

In Table 5.2, we show the results of running our tool on a set of benchmarks of input/output example audio samples. The audio samples were transformed in Audacity, using the Low Pass Filter and High Pass Filter effects. Since we use SuperCollider's filter implementations on the backend, there may be very slight variation, but this is to be expected in real-world application as well. All experiments were run on an Intel Core i7-6820HQ CPU @ 2.70GHz with 16 GB of RAM and an Intel Sunrise Point-H HD Audio sound card.

We can also breakdown the runtime cost of synthesis into the two different stages - 1) initial program selection, and 2) gradient descent. The initial program selection phase is a mostly fixed cost, as we always evaluate the same distribution of initial value. On average this process takes roughly 40 seconds. We outline future directions of research that may be able to reduce this cost in Section 5.7.

1. The exact version of the code used for this evaluation is available at commit `https://github.com/santolucito/DSP-PBE/tree/d022954164b830395bddb21cdc94046ed6882083`.

5.7 Refinement Type Driven Synthesis

In order to find an initial value for gradient descent, we could use refinement types [137]. In this section we explore a possible optimization for selecting an initial DSP program for gradient descent. This has not yet been implemented, but we present the theory behind the approach.

5.7.1 Refinement Types for DSP

Refinement types are a way of giving an abstract description of the behavior of a function. For example, using a similar syntax to the refinement type system for Haskell, LiquidHaskell [138], given the function `map ::[a] → [b]` we can further provide a refinement types that captures some properties of the behavior of this function over values:

$$\texttt{f :: xs:[a]} \rightarrow \texttt{ys:[b]} \mid \texttt{length xs == length ys}$$

In this case, the refinement type describes that the length of the lists are still equal after applying the `map` function.

In a similar style for DSP, we can write predicates about the filters available to us during synthesis. For example, a low-pass filter could be described as the refinement type that says the amplitude of the frequencies greater than the threshold frequency have decreased in the output Audio. For brevity in notation, we will only treat a single time slice from the waterfall plot here, but the concept generalizes when quantified over all time slices as well.

$$\begin{aligned}&\texttt{lpf :: t:Float} \rightarrow \texttt{xs:Audio} \rightarrow \texttt{ys:Audio} \mid \\ &\forall f_1 \in \texttt{spectrogram(xs)}.\, \forall f_2 \in \texttt{spectrogram(ys)}. \\ &(f_1 > \texttt{t} \wedge f_2 > \texttt{t} \wedge f_1 == f_2) \Rightarrow \texttt{amp}(f_1) > \texttt{amp}(f_2)\end{aligned}$$

Where `t` represents the level at which the lowpass filter is applied, `spectrogram` represents the spectrogram of the sound sample, f_i represents a frequency, and `amp()` represents the amplitude of the frequency.

Additionally, a high-pass filter could be described as the refinement type that says the amplitude of the frequencies less than the threshold frequency have decreased in the output Audio.

$$\texttt{hpf :: t:Float} \rightarrow \texttt{xs:Audio} \rightarrow \texttt{ys:Audio} \mid$$
$$(\forall f_1 \in \texttt{spectrogram(xs)}.\forall f_2 \in \texttt{spectrogram(ys)}).$$
$$(f_1 < \texttt{t} \wedge f_2 < \texttt{t} \wedge f_1 == f_2) \Rightarrow \texttt{amp}(f_1) > \texttt{amp}(f_2)$$

Notice that in these refinement types, we only need to calculate the spectrogram for the input and output statically. As opposed to the current technique of generating filters, applying them, and the calculating the aural distance, this approach is relatively static. We could quickly check many threshold values over the input and output examples. This will only yield a rough boolean estimation of whether this threshold should even be considered, but this is enough information for us to select an initial program to pass to our gradient descent algorithm. As the search for an initial filter takes roughly 40 seconds out of our current benchmarks, this could dramatical increase the speed of synthesis.

5.7.2 Combination of Search Algorithms

Beyond just using the refinement types to select an initial program for gradient descent, we can use refinement types in as part of the main search strategy as well. We briefly describe here a way to use refinement types in combination with gradient descent to handle more complex combinations of DSP filters. So far in our work (*c.f.* Sec. 8.5) we have synthesized filters with a fixed form - all our solutions use a single low-pass filter, and a single high-pass filter. Ideally, we would be able to synthesize solutions that use any arbitrary combination of filters. In order to do this, we would need an iterative solution that can find one filter at a time.

In this approach, given input example `x:Audio` and output example `y:Audio`, we would first find a filter $\mathcal{F}$' using the approach described in Sec. 5.5 and Sec. 5.7.1. We will say that this $\mathcal{F}$' has the refinement type r_1. However, this filter might not return a satisfactory result. We could then continue the search using the output of $\mathcal{F}$' `(x)` as the new input example, `z:Audio`. Now the synthesis task is to find a filter $\mathcal{F}$" (with refinement type r_2) using input example `z:Audio` and

output example `y:Audio`. Essentially, $\mathcal{F}$' has gotten us the first half of the way, and $\mathcal{F}$" will get us the second half of the way. With this, we can start to use more information rich refinement types, such as below:

$$\mathcal{F} \;::\; \texttt{x:Audio} \rightarrow \texttt{y:Audio} \mid \exists\, \texttt{z:Audio}.\, r_1\texttt{(x,z)} \wedge r_2\texttt{(z,y)}$$

5.8 Future Work and Conclusions

The main contribution of this paper is to pose the problem of DSP-PBE. While we have presented a prototype implementation of a DSP-PBE tool, this primarily functions as a proof-of-concept. There remains significant room for optimization in both the distance calculation and the search algorithm. In future work, we also plan to expand beyond commutative filters to be able to synthesize effects such as delay lines.

We briefly revisit the motivating conceptual example from Sec. 5.3 where we want to synthesize the filter that transforms a trumpet into a trombone and apply that filter to a violin. In order to achieve this we need more complex filters than high-pass, low-pass, and amplitude adjustment. Despite a lack of a formal approach for non-commutative filters, we added a pitch shift filter and the `ringz` delay line filter from SuperCollider into our tool using the current approach. Though we had no reason to believe that our current approach should produce useful results when allowing more complex filters in the search space of synthesis, we found the results to be at least interesting, and even reasonable. The synthesized filter to transform a trumpet into a trombone was as follows:

```
SetVolume: 1.00%
LoPass: freq@1000 amp@0.91 >>>
HiPass: freq@100 amp@0.00 >>>
PitchShift: freq@-1600 amp@0.91 >>>
Ringz: freq@9100 delay@0.10 amp@0.23 >>>
WhiteNoise: amp@0.00
```

The trumpet and trombone input audio files, as well as the audio of a violin with this filter applied is available as a Soundcloud playlist [139].

Although with the current tool, synthesis times presented might be prohibitively slow for many use cases, especially on such small programs, we should be encouraged by progress in other domains

of program synthesis. In the SyGuS program synthesis competition, which has run for four years, tools have seen an exponential speed up and increase in the range of programs that can be synthesized. As one example, in the 2014 competition the `LinExpr_eq1.sl` benchmark was only solved by one tool, and took 1128 seconds [140]. In the 2017 competition, the same benchmark was solved by all tools, with the fastest taking only 199 seconds [141].

Chapter 6

Grammar Filtering for Syntax Guided Synthesis

Work presented in this chapter was completed in collaboration with Kairo Morton, Elven Shum, William Hallahan, and Ruzica Piskac. Sections of this work have been previously published [16], and are reproduced here, at times in their original form.

6.1 Motivation

The idea of automated code synthesis is an area of research with a long history (cf. the Church synthesis problem [142]). However, due to the problem's undecidability and high computational complexity for decidable fragments, for almost fifty years the research in program synthesis was mainly focused on addressing theoretical questions and the size of synthesized programs was relatively small. However, the state of affairs has drastically changed in the last decade. By leveraging advances in automated reasoning and formal methods, there has been a renewed interest in software synthesis. The research in program synthesis has recently focused on developing efficient algorithms and tools, and synthesis has even been used in industrial software [143]. Today, machine learning plays a vital role in modern software synthesis and there are numerous tools and startups that rely on machine learning and big data to automatically generate code [144, 145].

With numerous synthesis tools and formats being developed, it was difficult to empirically evaluate and compare existing synthesis tools. The Syntax Guided Synthesis (SyGuS) format

language [146, 147] was introduced in an effort to standardize the specification format of program synthesis, including PBE synthesis problems. The SyGuS language specifies synthesis problems through two components - a set of constraints (eg input-output examples), and a grammar (a set of functions). The goal of a SyGuS synthesis problem is to construct a program from functions within the given grammar that satisfies the given constraints. With this standardized synthesis format and an ever expanding set of benchmarks, there is now a yearly competition of synthesis tools [9], which pushes the frontier of scalable synthesis further.

The SyGuS Competition splits synthesis problems into tracks, for example PBE Strings or PBE BitVectors, assigning a different grammar for each track - and sometimes even varying the grammar within a single track. As the grammar defines the search space in SyGuS, this allows benchmark designers to ensure problems are relatively in-scope of current tools. However, when synthesis is deployed in real-world applications, we must allow for larger grammars that account for the wide range of use-cases users require [148]. While larger grammars allow for more expressive power in the synthesis engine, it also slows down the whole synthesis process.

In our own experimentation, we found that by manually removing some parts of the grammar from the SyGuS Competition benchmarks, we can significantly improve synthesis times. Accordingly, we sought to automate this process. Removing parts of a grammar is potentially dangerous though, as we may remove the possibility of finding a solution altogether. In fact, understanding the grammar's impact on synthesis algorithms is a complex problem, connected to the concept of overfitting [149].

In this chapter, we utilize machine learning to automate an analysis of a SyGuS grammar and a set of synthesis constraints. We generate a large number of SyGuS problems, and use this data to train a neural network. Given a new SyGuS problem, the neural network predicts how likely it is for a given grammar element to be critical to synthesizing a solution to that problem. Our key insight is that, in addition to criticality, we predict how much time we expect to save by removing this grammar element. We combine these predictions to efficiently filter grammars to fit a specific synthesis problem, in order to speed up synthesis times. Even with these reduced grammars, we are still able to find solutions to the problems.

We implemented our approach in a modular tool, GRT, that can be attached to any existing SyGuS synthesis engine as a blackbox. We evaluated GRT by running it on the SyGuS Competition Benchmarks from 2019 in the PBE Strings track. We found GRT outperformed CVC4, the winner of

the SyGuS Competition from 2019, reducing the overall synthesis time by 47.65%. Additionally, GRT was able to solve a benchmark for which CVC4 timed out.

In summary, the core contributions of our work are as follows:

1. A methodology to generate models that can reduce time needed to synthesize PBE SyGuS problems. In particular, our technique reduced the grammar by identifying which functions to try to eliminate to increase the efficiency of a SyGuS solver. It also learns a model to predict which functions are critical for a particular PBE problem.

2. A demonstration of the effectiveness of our methodology. We show experiments on existing SyGuS PBE Strings track that demonstrates the speed up resulting from using our filtering as a preprocessor for an existing SyGuS solver. Over the set of benchmarks, our techniques decreases the total time taken by synthesis by 47.65%.

6.2 Background

A SyGuS synthesis problem is a tuple (C, G) of constraints, C, and a context-free grammar, G. In our case we restrict the set of constraints to the domain of PBE, so that all constraints are in the form of pairs (i, o) of input-output examples. We write $G \setminus g$ to denote the grammar G, but without the terminal symbol g. The set of terminal symbols are the component functions that can be used in constructing a program (e.g. +, -, str.length). We also use the notation, $\pi(G)$, to denote the projection of G into its set representation, which is the set of the terminal symbols in the grammar.

The problem statement of syntax-guided synthesis (SyGuS) is; given a grammar, G, and a set of constraints C, find a program, $P \in G$, such that the program satisfies all the constraints – $\forall c \in C. P \vdash c$. For brevity, we equivalently write $P \vdash C$. If our synthesis engine is able to find such a program in t seconds or less, we write that $(G, C) \rightsquigarrow_t P$. We use the notation T_G^C to indicate the time to run $(G, C) \rightsquigarrow_t P$. If the SyGuS solver is not able to find a solution within the timeout ($T_G^C > t$), we denote this as $(G, C) \not\rightsquigarrow_t P$. We typically set a timeout on all synthesis problems of 3600 seconds, the same value of the timeout used in the SyGuS competition. We write $(G, C) \rightsquigarrow P$ and $(G, C) \not\rightsquigarrow P$ as shorthand for $(G, C) \rightsquigarrow_{3600} P$ and $(G, C) \not\rightsquigarrow_{3600} P$, respectively.

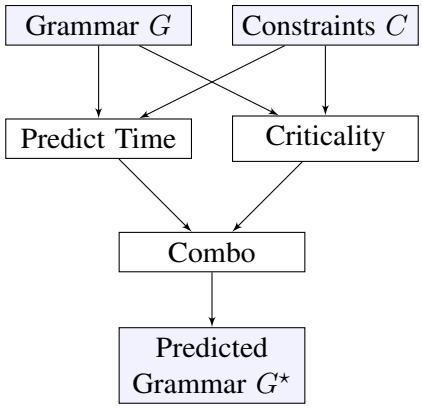


Figure 6.1: GRT uses the grammar G and constraints C to predict how critical each function is, and the amount of time that would be saved by eliminating it from the grammar. Then, it outputs a new grammar $G^\star$, which it expects will speed up synthesis over the original grammar (that is, it expects that $T^C_{G^\star} < T^C_G$).

We define G as the grammar constructed from the maximal set of terminal symbols we consider for synthesis. We call a terminal, g, within a grammar, *critical* for a set of constraints, C, if $(G \setminus g, C) \not\rightsquigarrow P$. For any given set of constraints, if a solution exists with G, there is also a grammar, G_{crit}, that contains exactly the critical terminal symbols required to find a solution. More formally, G_{crit} is constructed such that

$$(G_{crit}, C) \rightsquigarrow P \wedge \forall g \in G_{crit}.\, (G \setminus g, C) \not\rightsquigarrow P$$

Note that G_{crit} is not unique.

The goal of our work is to find a grammar, $G^\star$, where $\pi(G_{crit}) \subseteq \pi(G^\star) \subseteq \pi(G)$. This will yield a grammar that removes some noncritical terminal symbols so that the search space is smaller, but still sufficient to construct a correct program.

6.3 Overview

Our system, GRT, works as a preprocessing step for a SyGuS solver. The goal of GRT is to remove elements from the grammar and thus, by having a smaller search space, save time during synthesis. To do this we combine two metrics, as shown in Figure 10.1: our predicted confidence that a grammar element is not needed, and our prediction of how much time will be saved by removing that element. We focus on removing only elements where we are both confident that the grammar element is

noncritical, and that removing the grammar element significantly impacts synthesis times. By giving the constraints and the grammar definition to GRT, we predict which elements of the grammar can be safely removed. By analyzing running times we predict which of these elements are benefical to remove. We describe GRT in three sections, addressing dataset generation, the training stage, and our evaluation.

6.4 Data Generation

In order to learn a model for GRT, we need to generate a labelled dataset that maps constraints to grammar components in G_{crit}. This will allow us to predict, given a new set of constraints C', which grammar elements are noncritical for synthesis, and accordingly prune our grammar. The generation of data for application to machine learning for program synthesis is a nontrivial problem, requiring careful construction of the dataset [150]. We break the generation of this dataset into two stages: first, we generate a set of programs, $\mathcal{P}$ from G. Then, for each program in $\mathcal{P}$, we generate constraints for that program. We additionally need a dataset of synthesis times, in order to predict how long synthesis takes for a given set of constraints.

6.4.1 Criticality Data

To generate a set of programs $\mathcal{P}$, that can be generated from a grammar G, we construct a synthesis query with no constraints. We then run CVC4 with the command `-sygus-stream`, which instructs CVC4 to output as many solutions as it can find. With no constraints, all functions satisfy the specification, and CVC4 will generate all permutations of (well-formed and well-typed) functions in the grammar, until the process is terminated (we terminate after generating n programs). Because CVC4 generates solutions of increasing size, we collect all generated programs, then shuffle the order to prevent data bias with respect to the order (size) in which CVC4 generated programs.

After generating programs, we generate corresponding constraints (in the form of input-output examples for PBE) for these functions. To do this, for each program, P, we randomly generate a set of inputs I, and compute the input-output pairs $C = \{(i, P(i)) \mid i \in I\}$. We then form a SyGuS problem (G, C), where we know that the program P satisfies the constraints, and is part of the grammar: $P \vdash C$ and $P \in G$. This amounts to programs that *could* be synthesized from

the constraints (i.e. $(G, C) \rightsquigarrow_{\infty} P$). It is important that our dataset represent programs that *could* be synthesized, as opposed to what *can* be synthesized (i.e. $(G, C) \rightsquigarrow_{3600} P$). This is important because we will use this data set to try to learn the "semantics" of constraints, and we do not want to use this data set to additionally, inadvertently learn the limitations of the synthesis engine.

At this point, we have now constructed a dataset of triples of grammars (fixed for all benchmarks), constraints, and programs, $D = \{(G, C_1, P_1) \ldots (G, C_n, P_n)\}$. In order to use D to helps us predict G_{crit}, we break up each triple by splitting each constraint set C into its individual constraints. For a triple (G, C, P), where $C = \{c_1 \ldots c_m\}$, we generate a new set of triples $\{(G, c_1, P) \ldots (G, c_m, P)\}$. The union of all these triples of individual constraints form our training set, $\mathcal{TR}_{crit}$, that will be used to predict critical functions in the grammar for a given set of constraints.

6.4.2 Timing Data

In addition to a training set for predicting G_{crit}, we also need a separate training set for predicting the time that can be saved by removing a terminal from the grammar. This dataset maps grammar elements $g \in G$ to the effect on synthesis times, $\mathbb{R}$, when g is dropped from the grammar. To do this we require synthesis problems that more closely model the types of constraints that humans typically write. We collect these set of benchmarks from users of the live coding interface for SyGuS [148]. Because we had limited number of human-generated constraint examples, we augmented this with constraints generated from $\mathcal{TR}_{crit}$.

We run synthesis for each problem with the full grammar, as well as with all grammars constructed by removing one element, g. For every synthesis problem benchmark, $1 \leq i \leq m$, we record the difference in synthesis times between running with the full grammar, and removing g:

$$T_G^{C_i} - T_{G \setminus g}^{C_i}$$

Thus, we create a training set, $\mathcal{TR}_{time}$, relating each terminal $g \in \pi(G)$ and a set of constraints, to the time it takes to synthesize a solution without that terminal.

6.5 Training

6.5.1 Predicting criticality

Our goal is to predict, given a set of constraints C, if a terminal g belongs to the set of terminals $\pi(G_{crit})$ for C. To do this, we use a Feedforward Neural Network (Multi-Layer Perceptron), with an extra embedding layer to encode the string valued input-output examples into feature vectors. We train the neural network to predict the membership of each terminal $g \in \pi(G)$ to the critical set $\pi(G_{crit})$, based on a single constraint $c \in C$. This prediction produces a 1D binary vector of length $|\pi(G)|$, where 1 at position i in the binary vector indicates the terminal in position i is predicted to belong to the critical set.

When a SyGuS problem has multiple ($|C| \geq 2$) constraints, we run our prediction on each constraint individually. We then use a voting mechanism to come to consensus on the construction of $G^{\star}$. After computing $|C|$ binary vectors across all constraints, the vectors are summed to produce a final voting vector. The magnitude of each element in this final voting vector represents the number of votes "from each constraint" that the terminal represented by that element is in the critical set. We then use this final voting vector in combination with our time predictions.

6.5.2 Predicting time savings

It is only worthwhile to remove a terminal symbol g from a grammar G if $T^{C}_{G\setminus g}$ is less than T^{C}_{G}. If a g stands to only give us a small gain in synthesis times, it may not be worth the risk that we incorrectly predicted its criticality.

To predict the amount of time saved by removing a terminal g we examine the distribution of times in our training set $\mathcal{TR}_{time}$. For each terminal g, we calculate A_g, the average time increase that results from removing g from the grammar. Denoting the time to run $(G, C) \rightsquigarrow P$ as T^{C}_{G}, we can write A_g as:

$$A_g = \frac{\sum_{i=1}^{n} T^{C_i}_{G} - T^{C_i}_{G\setminus g}}{n}$$

If a terminal g has a negative A_g, then removing it from the grammar actually slows down synthesis, on average. As such, dropping the terminal from the grammar is not generally helpful. Thus, we only consider those terminals with a positive A_g in our second step.

6.5.3 Combining predictions

With our predictions of the criticality a terminal g and of time saved by removing g, we must make a final decision on whether or not we should remove g. To do this, we take the top three terminals with the greatest average positive impact on synthesis time over the training set, as computed with A_g. These tended to be terminals that mapped between types which saved more time due to the internal mechanisms and heuristics of the CVC4 solver. We then use the final voting vector from our criticality prediction to choose only two out of the three to remove from G to form $G^\star$. We chose to remove only two terminals from G in order to minimize the likelihood of generating a $G^\star$, such that $\pi(G^\star) \subseteq \pi(G_{crit})$. We conjecture that the number of terminals removed is a grammar-dependent parameter that must be selected on a per grammar basis, just as the number of terminals with $A_g > 0$ is grammar specific.

6.5.4 Falling back to the full grammar

There is some danger that $G^\star$ will, in fact, not be sufficient to synthesize a program. Thus, we propose a strategy that

- first, tries to synthesize a program with the grammar $G^\star$
- second, if synthesis with $G^\star$ is unsuccessful, falls back to attempting synthesis with the full grammar G.

We determine how long to wait before switching from $G^\star$ to G by finding an x that minimizes:

$$\sum_{i=1}^{n} \begin{Bmatrix} T_{G^\star}^{C_i} & T_{G^\star}^{C_i} < x \\ \min(x + T_G^{C_i}, t) & T_{G^\star}^{C_i} > x \end{Bmatrix}$$

where $C_1 \ldots C_n$ are the constraints from the training set, and t is the timeout for synthesis.

Ideally, as captured in the first line of the sum, $(C_i, G^\star) \rightsquigarrow_x P$ will finish before $T_{G^\star}^{C_i}$ = x. However, if a benchmark does not finish in that time, it will fall back on the full grammar. Then, either $(C_i, G^\star) \rightsquigarrow_{t-x} P$ will succeed, and synthesize the expression in total time $x + T_G^{C_i}$, or synthesis will timeout, in total time $(t - x) + x = t$.

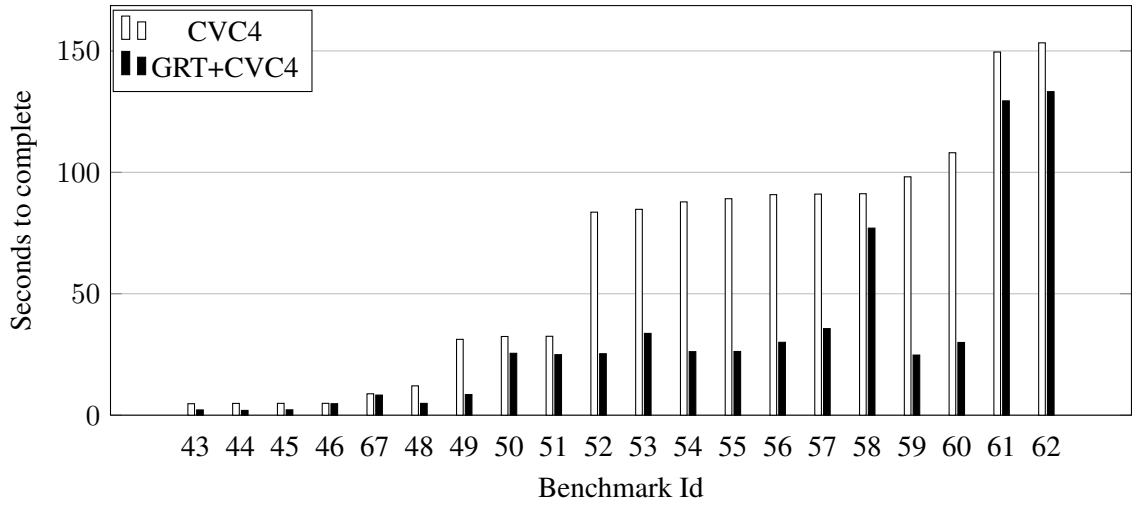


Figure 6.2: The top 20 problems with longest synthesis time for CVC4 (excepting timeouts), and the corresponding synthesis times for GRT+CVC4.

6.6 Experiments

The SyGuS competition [151] provides public competition benchmarks and results from previous years. In particular, the PBE Strings dataset provides a collection of PBE problems over a grammar that includes string, integer, and Boolean manipulating functions. First, we describe our approach to generating a training set of PBE problems over strings. Then, we present our results running GRT against the 2019 competition's winner in the PBE Strings track, CVC4 [9, 10, 17]. We are able to reduce synthesis time by 47.65% and synthesize a new solution to a benchmark that was left unsolved by CVC4.

6.6.1 Technical details

The data triples generated during our initial data generation process of $\mathcal{TR}_{crit}$ are triples of strings. However, the neural network cannot process input-output pairs of type string as input. Thus, this data must be encoded numerically before it can be utilized to train the neural network. Each character in the input-output pairs is converted to its ASCII equivalent integer value. The size of each pair is then standardized by adding a padding of zeros to the end of each newly encoded input and output vector respectively. This creates two vectors: the encoded input and the encoded output, both of which have a length of 20. These two vectors are then concatenated to give us a single vector for training. By the

end of this process the triples created in our first data generation step are now one vector of type $\mathbb{N}^{40}$ representing the input-output pair and a correct label P that will be predicted.

To generate the training set for predicting synthesis times, $\mathcal{TR}_{time}$, we combine human generated and automatically generated SyGuS problems. Specifically, we use 10 human generated SyGuS problems, and 20 randomly selected problems from $\mathcal{TR}_{crit}$.

The overall architecture of our model can be categorized as a multi-layer perceptron (MLP) neural network. More specifically, our model is made up of five fully connected layers: the input layer, three hidden layers, and the output layer. By using the Keras Framework, we include an embedding layer along with our input layer which enables us to create unique vector embeddings of length 100 for any given input-output pair in the dataset. This embedding layer learns the optimal weights used to create these unique vectors through the training process. Thus, we create an encoding of the input-output pairs for training, while simultaneously standardizing the scale of the vector before it reaches the first hidden layer. The hidden layers of the model are all fully connected, and all use the sigmoid activation function. In addition, we implement dropout during training to ensure that overfitting does not occur. The size of the hidden layers was calculated using a geometric series to ensure that there was a consistent decrease in layer size as the layers get closer to the output layer. Specifically, the size of each hidden layer was calculated by:

$$HL_{size}(n) = input_{size}(\frac{output_{size}}{input_{size}})^{\frac{n}{L_{num}+1}}$$

where L_{num} represents the total number of layers in the network. Our model used the Adam optimization method and the binary-cross entropy loss function as it is well suited for multi-label classification. Overall, our model was trained on 124928 data points for 15 epochs with a batch size of 200 producing a training time of 228 seconds.

6.6.2 Results

After generating our data sets and training our model, we wrote a wrapper script to run GRT as a pre-processor for CVC4's SyGuS engine. We compared the synthesis results of GRT+CVC4 with the synthesis results of running CVC4 alone. All experiments were run on MacBook Pro with a 2.9 GhZ Intel i5 processor with 8GB of RAM.

CVC4 uses a default random seed, and is deterministic over the choice of that seed, so the results of synthesis from CVC4 on a given grammar and set of constraints are deterministic. We note that our training data in no way used any of the SyGuS benchmarks.

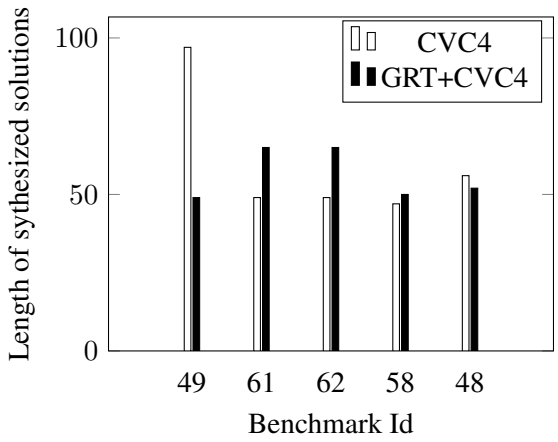


Figure 6.3: When the GRT+CVC4 found a different solution than CVC4, it was on average shorter than the solution found with the full grammar.

GRT+CVC4 outperformed directly calling CVC4 on 32 out of 64 benchmarks (50%), with a reduction in total synthesis time over all benchmarks from 1304.87 seconds with CVC4 to 683.09 seconds with GRT+CVC4. On one benchmark, CVC4 timed out and was not able to find a solution (even when the timeout was increased to 5000 seconds), while GRT+CVC4 found a solution within the timeout specified by the SyGuS Competition rules (3600 seconds). On one benchmark, both CVC4 and GRT+CVC4 timeout (TO) and are not able to find a solution. On the other 31 benchmarks, CVC4 performed the same (within ± 0.1s) with and without the preprocessor. All the benchmarks for which CVC4 performed the same as GRT+CVC4 finish in under 2 seconds, and 28 of the 31 finish in under a second. In these cases there was little room for improvement even with GRT+CVC4.

Figure 6.4 shows the exact running times with both the full and reduced grammars from the benchmarks with the 30 largest running times with the full grammar. These are the benchmarks for which the synthesis times and size of the solution diverge most meaningfully, however all other data is available in the supplementary material for this paper. Figure 6.4 also shows $|P|$ and $|P^*|$, the sizes of the programs found by the CVC4 and GRT+CVC4, respectively. We define size of a program as the number of nodes in the abstract syntax tree of the program. In terms of the grammar G, this is the number of terminals (including duplicates) that were composed to create the program.

In Figure 6.2, we present a visual comparison of the results for the 20 functions that took CVC4 the longest, while still finishing in the 3,600 second time limit. We note that we have the largest gains on the problems for which CVC4 is the slowest. Problems that CVC4 already handles quickly stand to benefit less from our approach.

In order to get a better baseline to understand the impact of GRT on running times, we ran a version of GRT with only the criticality prediction, which we call GRTC. In this case, GRTC+CVC4 actually performed worse than CVC4 by itself, increasing the running time on 53 out of the 62 benchmarks that did not timeout on CVC4.

On all but 5 benchmarks, CVC4 synthesized the same program when running with G and $G^\star$. The sizes of the programs (in terms of the number of terminal symbols used) for the benchmarks on which CVC4 synthesized different programs are shown in Figure 6.3. While on some benchmarks GRT+CVC4 produced a larger solution than CVC4, as a whole the sum of the size of all solutions for CVC4 was 806, while for GRT+CVC4 it was 789. Thus, overall, we were able to outperform CVC4 on size of synthesis as well.

The SyGuS competition scores each tool using the formula: $5N + 3F + S$, where N is the number of benchmarks solved (non-timeouts), F is based on a "pseudo-logarithmic scale" [151] indicating speed of synthesis, and S is based on a "pseudo-logarithmic scale" indicating size of the synthesized solution. On all three of these measurements, GRT+CVC4 performed better than CVC4. There are number of other synthesis tracks available in the SyGuS competition, which do not involve PBE constraints. We note that our approach can selectively be applied as a preprocessing step for input in the PBE track without incurring an overhead on other synthesis tasks.

Although we implemented a strategy to manage a switch from the reduced grammar back to the full grammar, we found in practice that the optimal strategy for our system was to exclusively use the reduced grammar. Because we had conservatively pruned the grammar, we had no need to switch back to the full grammar.

6.7 Related

One approach to SyGuS is to directly train a neural network to satisfy the input/output examples [152–157]. However, such approaches struggle to generalize, especially when the number of examples is small [158]. Some existing work [159, 160] aims to represent aspects of the syntax and semantics of a language in a neural network. In contrast to these existing approaches, which aim to outright solve SyGuS problems, our work acts as a preprocessor for a separate SyGuS solver. However, one could also explore using our work as a preprocessor for one of these existing neural network directed

synthesis approaches. Other works have explored combining logic-directed and machine learning guided synthesis approaches [11]. This work sought to split synthesis tasks between generating high level sketches with neural networks, and fill in the holes of the sketch with an enumerative solver. Our work could be complementary to this, by assisting in pruning of the search space needed to fill in the holes.

Like our work, DeepCoder [145] and Neural-Guided Deductive Search (NGDS) [161] identify pieces of a grammar that should be removed from the grammar. However, in our parlance, these works only consider *criticality*, which measures how important a part of the grammar is to completing synthesis. Unlike our work, they do not consider the time savings from removing or keeping a part of the grammar. NGDS [161] does note that different models could be trained for different pieces of a grammar, however, it provides no means of automating this process. Rather, the user would have to manually elect to train individual neural networks for different grammatical elements. Work by Si et al [162] aims to learn an efficient solver for a SyGuS from scratch, rather than, as in our work, acting as a preprocessor for a separate solver.

6.8 Discussion

In a way, by training on a dataset we generate from the output of the interpreter of the language, we are encoding an approximation of the semantics into our neural network. While the semantic approximation is too coarse to drive synthesis itself, we can use it to prune the search space of potential programs. By predicting terminals impact on synthesis time, we more conservatively remove only terminals likely to have a positive impact. In conjunction with analytically driven tools, we can then significantly improve synthesis times with very little overhead.

While we have presented GRT, which demonstrates a significant gain in performance over all existing SyGuS solvers, we still have many opportunities for further improvement. In our prediction of the potential time saved by removing a terminal from the grammar, we have simply used the average expected value over all samples in the dataset. By using a neural network here, we may be able to leverage some property of the SyGuS problem constraints to have more accurate potential time savings predictions. This would allow us, possibly in combination with a more advance prediction combination strategy, to more aggressively prune the grammar. The drawback to this approach is

that we may then potentially remove too much from the grammar. One of the key features of GRT is that it introduces no new timeouts, that is, it does not remove any critical parts of the grammar.

Additionally, our prediction of criticality of a terminal uses a voting mechanism to combine the prediction based on each constraint. While this worked well in practice, this strategy ignores the potential for interaction between constraints. In our preliminary exploration, we were not able to construct a model that captures this inter-constraint interaction in a useful way. This may be a path for future work. In a similar vein, there exist a number of other works that define a criticality measure for each terminal in the SyGuS grammar [145, 161]. It may be possible to leverage these in place of our criticality measure, and in combination with our time savings prediction, to achieve better results.

So far we have only explored the PBE Strings track of the SyGuS Competition. The competition also features a PBE BitVectors track where our technique may have significant gains as well. This would require a new encoding scheme, but the overall approach would remain similar. In general, extending this work to allow for other PBE types, as well as more general constraints, would broaden the potential real-world application of SyGuS.

id	benchmark file	T_G^C	$T_{G^\star}^C$	$\|P\|$	$\|P^\star\|$
34	lastname-small.sl	1.80	1.84	4	4
35	bikes-long.sl	1.97	1.76	3	3
36	bikes-long-repeat.sl	2.08	1.71	3	3
37	lastname.sl	2.31	1.83	4	4
38	phone-6-short.sl	3.23	1.22	11	11
39	phone-7-short.sl	3.26	1.26	11	11
40	initials-long-repeat.sl	3.33	2.54	7	7
41	phone-5-short.sl	3.72	1.51	9	9
42	phone-7.sl	4.57	2.03	11	11
43	phone-8.sl	4.72	2.17	11	11
44	phone-6.sl	4.85	1.97	11	11
45	phone-5.sl	4.88	2.20	11	11
46	phone-9-short.sl	4.88	4.73	52	52
47	phone-10-short.sl	8.81	8.28	49	49
48	phone-9.sl	12.08	4.86	56	52
49	phone-10.sl	31.23	8.49	97	49
50	lastname-long.sl	32.40	25.49	4	4
51	lastname-long-repeat.sl	32.49	24.92	4	4
52	phone-6-long-repeat.sl	83.59	25.31	11	11
53	phone-5-long-repeat.sl	84.77	33.68	11	11
54	phone-7-long.sl	87.83	26.15	11	11
55	phone-7-long-repeat.sl	89.13	26.23	11	11
56	phone-5-long.sl	90.81	30.01	11	11
57	phone-8-long-repeat.sl	91.04	35.64	11	11
58	phone-9-long-repeat.sl	91.19	77.02	47	50
59	phone-6-long.sl	98.15	24.75	11	11
60	phone-8-long.sl	108.06	29.94	11	11
61	phone-10-long-repeat.sl	149.53	129.43	49	65
62	phone-10-long.sl	153.32	133.22	49	65
63	initials-long.sl	TO	TO	-	-
64	phone-9-long.sl	TO	3516.21	-	49

Figure 6.4: Synthesis results over the 30 longest running benchmarks from SyGuS Competition's PBE Strings track.

Part III ❖ Synthesis for Deployment

In this dissertation we examine the impact of synthesis on software systems through three lenses - design, implementation, and deployment. With Temporal Stream Logic, we have introduced a methodology for leveraging synthesis in designing the high level control flow of functions. With our work on Digital Signal Processing Programming by Example and neural network guided grammar reduction for SyGuS, we have also explored how synthesis can be used for the lower-level implementation aspects of software. In this part of the dissertation, we examine a third and equally critical aspect of software systems, which is their deployment.

In particular, we present a technique for the analysis of configuration files for software systems. Configuration files provide users and developers alike a power interface to change key aspects of how a software system runs. However, with this expressive power also comes a high potential for the introduction of incorrect behavior. In fact, system failures resulting from configuration errors are one of the major reasons for the compromised reliability of today's software systems [163].

We examine two types of configuration files. First, we introduce a technique for analysis of application configurations, found in systems such as Apache and MySQL. The main challenge in dealing with such legacy configuration systems is that lack of formal specifications of the configuration language. As such, we adapt a technique called Association Rule Learning [] to the context of configuration files. Second, we look at analysis of Continuous Integration configurations, where the configuration is less centralized to a single file. In this case, while the concept of a configuration is more difficult to specify, we do have a labelled dataset of correct and incorrect continuous integration configurations. This allows us to leverage supervised learning techniques, such as decision trees, to generate a model that alerts users to potential misconfigurations, while additionally providing an explanation of the source of the error.

Chapter 7

Synthesizing Configuration File Specifications with Association Rule Learning

Configuration file analysis is like sanitation logisitics, no one wants to do it, but everyone is glad someone else is doing it.

- Rahul Dhodapkar

Work presented in this chapter was completed in collaboration with Rahul Dhodapkar, Aaron Shim, Ennan Zhai and Ruzica Piskac. Sections of this work have been previously published [21], and are reproduced here, at times in their original form.

This chapter presents ConfigV, a verification framework for general software configurations. Although many techniques have been proposed for configuration error detection, these approaches can generally only be applied after an error has occurred. Proactively verifying configuration files is a challenging problem, because 1) software configurations are typically written in poorly structured and untyped "languages", and 2) specifying rules for configuration verification is challenging in practice.

Our framework works as follows: in the pre-processing stage, we first automatically derive a specification. Once we have a specification, we check if a given configuration file adheres to that

specification. The process of learning a specification works through three steps. First, ConfigV parses a training set of configuration files (not necessarily all correct) into a well-structured and probabilistically-typed intermediate representation. Second, based on the association rule learning algorithm, ConfigV learns rules from these intermediate representations. These rules establish relationships between the keywords appearing in the files. Finally, ConfigV employs rule graph analysis to refine the resulting rules. ConfigV is capable of detecting various configuration errors, including ordering errors, integer correlation errors, type errors, and missing entry errors. We evaluated ConfigV by verifying public configuration files on GitHub, and we show that ConfigV can detect known configuration errors in these files.

7.1 Motivation

Configuration errors (also known as misconfigurations) have become one of the major causes of system failures, resulting in security vulnerabilities, application outages, and incorrect program executions [164–166]. In a recent highly visible case [167], Facebook and Instagram became inaccessible and a Facebook spokeswoman reported that this was caused by a change to the site's configuration systems. In another case, a system configuration change caused an outage of AT&T's 911 service for five hours. During this time, 12,600 unique callers were not able to reach the 911 emergency line [18]. These critical system failures are not rare – a software system failures study [163], reports that about 31% of system failures were caused by configuration errors. This is even higher than the percentage of failures resulting from program bugs (20%).

The systems research community has recognized this as an important problem and many efforts have been proposed to check, troubleshoot, and diagnose configuration errors [168–171]. However, these tools rely on analyzing the source code of the target systems or need to run the system multiple times to understand the source of the errors. The support for a static analysis style verification for configuration files is still not on the level of automated verification tools used for regular program verification [172–174] that can preemptively detect errors.

There have been several prior efforts that attempt to proactively verify configuration files [171, 175–177]. However, state-of-the-art efforts have limitations that are impractical to meet in practice. In general, these efforts fall into two categories with respect to their limitations.

- Tools that can detect sophisticated configuration errors, *e.g.* , ConfigC [175], but at the cost of scalability. However, ConfigC requires a training set of 100% correct configuration files to extract configuration rules. Such an assumption is too strong to be impractical, because existing investigation studies [163, 178] have demonstrated determining or obtaining 100% correct configuration files to drive rules is almost impossible in reality.

- Tools that do not rely on any strong assumptions *e.g.* , EnCore [176], PCheck [171] and ConfValley [177], but cannot handle as sophisticated errors. These tools have benefit that they do not require 100% correct training setsi In particular, these efforts cannot detect more complex configuration errors, such as ordering errors, fine-grain correlations, missing entry errors.

In this paper, we propose a framework, VeriCI, for automated verification of configuration files, which mimics standard automated verification techniques: for a given configuration file, we checked if it adheres to a specification, and if there are issues we report them to the user. Our framework surmounts both of the aforementioned shortcomings present in existing tools.

There are two main obstacles to directly applying existing automatic program verification techniques to configuration files. First, a lack of specifications on the properties of configuration files makes the verification task poorly defined. There are surprisingly few rules specifying constraints on entries, even written in plain English. Since asking users to write an entire specification for configuration files is impractical and error-prone, we instead take a specification generation approach. Second, the flat structure of configuration files is only a sequence of entries assigning some value to system variables (called *keywords*) and provides little structural information. In particular, the keywords in configuration files are often untyped, a useful property to leverage in program verification.

Our main task was to first derive a specification for configuration files. We first process a training set of 256 industrial configuration files. This training set of files is available online [179]. As these files have been deployed in industry, they contain relatively few errors. This yields a mostly correct training set.

In order to learn a specification from this large set of configuration files, the first step is to translate the training set into a more structured typed representation. We then apply our learning process to learn an abundant set of rules specifying various properties that hold on the given training

set. The learning process is a more expressive form of *association rule learning* [180], which is used to learn predicate relations between multiple keywords in a configuration file. The rules, in general, specify properties such as, pairwise keyword ordering (one keyword must appear before another), or numerical value inequalities (the value of one keyword should always be greater than another). This set of learned rules constitutes a specification for a correct configuration file, which are then used to efficiently check the correctness of a user's configuration files and detect potential errors. The learning process is language-agnostic and works for any kind of configuration files, though all of the files in the training set need to be of the same language (such as MySQL or HTTPD configuration files).

VeriCI detects errors that may cause total system failures, but can also find more insidious errors, such as configurations that will slow down the system only when the server load increases beyond a certain threshold. Since these runtime errors may only be triggered after some time in a deployment environment, the standard debugging techniques [181] of starting a system multiple times with different configuration settings will not help detect these misconfigurations.

VeriCI works by analyzing a training set of partially correct files. A file in the training set might contain several different errors, but errors only appear in a small percentage of files. We first translate those files into an intermediary typed language. VeriCI then learns rules based on the inferred types of the keywords from the training set. However, since the files might also contain errors we take this into account when learning correct rules. All learned rules are annotated with the probability of correctness. To ensure the learned rules are correct enough, VeriCI employs a graph analysis to refine the set of learned rules. It analyzes the rule graph to rank the importance and relevance of the learned rules.

We implemented VeriCI in Haskell and evaluated it on almost 1000 real-world configuration files from GitHub. We demonstrate that we are able to detect known errors, based on StackOverflow posts, in this data set. Furthermore, we find compelling evidence that our optimizations are effective. For example, we introduce a probabilistic type inference system that removes 1043 false positives errors reports to reduce the total fine grained relation errors to 325. The rule graph analysis drastically improves the ranking of importance of the error reports, which allows users to more quickly correct the most critical misconfigurations. Additionally, VeriCI scales linearly, whereas a previous tool [175] showed exponential slow downs on the same benchmark set.

In summary, we make the following contributions:

- We propose the automated configuration verification framework, VeriCI, that can learn specifications from a training set of configuration files, and then use the specifications to verify configuration files of interest.
- We describe the logical foundation of using association rule learning to build a probabilistic specification for configuration files.
- We analyze the learned rules to further refine the generated specification and we empirically show the usefulness of this approach.
- We implement a VeriCI prototype and evaluate it by detecting sophisticated configuration errors from real-world dataset.

7.2 Motivating Examples

In this section we illustrate capabilities of VeriCI by using several real-world misconfiguration examples. These examples are sophisticated configuration errors that were reported on StackOverflow [182], a popular forum for programmers and administrators.

Example 1: Missing Entry Errors. Many critical system outages result from the fact that an important entry was missing from the configuration file. We call such a problem a *missing entry error*. In a public misconfiguration dataset [179], many misconfiguration issues were reported exactly to be missing entry errors. Below is a real-world missing entry error example [183]: when a user configures her PHP and PostgreSQL, she needs to use both `pgsql.so` and `curl.so` in the `/etc/php5/conf.d/curl.ini` configuration file. This is usually achieved by the following entries in the curl configuration file:

```
extension=pgsql.so
extension=curl.so
...
```

However, in this example the user accidentally left out the `extension=pgsql.so` entry, as done by many users [163, 183], causing a segmentation fault. If the user would run VeriCI on her file, our tool returns:

```
MISSING ENTRY ERROR: Expected "extension=pgsql.so"
in the same file: "extension=curl.so"
```

Example 2: Fine-grained Integer Correlation Errors. Our second misconfiguration example [184] comes from a discussion on StackOverflow. The user has configured her MySQL as follows:

```
max_connections             = 64
thread_cache_size           = 8
thread_concurrency          = 8
key_buffer_size             = 4G
max_heap_table_size         = 128M
join_buffer_size            = 32M
sort_buffer_size            = 32M
```

The user then complains that her MySQL load was very high, causing the website's response speed to be very slow. The accepted answer to the post reveals that the value `key_buffer_size` is used by all the threads cooperatively, while `join_buffer` and `sort_buffer` are created by each thread for private use. By further consulting the MySQL manual, we are instructed that when setting `key_buffer_size` we should consider the memory requirement of other storage engines. In a very indirect manner, we have learned that there is a correlation between `key_buffer_size` and other buffer sizes of the system. VeriCI learns the specific constraint that `key_buffer_size` should not be greater than `sort_buffer_size` * `max_connections`. If we run VeriCI on the above configuration file, VeriCI will give an explicit answer:

```
FINE GRAINED ERROR: Expected
  "max_connections" * "sort_buffer_size" > "key_buffer_size"
```

We call these errors *fine-grained integer correlations*. VeriCI can also detect simpler integer correlation: one entry's value should have a certain correlation with another entry's value. For instance, in MySQL, the value of `key_buffer` should be larger than `max_allowed_packet`. While several existing tools [175, 176] can detect simple integer correlation errors, VeriCI is, to the best of our knowledge, the first system capable of detecting such complex fine-grained integer correlation errors.

Example 3: Type Errors. Many system availability problems are caused by assigning incorrect values of an incorrect type to a keyword. Consider the following misconfiguration file from github [185]: a user tries to install MySQL and she needs to initiate the path of the log information generated by MySQL. This user puts the following entry assignment in her MySQL configuration file:

```
slow-query-log = /var/log/mysql/slow.log
```

This misconfiguration will lead to MySQL fails to start [186]. With VeriCI, this user can get the following result:

```
TYPE ERROR: Expected an integer type for "slow-query-log"
```

This was indeed the error, since in MySQL there is another entry named "slow-query-log-file" used to specify the log path.

Example 4: Ordering Errors. Ordering errors in software configurations were first reported by Yin *et al.* [163], but not many existing tools can detect them. The following example contains an ordering error in a MySQL configuration file that causes the system to crash [187].

```
innodb_data_file_path          = ibdata1:10M:autoextend
innodb_data_home_dir           = /var/lib/mysql
innodb_flush_log_at_trx_commit = 1
innodb_lock_wait_timeout       = 50
```

By invoking VeriCI the user receives a correct report that `innodb_home_dir` should appear before `innodb_data_file_path`, as shown below:

```
ORDERING ERROR: Expected "innodb_data_home_dir[mysqld]" BEFORE
  "innodb_data_file_path[mysqld]"
```

7.3 The VeriCI Framework Overview

Fig. 7.1 gives an overview of the VeriCI framework. The main part is dedicated to learning and inferring the specification for configuration files. This process is done offline, before the user even starts to use VeriCI. There are three main steps in the process: translation, learning, and rule refinement.

Translator. The translator module first parses the input training set of configuration files and transforms them into a typed intermediate representation. Entries in a configuration file follow a

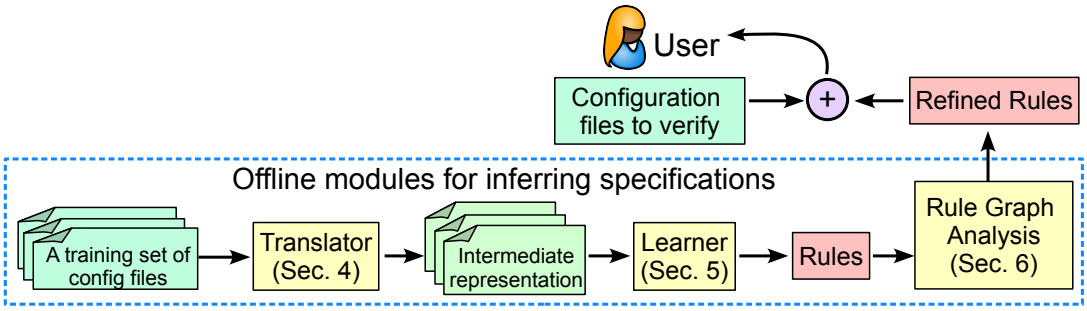


Figure 7.1: VeriCI's workflow. The dashed box is the specification learning module. The yellow components are key modules of VeriCI.

key-value pattern, where some environmental variable ("key") is assigned a value. However, it is not always possible to fully determine the type of the key by inspecting the value at a single entry [166]. We address this problem by introducing *probabilistic types*. Rather than giving a variable a single type, we assign several types over a probability distribution that can later be resolved to type upon which we will learn rules.

Learning. The learner converts the intermediate representation from the translator to a set of rules. It employs a variation on the *association rule algorithm* [180], to generate this list of rules, which describe properties of a correct configuration file. This is a probabilistic verification approach that learns a specification for a correct file over an unlabeled training set of both correct and incorrect configurations files. The learning algorithm uses various instances of a rule interface to learn different classes of rules, such as ordering or integer relations. These rules are then considered necessary for any correct configuration files, and can be used for verification.

Rule Graph Analysis. Finally, the logically structured representation of learned rules allows for a further *rule graph analysis*. The purpose of this module is to refine the learned rules. To this end, we introduce the concept of a rule graph that can be built from the output of the modified association rule learning algorithm. We analyze the properties of this graph to construct a ranking of rules by their importance, as well as to produce a measure of complexity for any configuration of the target system. While the metrics in used in VeriCI are effective, they are not intended to be exhaustive. The information contained in the structured representation of the learned rules is a unique benefit of the learning algorithm, that has potential to be leveraged in many new ways.

7.4 Translator

The translator takes as input a training set of configuration files and transforms it into a typed and well-structured intermediate representation. The translator can be seen as a parser used to generate an intermediate representation for the learner module (cf. Sec. 7.5). Parsing a file for translation is application dependent since each configuration language (MySQL, Apache, PHP) uses a different grammar. VeriCI allows users to provide extra help to the translator for their specific system configurations.

The translator converts each key-value assignment $k = v$ in the configuration file to a triple (k, v, τ), where τ is the type of v. There are two major challenges in this step. First is that configuration files' keywords are not necessarily unique and may have some additional context (modules or conditionals). To solve this, we rely on the fact that keywords in a configuration file must be unique within their context, and rename all keywords with their context. The set of unique keys, $\mathcal{K}$, for the sample training set in Fig. 7.2 would then be `["foo[server]","bar[client]"]`.

Listing 7.1: file1.cnf

```
[server]
foo = ON
[client]
bar = 1
```

Listing 7.2: file2.cnf

```
[server]
foo = ON
[client]
bar = ON
```

Listing 7.3: file3.cnf

```
[server]
foo = OFF
[client]
bar = OFF
```

Figure 7.2: A sample training set of configuration files

Probabilistic Types. An additional challenge is that it is not always possible to fully determine the type of key based on a single example value. For this reason, we introduce *probabilistic types*, as contrasted with *basic types*. In VeriCI, the finite set of basic types contains strings, file paths, integers, sizes, and Booleans. Taking the configuration 7.1, we can assume `foo` is a Boolean type by the grammar of MySQL, but the keyword `bar` could be many types. If we choose the type based on the first example, `bar` will be a integer type. If we choose a type that fits all examples, `bar` will be a string. However the correct classification needed is a Boolean type. For illustrative purposes, let us assume that there is an important rule we must learn about this configuration language, where two Boolean keywords should always have the same values, *e.g.* $eq(\texttt{foo}, \texttt{bar})$. If we take `bar`:: int, we do not learn the above rule, nor do we learn this rule with `bar`:: $string$ - only with `bar`:: $bool$ is the rule is valid. To resolve this ambiguity, and choose the best type, the translator assigns a

$$\frac{\begin{array}{c} c_{int} = |\{\forall C \in \mathcal{TR}.\, \forall (k,v) \in C.\, v \in \mathbb{Z}\}| \\ c_{bool} = |\{\forall C \in \mathcal{TR}.\, \forall (k,v) \in C.\, v \in \{0,1,ON,OFF\}\}| \end{array}}{k : \tilde{\tau}[int = c_{int}, bool = c_{bool}]}\ \textsc{ptype}$$

$$\frac{k : \tilde{\tau} \qquad p_{int}(\tilde{\tau})}{k :: int}\ \textsc{int} \qquad \frac{k : \tilde{\tau} \qquad p_{bool}(\tilde{\tau})}{k :: bool}\ \textsc{bool}$$

$$\frac{k_1 :: \tau \qquad k_2 :: \tau}{eq(k_1, k_2) :: Rule}\ \textsc{eq} \qquad \frac{k1, k2 \in C \qquad k1 \neq k2}{ord(k1, k2) :: Rule}\ \textsc{order}$$

Figure 7.3: Type judgments for a probabilistic type system with $\mathcal{T} = \{bool, int\}$ and an equality and ordering rule

distribution of types to a keyword based on examples from the training set of configuration files (denoted $\mathcal{TR} = \{C\}$).

A probabilistic type is a set of counts over a set $\mathcal{T}$ of basic types. Formally, we define a space of probabilistic types $\tilde{\mathcal{T}}$, where $\tilde{\tau} \in \tilde{\mathcal{T}}$ has the form $\tilde{\tau} = \{(\tau_1, c_1), \ldots, (\tau_n, c_n)\}$, such that $\tau_i \in \mathcal{T}$, $c_i \in \mathbb{Z}$. Fig. 7.3 provides a calculus for probabilistic types over an example subset of basic types. This allows us to give a clear definition of the set of possible (*i.e.* well-typed) equality rules. In a general sense, probabilistic types can be seen as discrete probability distributions over basic types weighted by occurrence.

Every unique keyword $k \in \mathcal{TR}$ has a probabilistic type, expressed $k : \tilde{\tau}$, as opposed to the basic type notation $k :: \tau$. The count for $(\tau_i, c_i) \in \tilde{\tau}$ should be equal to the number of times a key in $\mathcal{TR}$ has a potential match to type τ_i. The judgment PTYPE counts, over all files $C \in \mathcal{TR}$, the times the value of a key matches a user defined set of acceptable values for each type $\tau_i \in \tilde{\tau}$. We use the notation $\tilde{\tau}[\tau_i = N]$ to create a probabilistic type with the count of N for $\tau_i \in \tilde{\tau}$.

In the BOOL judgment, a keyword with a probabilistic type $k : \tilde{\tau} \in \tilde{\mathcal{T}}$ can be resolved to the basic type *bool* when the $\tilde{\tau}$ satisfies the resolution predicate p_{bool}, *i.e.* the probabilistic type has sufficient evidence. The definition of sufficient evidence must be empirically determined by the user depending on the quality of the training set.

In order to define resolution predicates, we use the notation $|\tau_i \tilde{\tau}|$ to select c_i from a $\tilde{\tau} \in \tilde{\mathcal{T}}$. As an example, for the sample training set provided in Fig. 7.2, we might choose to set $p_{bool}(\tilde{\tau}) = |bool_{\tilde{\tau}}| \geq 3 \wedge |int_{\tilde{\tau}}| \leq 1$ and $p_{int}(\tilde{\tau}) = |int_{\tilde{\tau}}| \geq 3$. We then can run inference on that sample

training set to derive a probabilistic type `bar`: $[bool = 3, int = 1]$. This is then resolved, as required in our earlier example, to `bar`:: $bool$.

Note that a user may pick predicates for probabilistic type resolution that result in overlapping inference rules. For example, if a user instead picked $p_{int}(\tilde{\tau}) = |int_{\tilde{\tau}}| \geq 1$, this predicate would overlap with the p_{bool} predicate. This means that basic types are not unique in this system: $k :: \tau \wedge k :: \tau' \not\Rightarrow \tau = \tau'$.

In the case that there is not enough evidence to resolve a probabilistic type to a basic type, no type-dependent rules may be learned over that keyword. However, we are still able to learn rules such as ORDER, which do not require any resolved type.

7.5 Learner

The goal of the learner is to derive rules from the intermediate representation of the training set generated by the translator. We describe an interface to define the different classes of rules that should be learned. Each instance of the interface corresponds to a different class of configuration errors, as described in Sec. 7.2.

To learn rules over sets of configuration files, we use a generalization of *association rule learning* [180], a technique very briefly summarized as inductive machine learning. Association rule learning is used to learn how frequently items of a set appear together. For example, by examining a list of food store receipts, we might learn that when a customer buys bread and peanut butter, the set of purchased items is also likely to include jelly. Since configuration files have multiple complex relations, we extend these association relationships to generalized predicates.

In traditional association rule learning, rules take the form of an implication:

$$r = \{S_0, ..., S_{|S|}\} \in \ valid \ \Rightarrow \{T_0, ..., T_{|T|}\} \in \ valid$$

where S and T are source and target sets of words. This rule states that if the set of words S appear in some $valid$ (*i.e.* non-rule breaking) file, the words T must also appear in that file. We generalize this relation so that a rule, r, instead takes the following form:

$$r = [S_0, ..., S_{|S|}] \in \mathcal{C}_{valid} \Rightarrow \mathcal{C}_{valid} \vdash p([S_0, ..., S_{|S|}], [T_0, ..., T_{|T|}])$$

This rule relates the keywords $S_i, T_j \subseteq \mathcal{K}$ for $0 \leq i \leq |S|, 0 \leq j \leq |T|$ with a predicate p. Whenever the set of keywords S is present in some valid configuration file, $\mathcal{C}_{valid}$, the predicate p must hold over S and T given the context of $\mathcal{C}_{valid}$. In keeping with vocabulary from association rule learning, we refer to S_r and T_r as the source and target keyword lists of rule r respectively. A rule states that if the source list of keyword appears in a valid configuration file, the given predicate must hold between the source and target lists. For clarity in notation, we define all predicates to have arity 2 by separating the source and target keywords into lists of length $|S|$ and $|T|$ respectively, but these can be equivalently thought of as predicates of arity $|S| + |T|$ as notated in Fig. 7.3.

Taking the food store example, our learned rule would be:

$$[bread, butter] \in \ valid \Rightarrow \ valid \vdash assoc([bread, butter], [jelly])$$

where the $assoc$ predicate denotes that the words must all appear in a file. In this way, our formalism is more expressive than the classic association rule learning problem.

The set $\mathcal{K}$ is the set of unique keys from the training set (denoted $\mathcal{TR}$) and the predicate p is one of the classes of configuration errors. The task of the learning algorithm is to transform a training set to a set of rules, weighted with *support* and *confidence*. The set of rules learned from training set $\mathcal{TR}$ constitutes a specification for a configuration file to be considered correct.

The two metrics, support and confidence, are used in association rule learning, as well as other rule based machine learning techniques [188, 189]. We use slightly modified definitions of these, stated below, to handle arbitrary predicates as rules. During the learning process, each rule is assigned a support and confidence to measure the amount and quality of evidence for the rule.

$$\textit{support(r)} = \frac{|\{C \in \mathcal{TR} \mid S_r \cup T_r \subseteq C\}|}{|\mathcal{TR}|}$$

$$\textit{confidence(r)} = \frac{|\{C \in \mathcal{TR} \mid C \ \vdash p_r(S_r, T_r)\}|}{support(r) * |\mathcal{TR}|}$$

Support is the frequency that the set of keywords in the proposed rule, $S \cup T$, have been seen in the configuration files C in the training set $\mathcal{TR}$. Confidence is the percentage of times the rule predicate has held true over the given keywords. In the learning process, each class of rule is manually assigned a support and confidence threshold, t_s and t_c respectively, below which a rule will be rejected for lack of evidence. We denote the set rules that are learned and included as part of the final specification are as follows:

$$\begin{aligned} Learn(\mathcal{TR}) = \{r|\ & support(r) > t_s \wedge \\ & \textit{confidence(r)} > t_c \wedge \\ & S,\ T \subseteq \mathcal{K}\} \end{aligned} \tag{7.1}$$

7.5.1 Error Classes

Each class of error forms a rule with a predicate p, and choices of $|S|$ and $|T|$, the sizes of the source and target keyword lists. Ordering errors require $|S| = 1, |T| = 1$ and use the predicate $order$ which means if both keywords S_0, T_0 appear in a file, S_0 must come before T_0. For example, the ordering error given in Sec. 7.2 is expressed:

$$[\texttt{innodb\_data\_home\_dir}] \in \mathcal{C}_{valid} \Rightarrow$$
$$\mathcal{C}_{valid} \vdash order([\texttt{innoddb\_data\_home\_dir}], [\texttt{innodb\_data\_file\_path}])$$

Missing keyword entry errors require $|S|, |T| = 1$ and use the predicate $missing$ to mean the keyword T_0 must appear, in any location, in the same file as the keyword S_0 in any configuration file. The type rule is a set of rules over multiple predicates, where $|S| = 1, |T| = 1$. These type rules take the form $S_0 \in \mathcal{C}_{valid} \Rightarrow \mathcal{C}_{valid} \vdash is\_([S_0], [S_0])$ where _ (underscore) matches all the basic types (bool, int, size, etc), as shown in Fig. 7.4. In this case, the source and target are the same keyword, since a type constraint is only on a single keyword.

VeriCI also supports two types of integer correlation rules, coarse-grained and fine-grained. Both integer correlation rules are set of rules over the set, $compare$, of predicates $\{<, =, >\}$. Coarse grain rules require $|S|, |T| = 1$, and the predicates have the typical interpretation. Fine-grained rules use $|S| = 2, |T| = 1$, and interpret the predicates such that for $k_1,\ k_2 \in S,\ k1 * k2$ must have the

$$\frac{k_1 :: bool}{isbool([k_1],[k_1]) :: Rule}\ \text{BOOL} \qquad \frac{k_1 :: int}{isint([k_1],[k_1]) :: Rule}\ \text{INT}$$

$$\frac{}{missing([k_1],[k_2]) :: Rule}\ \text{MISSING} \qquad \frac{k_1, k_2 :: int}{compare([k_1],[k_2]) :: Rule}\ \text{COARSE\_GRAIN}$$

$$\frac{k_1, k_2, k_3 :: int}{compare([k_1, k_2],[k_3]) :: Rule}\ \text{FINE\_GRAIN} \qquad \frac{k_1, k_3 :: size \qquad k_2 :: int}{compare([k_1, k_2],[k_3]) :: Rule}\ \text{FINE\_GRAIN}$$

Figure 7.4: An expanded set of typing judgements for valid rules

predicate relation to T_0. The integer correlation rules also use probabilistic types to prune the search space. To avoid learning too many false positives, we restrict this rule to either $size * int = size$, or $int * int = int$, as shown in Fig. 7.4. Without probabilistic typing, we would also learn rules which do not have a valid semantic interpretation, for example a relation between three size keywords ($k1, k2, k3 :: size$) of the form $k1 * k2 > k3$.

We chose the predicates in Fig. 7.4 based on our own experience, as well as prior issues identified in [163, 166]. Other predicates might also be interesting for other configuration languages, for example comparing IP subnets. Predicates are not language dependent and we have defined the core predicates that should be needed for most configuration languages here. If a user requires other predicates for their particular use case, they may be added following the same format as above.

7.5.2 Checker

With the rules generated by the learner module, VeriCI checks whether any entry in a target configuration file violates the learned rules and constraints. VeriCI parses a single verification target configuration file with the translator from Sec. 7.4 to obtain a set of key-value pairs, $\mathcal{C}$, for that file. Then, the checker applies the learner from Eq. 7.1 with *Learn*($\mathcal{C}$), to build the set of relations observed in the file, with the thresholds $t_s, t_c = 100\%$. The checker will then report the following set of errors:

$$\begin{aligned} \textit{Errors}(\mathcal{C}) = \{r \mid\ & S_r \cup T_r \in \mathcal{C} \wedge \\ & r \in \textit{Learn}(\mathcal{TR}) \wedge \\ & r \notin \textit{Learn}(\mathcal{C}) \end{aligned}$$

For any relation from the verification target that violates a known rule, the checker will report the predicate and keyword sets associated with that rule as an error. Since this is a probabilistic approach, in our tool VeriCI, we provide the user with the support and confidence values as well to help the user determine if the rule must be satisfied in their particular system. For instance, the `key_buffer` misconfiguration from Sec. 7.2 will only be noticeable if the system experiences a heavy traffic load, so the user may choose to ignore this error if they are confident this will not be an issue.

7.6 Rule Graph Analysis

The learner outputs a set of rules learned from the training set as described in Sec. 7.5. Recall that a rule is an implication relationship of the basic form $r = S_r \in \mathcal{C}_{valid} \Rightarrow \mathcal{C}_{valid} \vdash p(S_r, T_r)$. This data is necessary to perform the core verification task, but can also be used for further analysis. By interpreting the rules as a graph (which we call the *rule graph*), we can use tools from graph theory to extract information about the configuration space that can improve the quality of the learned model. We inspect properties of this rule graph to sort reported errors by those most likely to be valid. To demonstrate the additional value of the rule graph, we also use it to estimate the complexity of a configuration file.

Accessibility of the rule graph is a useful property of the association rule learning technique applied by VeriCI. While it is possible to analyze the models from other machine learning techniques, such as neural networks [190] and conditional random fields [191], these analyses require a deep knowledge of the applied techniques. In contrast, the rule graph is a relatively simple, yet information rich, representation of the learned model. The following section provides a precise definition of the rule graph and demonstrates useful metrics we derive for the purposes of ranking reported errors and complexity analysis.

7.6.1 Rule Ordering

We define the *rule graph* as a directed hypergraph $H = (V, E)$, with vertices $V = \{keywords\}$ and labeled, weighted edges $E = \{(V_s, V_t, l, w)\}$. The set of edges is constructed from the learned rules, using the source and target keyword sets as sources and targets respectively, the predicates as labels, and the confidence as weights:

$$\forall r \in Learn(\mathcal{TR}).\ \exists e\ \in E.$$
$$V_s = S_r \wedge V_t = T_r \wedge l = p_r \wedge w = \textit{confidence(r)}$$

We will also denote $E_{V_1,V_2} \subset E$ as the *slice set* of E over V_1, V_2. We can think of E_{V_1,V_2} as the subset of edges in E such that each source set V_s shares a vertex with V_1 and each target set V_t shares a vertex with V_2. Formally:

$$E_{V_1,V_2} = \{(V_s, V_t, l, w) \in E\ \mid \exists v_1 \in V_1 \wedge v_1 \in V_s \wedge \exists v_2 \in V_2 \wedge v_2 \in V_t\}$$

We denote a standalone vertex v in our subscripts as notational convenience for the singleton set containing that vertex v.

The size of an edge set, $|E|$, is the sum of all weights in that set. The use of the support and confidence thresholds t_s and t_c in the learner ensure that all weights in the rule graph are positive. We then define a measure of degree, $\mathcal{D}(v)$, for each vertex v as the sum of in-degree and out-degree.

$$|E| = \sum_{(S,T,l,w)\in E} w$$

$$\mathcal{D}(v) = \sum_{v'\in V} |E_{v,v'}| + \sum_{v'\in V} |E_{v',v}|$$

As an example of the rule graph construction and degree calculation, consider the rules presented in Fig.7.5. We assume that those rules are the result of learning over some training set, and have confidence values as notated in the graph. We can then construct the rule graph as demonstrated on the right of Fig. 7.5. The rule graph is then used to calculate the degree of the keywords, which indicate the estimated importance of each keyword.

We may now use this measure to rank our errors. The more rules of high confidence are extracted for a keyword by the learner, the higher the $\mathcal{D}(v)$ of the corresponding vertex in the rule graph. In our final analysis, we use this classification of rules to order the reported *errors* by estimated importance.

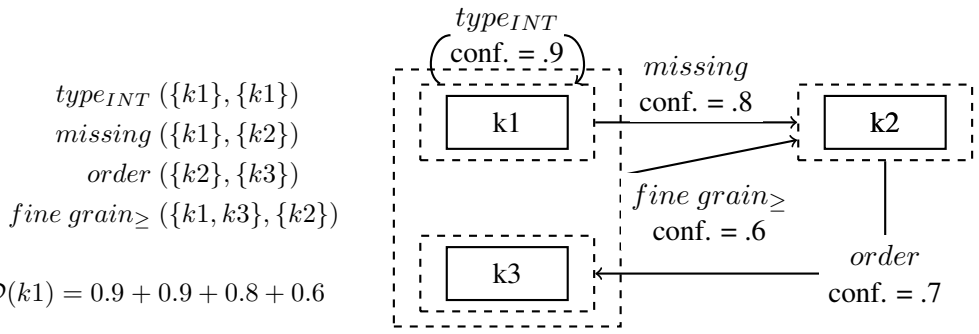


Figure 7.5: A rule graph constructed from the example rules over keywords k1, k2, k3 on the left. Dashed lines indicate the source and target vertex sets for each rule. The rules $type_{INT}$, $missing$, and $fine\ grain_{\geq}$ are in the slice sets used to calculate the degree of k1.

Keywords (specifically their corresponding vertices) of low $\mathcal{D}(v)$ may be rarer configuration parameters where rules learned are more likely to be governed by technical necessity, rather than industry convention. As such, errors reported involving low-degree keywords are more likely to be errors of high significance and should be presented with high importance to users of VeriCI.

Specifically, for an error reported by VeriCI on a rule r involving keywords K, we rank the errors by:

$$\textit{RANK(r)} = \frac{\sum_{k \in K} \mathcal{D}(k)}{|K|}$$

The results from ranking errors in this way are presented in Sec. 8.5.

7.6.2 Complexity Measure

We may also use the rule graph to advance our general knowledge of the configuration space, outside the strict confines of a verification system. As an example, we present a heuristic for configuration file complexity based on the topology of the rule graph. This measure of complexity could be used by software organizations to manage configuration files in much the same way as Kolomogrov complexity [192] is used to manage code - identifying potentially brittle configurations for targeted refactoring.

For a configuration file with a set of keywords K and a rule graph $H = (V, E)$, we define our complexity measure:

$$\mathcal{C}(K,H) = \sum_{k \in K} \begin{cases} 1 * (1 - \frac{|E_{k,K}|}{|E_{k,V}|}) & \text{if } |E_{k,V}| > 0 \\ 1 & \text{otherwise} \end{cases} \tag{7.2}$$

The complexity measure, $\mathcal{C}$, can be thought of as an extension of the naïve line-counting measure of complexity. When a keyword in the configuration file is present in the rule graph, we may consider the set $E_{k,K}$ to be all learned rules involving keyword k that are relevant to the configuration file being examined. The set $E_{k,V}$ denotes *all* learned rules involving k. Given these sets, we may think of $\frac{|E_{k,K}|}{|E_{k,V}|}$ as representing the amount that k is constrained in the current configuration file relative to how much it could be constrained in the global configuration space. The more constrained a configuration keyword in a particular configuration file, the less it should contribute to the complexity (hence $1 * (1 - \frac{|E_{k,K}|}{|E_{k,V}|})$). If a keyword is not constrained at all in the current configuration file or is not present in the rule graph, we revert to the standard counting metric of complexity.

While an in-depth evaluation of the complexity metric presented here is out of scope for this paper, we use this measure to demonstrate the flexibility of the rule graph, and potential for further applications.

7.7 Implementation and Evaluation

We have implemented a tool, VeriCI, and evaluated it based on real-world configuration files taken from GitHub. VeriCI is written in Haskell and is available open source at `https://github.com/santolucito/configV`. Thanks to Haskell's powerful type system, the implementation can easily be extended with new rule classes or applied to different configuration languages with minimal change to the rest of the code base. When adapting VeriCI to a new language, the key components that must be provided are a parser into our intermediate representation and a training set of configuration files in the new language. In some cases, a user may also require new types or predicates to learn over. In this case, a user only needs to provide the functions for the rule interface (a typeclass in Haskell) to 1) learn relations from a single file 2) merge two sets of rules and 3) check a file given some set of rules.

7.7.1 Evaluation

To evaluate our VeriCI prototype, we require a separate training set and test set. For the training set, $\mathcal{TR}$, we use a preexisting set of 256 industrial MySQL configuration files collected in previous configuration analysis work [179]. This is an unlabeled training set, though most of the files have some errors. For the test set, we randomly sampled 1000 MySQL configuration files from GitHub, and filtered the incorrectly formatted files out for a final total of 973 files. While our tool can detect incorrectly formatted files, we exclude them here as they do not add any useful information to the training set. The focus of our evaluation are MySQL files for comparability of results, but VeriCI can handle any configuration language (that can be parsed to the intermediate representation from Sec. 7.4).

We report the number of rules learned from the training set and the number of errors detected in the test set in Table 7.1. One interesting note is that without probabilistic types we learned 327 fine grained rules and detected 1368 fine grained errors. By introducing probabilistic types, we remove 224 incorrect rules and thereby remove 1043 false positives. We are guaranteed these are all false positives since there cannot be a correct rule of an incoherent type. For example, comparing the types $size * size$ and $size$ is not meaningful, and thus cannot be a correct rule, because of the semantic interpretation of the $size$ units as memory.

We also provide the support and confidence thresholds, t_s, t_c, used in this evaluation. These thresholds can be adjusted by the user to control the level of assurance that their file is correct. The ideal setting will depend on both the user preference and training set quality. In our evaluation, we adjust the support and confidence values such that when VeriCI is run on a file with no more than 3 known errors, VeriCI reports those errors and no others. Following common practice from association rule learning, initial values for support are confidence are generally in the range of 10% and 90% respectively.

We record the histogram of errors across the test set in Fig. 7.6. This is intuitively an expected result from randomly sampling GitHub – most repositories will have few errors, with an increasingly small number of repositories having many errors. When sampling repositories from GitHub, we searched for any files with the common MySQL extension '.cnf'. To ensure the files we found were really configurations for MySQL we searched specifically for files that contained keywords common

Table 7.1: Results of VeriCI

Class of Error	# Rules Learned	# Errors Detected	Support	Confidence
Order	12	63	6 %	94 %
Missing	53	55	2 %	71%
Type	94	345	12 %	70%
Fine-Grain	103	325	24 %	91%
Coarse-Grain	97	237	10 %	96%

to MySQL such as ‘mysql’ and ‘size’ and ‘innodb’. The repository sampling had a wide variety of sizes and contributors. The average number of contributors per repository was 5.85 with a standard deviation of 9.73. The average number of commits per repository was 2767.04 with a standard deviation of 10362.57. These numbers indicate the wide range of repositories sampled.

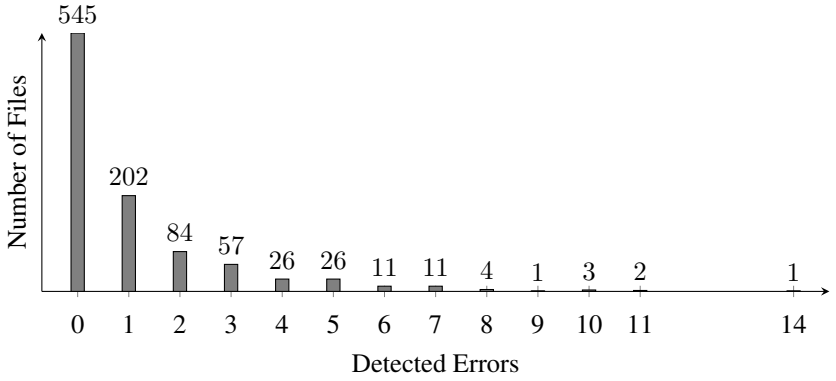


Figure 7.6: Histogram of errors

The errors reported over the GitHub files may have varying impacts on the system, ranging from failing to start, runtime crash, or performance degradation. However, since VeriCI is a probabilistic system, it is also possible that some errors are false positives - that is, a violation of the rule has no effect on the system. Note that in contrast to program verification, we do not have an oracle for determining if a reported error is a true error or a false positive. While we can run a program to determine the effect a specification has on the success of compiling/running the program, no such test exists for configuration files. Because configurations are dependent on the rest of the system (*i.e.* , the available hardware, the network speed, and the usage patterns), we cannot simulate all the conditions to determine if a reported error will cause system failure. As evidenced by Example 7.2, some misconfigurations will only cause greater than expected performance degradation, and only under particular traffic loads. In light of this, the definition of a true error is necessarily imprecise.

7.7.2 Issue Reporting Back to Users

As one metric to help evaluate the usefulness of our tool, we submitted GitHub issue reports for these bugs. Too avoid unnecessary burden on these users, we reported only at most the top three issues for each configuration file. Additionally, we submitted at most one report per user, so that users with multiple configuration files that we tested would receive a single message. Applying these filters resulted in a total of 215 issue reports submitted to open source repositories – the issues can be found at through the GitHub search at `https://goo.gl/KNFuuz`. At the time of writing this paper, we have received feedback on 21 of these issues. Because of the variety of feedback, we present a selection of the responses from users.

An interesting point is two users had opposite reactions to the same error. The error `Expected query-cache-type[mysqld] <= max-allowed-packet[mysqld]` was reported to two different users. While one user was appreciative and interested, replying with the message "Wow, thanks guys", the other user responded in colorful language that he does not believe that these two values should even be compared. A number of users responded that they could not provide feedback because the repository was outdated or not maintained and they were not sure of the effect such an error might have on the system.

We also received feedback that helped us discover some limitations of our training set and pointed us towards future improvements. For example, when we reported the error `Expected "key-buffer [isamchk]" in the same file as "[isamchk]"`, the user commented that `key-buffer` is a depreciated keyword that has been replaced with `key-buffer-size`. In this case, our training set of files had been used to configuration an earlier version of MySQL than the target verification file of the user. The error in this case is in fact a true positive for earlier versions of MySQL, but in later versions, `key-buffer-size` is preferred instead.

While this use case highlighted a limitation of our training set, it also provides further motivation that such a tool is important. As configuration specifications are constantly changed, maintaining a manually written set of specification is an increasingly impractical task. We have learned from this case study that, in practice, it will be important to ensure the training set of is using the same, or similar, system version as the verification target.

Table 7.2: Sampled misconfiguration files for error detection evaluation.

Errors	URLs	None	RG	PT	RG∧ PT
ORDERING ERROR: Expected "innodb_data_home_dir" BEFORE "innodb_data_file_path"	[193]	12/12	3/12	5/5	3/5
	[194]	11/11	2/11	3/3	3/3
	[195]	9/9	3/9	4/4	3/4
MISSING ERROR: Expected "key_buffer" WITH [isam-chk]	[196]	6/10	2/10	2/4	2/4
	[197]	2/3	3/3	2/3	3/3
	[198]	2/3	3/3	2/2	3/3
TYPE ERROR: Expected bool for "thread_cache_size"	[199]	1/12	1/12	1/4	1/4
	[200]	1/9	1/9	1/1	1/1
	[201]	1/8	1/8	1/3	1/3
FINE GRAINED ERROR: Expected "max_connections" * "sort_buffer_size" > "key_buffer_size"	[202]	30/34	18/34	6/7	3/7
	[203]	23/25	9/25	8/9	3/9
	[204]	20/23	14/23	6/7	5/7
INTEGER CORRELATION ERROR: Expected "max_allowed_packet" < "innodb_buffer_pool_size"	[205]	29/32	8/32	11/14	4/14
	[206]	22/23	2/23	9/10	2/10
	[207]	10/12	4/12	4/4	2/4

7.7.3 Sorting Error Reports

Although we cannot identify false positives, we can identify true positives by examining online forums, like StackOverflow. On these forums we find reports that particular configuration settings have caused problems on real-world systems. Furthermore, any error for which we can find evidence online is likely to be more problematic than errors that do not have an online record, using the reasoning that this error has caused enough problems for people to seek help online. Given that some errors are more severe than others, we would like VeriCI to sort the errors by their importance or potential severity. To achieve this sorting we use the rule graph analysis metric described in Sec. 7.6.1.

To estimate the impact of this metric, we track the rank of known true positives with, and without, the augmented rule ordering in Table 7.2. For this table, we picked the known true positive rules, listed in the Errors column, and pick configuration files in the test set that have these errors. We picked 3 files for each true positive by choosing the files with the highest number of total reported errors in order to clearly observe the effects of our optimizations. Although this gives a more clear picture of the effect of our optimizations, it results in a slightly inflated false positive rate.

We test the following conditions; just rule graph analysis (RG) to sort the errors, just probabilistic types to filter the rules (PT), and both optimizations at the same time (RG $\wedge$ PT). For each entry we list X/Y, where X is the rank of the known true positive, and Y is the total number of errors found in that file.

7.7.4 False Positive Rate

Because VeriCI detects complex and subtle misconfigurations that, for example, may cause performance degradation in a high traffic load, false positives are system and use-case dependent and therefore ill-defined. However, we report an estimation of the false positive rate for comparison to other tools. To estimate a false positive rate, we asked two industry experts, one from MongoDB and one from Microsoft, to independently classify all errors from Table 7.2. For each unique error reported in Table 7.2 (a total of 70 unique errors), the expert was asked to classify the error as: definitely false positive, potential true positive, or definitely true positive. The MongoDB expert rated 13/70 errors as definitely false positives. The Microsoft expert rated 8/70 errors as definitely false positives. The similarity between experts is suggestive that these are approximately correct classifications.

The resulting false positive rate is then estimated to be 11%-18%. This is within the range of the false positive rate in existing work. For example in the EnCore tool, which had a false positive rate of 13%, 21%, 32% for MySQL, Apache, and PHP respectively [176]. We note again that as opposed to a tool like EnCore, which is used mainly to detect initialization errors, thanks to the complex relations that can be learned, VeriCI also learns misconfigurations causing runtime performance degradation. This means VeriCI generates a larger rule set, and false positives cannot be guaranteed, *i.e.* there may be some environment conditions that will cause a "false" positive to become a true positive.

In contrast, a true positive can be confirmed as such based on evidence of unwanted system behavior. The errors listed in Table 7.2 are confirmed true positives, evidenced by posts on help forums. VeriCI detected and reports these errors in the 15 code repositories listed in the URL column of Table 7.2. These are real-world configuration files that contain errors that may be unknown to the maintainers of the repositories.

7.7.5 Runtime Performance

We also evaluate the speed of VeriCI. Generally, once a set of rules has been learned, it is not necessary to rerun the learner. However, using VeriCI on a new language does require rerunning the learner. We have only used VeriCI to build rules for MySQL here, but any configuration language

Table 7.3: Time for training over various training set sizes

# of Files for Training	ConfigC (sec)	VeriCI (sec)
0	0.051	0.051
50	1.815	1.638
100	13.331	4.119
150	95.547	10.232
200	192.882	12.271
256	766.904	15.627

might be analyzed with VeriCI given a parser and training set. Additionally, in an industrial setting, the available training set may be much larger than ours, so it is important that the learning process scales. We see in Table 8.6 that VeriCI scales roughly linearly.

We compare VeriCI to our prior work in configuration verification, ConfigC [175]. Although ConfigC does not support as advanced features as VeriCI, such as fine-grained correlations, probabilistic types, or rule graph ordering, we report training times here for both tools over the same training set on configuration files. ConfigC scales exponentially because the learning algorithm assumes a completely correct training set, and learns every derivable relation. With VeriCI, we instead only process rules that meet the required support and confidence, reducing the cost of resolving to a consistent set of rules. The times reported in Table 8.6 were run on four cores of a Kaby Lake Intel Core i7-7500U CPU @ 2.70GHz on 16GB RAM and Fedora 25. Even with support for the more advanced features, VeriCI still far outperforms ConfigC.

7.8 Related Work

Configuration verification has been considered a promising way to tackle misconfiguration problems [164]. Nevertheless, a practical and automatic configuration verification approach still remains an open problem.

Language-support misconfiguration checking. There have been several language-support efforts proposed for preventing configuration errors introduced by fundamental deficiencies in either untyped or low-level languages. For example, in the network configuration management area, administrators often produce configuration errors in their routing configuration files. PRESTO [208] automates the generation of device-native configurations with configlets in a template language. Loo *et al.* [209]

adopt Datalog to reason about routing protocols in a declarative fashion. COOLAID [210] constructs a language to describe domain knowledge about devices and services for convenient network reasoning and management. Compared with the above efforts, VeriCI mainly focuses on software systems, *e.g.* , MySQL and Apache, while our main purpose is to automate configuration verification rather than proposing new languages to convenient configuration-file writing.

The closest effort to VeriCI is ConfigC [175], which aims to learn configuration-checking rules from a given training set. Compared with VeriCI, ConfigC has the following disadvantages. First, ConfigC requires the configuration files in the training set must be correct, which is impractical because it is very difficult to determine a correct configuration set in reality. On the contrary, since VeriCI does not rely on the quality of configuration files in training sets, VeriCI is much more practical than ConfigC. Second, ConfigC covers fewer types of misconfigurations than VeriCI. For example, ConfigC cannot detect fine-grained integer correlation errors. Finally, the training time of ConfigC is much longer than VeriCI (as shown in Table 8.6). Furthermore, ConfigC was originally presented only as a tool, and did not provide the theoretical framework that would allow it to be used in other applications. This paper has provided an succinct explanation additionally both for VeriCI as well as giving ConfigC a theoretical context to more accurately compare against VeriCI.

Misconfiguration detection. Misconfiguration detection techniques aim at checking configuration efforts before system outages occur. Most existing detection approaches check the configuration files against a set of predefined correctness rules, named constraints, and then report errors if the checked configuration files do not satisfy these rules. Huang *et al.* [177], for example, proposed a language, ConfValley, to validate whether given configuration files meet administrators' specifications. Different from VeriCI, ConfValley does not have inherent misconfiguration checking capability, since it only offers a language representation and requires administrators to manually write specifications, which is an error-prone process. On the contrary, VeriCI does not need users to manually write anything.

Several machine learning-based misconfiguration detection efforts also have been proposed [171, 176, 211]. EnCore [176] introduces a template-based learning approach to improve the accuracy of their learning results. The learning process is guided by a set of predefined rule templates that enforce learning to focus on patterns of interest. In this way, EnCore filters out irrelevant information and reduces false positives; moreover, the templates are able to express system environment information

that other machine learning techniques cannot handle. Compared with EnCore, VeriCI has the following advantages. First, VeriCI does not rely on any template. Second, EnCore cannot detect missing entry errors, type errors, ordering errors and fine-grained integer correlation errors, but VeriCI can detect all of them. Finally, VeriCI is a very automatic system, but EnCore needs significant human interventions, *e.g.* , system parameters and templates.

PCheck [171] aims to add configuration checking code to the system source code by emulating potential commands and behaviors of the system. This emulation is a "white-box" approach and requires access to the system's source code. One drawback to this approach is that for some systems (*e.g.* , ZooKeeper) whose behavior is hard to emulate, PCheck cannot automatically generate the corresponding checking code. Due to the emulation based testing strategy, PCheck's scope is limited to reliability problems caused by misconfiguration parameters. In contrast, VeriCI is a "black-box" approach and only requires a training set of configuration files to learn rules. By using a rule learning strategy of examples, VeriCI is able to detect general misconfiguration issues that are outside the scope of emulation testing (*e.g.* memory or thread usage settings), including performance, security, availability and reliability.

Misconfiguration diagnosis. Misconfiguration diagnosis approaches have been proposed to address configuration problems post-mortem. For example, ConfAid [168] and X-ray [212] use dynamic information flow tracking to find possible configuration errors that may have resulted in failures or performance problems. AutoBash [169] tracks causality and automatically fixes misconfigurations. Unlike VeriCI, most misconfiguration diagnosis efforts aim at finding errors after system failures occur, which leads to prolonged recovery time.

Association Rule Learning. The approach we have presented not only generalizes association rule learning, but also another learning strategy called *sequential pattern mining* [213]. Ordering rules are similar to the patterns learned in sequential pattern mining, although we restrict our ordering rules to $|S|, |T| = 1$ since these are the most common misconfigurations we encounter in practice. While a major limitation to sequential pattern mining is the scalability of the problem [214], we escape this issue with $|S|, |T| = 1$ and the fact that a single configuration file tends to be less a few thousand lines. There has been other work in these same classes of algorithms [188, 189] for various applications and variations on the core problem. A future direction for this work is to integrate advances in these domains into the configuration file verification problem.

Chapter 8

Continuous Integration Configurations

Work presented in this chapter was completed in collaboration with Jialu Zhang, Jurgen Cito, Ennan Zhai, and Ruzica Piskac. Sections of this work have been previously published [215], and are reproduced here, at times in their original form.

Continuous Integration (CI) allows developers to check whether their code can build successfully and pass tests across various system environments with every commit. To use a CI platform, a developer must provide configuration files within a code repository to specify build conditions. Incorrect configuration settings lead to CI build failures, which can take hours to run, wasting valuable developer time and delaying product release dates. Debugging CI configurations is a slow and error-prone process. The only way to check the correctness of CI configurations is to push a commit and wait for the build result. We present VeriCI, the first system for statically detecting errors in a given CI configuration before the developer sends the build request to the CI server. Our key insight is that CI environments contain previously underutilized information in the form of the commit history and the corresponding build histories. We leverage that labeled dataset to generate customized rules describing correct CI configurations, using supervised machine learning techniques. To more accurately identify root causes, we train a neural network that filters out constraints that are less likely to be connected to the root cause of build failure. We evaluate VeriCI on real world data from GitHub and achieve 91.0% accuracy of predicting a build failure and correctly identify the root cause in 88% of cases. We also conducted a between-subjects user study with 20 software developers, showing that VeriCI significantly helps users in identifying and fixing errors in CI.

8.1 Motivation

Continuous Integration (CI) is seeing broad adoption with the increasing popularity of the GitHub pull-based development model [216]. There are now a plethora of open-source, GitHub-compatible, cloud-based CI tools, such as TravisCI [217], CircleCI [218], Jenkins [219], GitLab CI [220], Codefresh [221] and TeamCity [222]. Over 930,000 open-source projects are using TravisCI alone [217]. A CI platform provides developers with continuous feedback on every commit indicating if their code successfully built and whether the given tests passed.

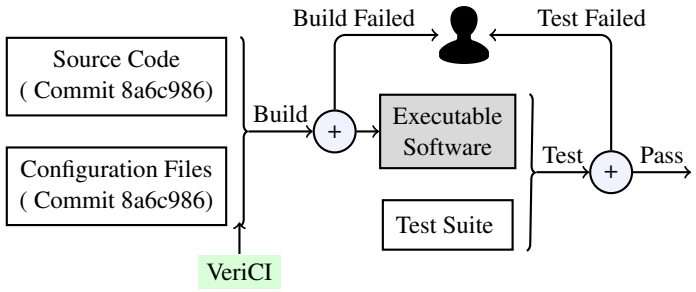


Figure 8.1: A typical build process of CI. Our tool, VeriCI, statically analyzes a commit to predict failing builds and reports the root cause of the failure.

Fig. 8.1 presents the typical development workflow when using CI. A developer provides code, a test suite, and CI configuration files. These configuration files specify build conditions such as the operating system, disk size, which compiler flags to use, which library dependencies are required, and similar properties. Then, each time a developer commits new code, the CI platform builds multiple versions of the executable software according to the configuration, with each version corresponding to the specified build conditions. The CI platform then runs these executables on the test suite with appropriate hardware. It has been shown that CI speeds up development (code change throughput) [223, 224], helps to maintain code quality [225, 226], and allows for a higher release frequency which "radically decreases time-to-market" [227].

However, for all the benefits CI brings, it also suffers from some serious limitations. If the executable fails to build, the tests cannot be executed, and the developer must try to fix the configuration before any functional correctness can be checked. For instance, CI builds for large projects can take hours, and in particularly painful cases, some builds can take almost 24 hours [228]. Furthermore, if the CI configuration is incorrect (e.g. uses a library incompatible with the specified OS) and the project fails to build, the user does not receive any feedback on the functional correctness of their

code. By running a query[1] on the historical build data available on TravisTorrent [229], we find that 662,442 hours or 75 years of server time was spent on builds that eventually resulted in a failed build. Such incorrect CI configurations result not only in lost server time, but also lost developer time, as developers must wait to verify that their work does not break the build. As one example, the release of a new version of the open-source project, SuperCollider, was delayed for three days because of slow TravisCI builds [230].

One of the main sources of difficulty in debugging CI configurations is that a developer cannot test the configuration for platforms that are not available locally. The only way to test a configuration is to push a commit and wait for the CI platform to complete the build. If these incorrect configurations could be quickly and statically checked on the client side before the build, the CI workflow would be considerably improved.

Learning Static Analyzers for CI Configurations.. To address these issues, we developed a tool, VeriCI, for detecting potential errors in a given CI configuration *before* the developer sends the build request to the CI platform. VeriCI is a language agnostic approach that learns correctness criteria for CI environments. A user can invoke VeriCI to predict the build status of their commit before the build is sent to the CI platform. Furthermore, VeriCI is the first tool with the ability to localize and report the root cause of an error when a build is likely to fail.

In VeriCI, we leverage two unique properties of the CI environment. First, as a result of integration with version control systems, the CI also comes with a labeled dataset of build logs over the commit history. Using this labeled data as a training set, we use supervised learning techniques, such as decision trees and neural networks, to learn properties of successful and failing builds. An example of such a property is version inconsistencies between libraries. We introduce a new type of decision tree learning, specialized to the task of generating error messages that help users identify the root cause of a build failure. Second, we use the fact that in our training set, as a product of being a version controlled repository, there is an incremental change between successive commits. This allows us to narrow down the search space for the root cause of the failed builds. We additionally combine our decision tree algorithm with a neural network to increase the accuracy of our error localization.

1. `https://bigquery.cloud.google.com/savedquery/199893009665:20252e2eab5c4e2c9105514cce4417f8`

We evaluated VeriCI on real world data from GitHub. VeriCI achieves 91.0% accuracy when predicting a build failure and correctly identifies the root cause in 88% of cases. Additionally, to assess the efficacy of generated error messages, we conducted a between-subjects user study with 20 software developers. The control group had access to the standard TravisCI output, while the treatment group was additionally provided with the output of VeriCI. Our study shows that VeriCI significantly helps users in identifying and fixing errors in CI. However, in cases where the root cause is easily extractable from the log file, we did not see significant differences between control and treatment groups.

Building tools for configuration support, management, and verification has been an active area of research [168, 169, 171, 231–234]. The topic of CI analysis in particular was explored in the 2017 Mining Software Repositories Mining Challenge [235]. While some of the work presented there focused on build prediction, these systems utilized metadata (commit messages, commit size, repository size, etc.) to make their prediction. These systems provided high accuracy of predicting build status, but have not yet addressed the issue of providing feedback to users that helps to fix failing builds.

In summary, this work makes the following new contributions:

1. Introduces the first approach to predicting CI build status that also identifies probable root cause locations in the source code and generates human-readable error messages.

2. An evaluation of our tool, VeriCI, that achieves 91.0% accuracy of predicting a build failure and identifies the root cause location in 88% of cases.

3. A user study that show developers using VeriCI can identify and fix potential CI build fixes more accurately for some cases, while highlighting limitations for other cases.

8.2 Motivating Examples

We give here two examples to illustrate the types of failures that developers face in the CI environment. In these examples, VeriCI generates error messages to guide developers to identify the root cause of the build failure.

8.2.1 Identifying an error within a single file

We start with an example of a TravisCI build failure from the sferik/rails_admin repository [236] on GitHub. This is a large repository with (at the time of writing) 4560 commits over 364 contributors, 7149 stars, and 2099 forks.

In commit `3fd3b32` an administrator merged the changes from a pull request into the master branch, which caused the TravisCI build to fail. The pull request changed 3 files by adding 125 lines and deleting 10 lines. Manually checking the log information is a tedious process that, in this case, did not provide any helpful information for understanding or correcting the issue. As a result, the next 14 commits in the repository still have the same failed TravisCI status. It was not until 20 days later that a contributor to the repository was able to correct this build failure [237].

We ran VeriCI on commit `3fd3b32`, which caused the build to fail, and VeriCI correctly predicted a build failure. Additionally, VeriCI also provided an explanation for its classification, by reporting two substrings from the codebase which are likely to be contributing to the build failure.

```
Predicted build failure based on these keywords
   - if options[:encoding_to].present? &&
        @encoding_to == Encoding::UTF_
   - rvm
```

We confirmed this as a true positive, as this is exactly the location of the change made by the user when fixing the build failure [237]. The user fixed the build by reverting most of the changes from the breaking pull request. VeriCI identified the "minimal" problem, *i.e.* , the smallest set of commands that the build system was not able to handle. Since there may be multiple solutions to a build failure, VeriCI does not try to suggest a repair, but rather helps the user identify the root cause of a build failure. This way the users can address the problem in whichever way they think is most appropriate.

8.2.2 Identifying errors spanning multiple files

In the previous example, VeriCI detected the root cause of an error which required changing a single line. However, many CI errors can be a result of complex relationships between multiple files and branches in a repository. To demonstrate this, we take another real-world example, from the

activescaffold/active_scaffold [238] repository which has (at the time of writing) 5191 commits over 87 contributors, 987 stars, and 323 forks.

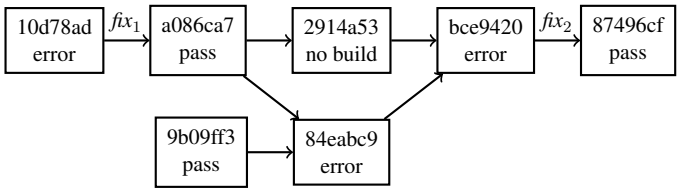


Figure 8.2: A GitHub history chain depicting dependencies between commits.

Fig. 8.2 shows a sequence of commits. Each commit is represented by a box and the arrows indicate the parent commit. A commit can have two child commits (as in commit `a086ca7`) when two users make a different change based on the same state of code. This difference was eventually resolved through a merge, resulting in a commit that has two parent commits (as in the case of commit `bce9420`).

To better understand these commits, the `git diff` command shows the difference between the current and previous commit. Fig. 8.3 depicts the `git diff` from a selection of commits from Fig. 8.2. Italics indicate which file was changed, while (+/-) indicate what was added or removed.

Listing 8.1: 10d78ad

version.rb

```
module Version
 MAJOR = 3
 MINOR = 4
- PATCH = 34
+ PATCH = 35
```

Listing 8.2: a086ca7

Gemfile.lock

```
- active_scaffold (3.4.34)
+ active_scaffold (3.4.35)
```

Listing 8.3: bce9420

```
- Gemfile.lock
+ Gemfile.rails − 4.0.x.lock
+ Gemfile.rails − 4.1.x.lock
+ active_scaffold (3.4.34)
```

Listing 8.4: 87496cf

Gemfile.rails − 4.0.x.lock,
Gemfile.rails − 4.1.x.lock

```
- active_scaffold (3.4.34)
+ active_scaffold (3.4.35)
```

Figure 8.3: The git diffs for a selection of the commits from Fig. 8.2.

In commit `10d78ad` in Fig. 8.2, the author upgraded the version number of the package from 3.4.34 to 3.4.35 in the `version.rb` file [238]. However, in order to correctly bump the version number in this project, the user needed to change the number in both the `version.rb` file, and

a `Gemfile.lock` file. Commit `a086ca7` shows how the user corrected this error so that the repository is again passing the build (cf. annotation fix_1 in Fig. 8.2).

At the same time, another user submitted a pull request with the old version which split the `Gemfile.lock` into two separate files for different versions of the rails library. In the merge process, the user inadvertently removed the original `Gemfile.lock` without copying over the fix for the version number. The merge resulted in the repository taking commit `bce9420` and is in a similar state to the previous broken commit `10d78ad`. There are now two files that are not consistent with `version.rb`.

By analyzing the previous data from fix_1 (depicted in Fig. 8.2), VeriCI has learned that the `PATCH` value in the `version.rb` is potentially problematic in this situation. Running VeriCI at the point of commit `bce9420` generates the error below:

```
Predicted build failure based on these keywords:
    - PATCH
    - simplecov
```

VeriCI predicted a build failure, and provides the explanation that this failure is likely due to one of the two listed error locations. We confirmed this error message as a true positive, as the next day the user discovered this fix on their own, and applied fix_2 to bring the repository back to a correct state [239].

8.3 Preliminaries

In this section we introduce the basic vocabulary in the continuous integration paradigm. We describe the formalism that we use to model the CI terms. This formalism is the basis for the future analysis and the learning process.

Repository Status. The core structures used in CI platforms are repositories. A repository, R contains all information required for a CI build, such as source code, automated tests, lists of library imports, build scripts, and similar files. The CI process monitors how a repository evolves over time, so we use R_t to denote the state of repository R at time t. While many version control systems allow for a branching timeline (called a *branch* in git), we linearize and discretize that timeline according to the build order in the CI tool, in accordance with the linearization of TravisTorrent [229]. This

way the time indexes are always non-negative, monotonically increasing integers. For a repository R_t, a typical CI tool usually has three possible outcomes:

1. All builds were successful and all tests pass
2. All builds were successful, but some tests fail
3. There was an error in the build process before tests could even be executed

Since the focus of this work is analysing the CI *configuration* of the repository, we distinguish only between successful builds and failing builds. If all builds were successful (cases 1 and 2), there is no error in the CI configuration and call the status of the repository "passing", and denote this build success with $S(R_t) = P$. In case 3, when a build failed, we call the status of the repository "failing" (or, a "failed build"), and we denote this with $S(R_t) = F$.

Of a particular interest are changes in the repositories that cause the build status to change, for example when the repository status changes from passing to failing. To capture this, we introduce the following notation:

$$S(R_{t,t+1}) = PF :\Leftrightarrow S(R_t) = P \wedge S(R_{t+1}) = F$$
$$S(R_{t,t+1}) = FP :\Leftrightarrow S(R_t) = F \wedge S(R_{t+1}) = P$$

Repository Summary. Our main algorithm for build prediction takes as input a dataset of feature vectors and labels. A feature vector is a set of tuples of the type $(\textit{String}, \mathbb{R})$, and the label is the build status (passing or failing). In order to map a repository R_t to a feature vector, we introduce the notation of a repository summary, $\hat{R}_t$. A repository summary, $\hat{R}_t$, is feature vector capturing an abstraction of that repository's CI configuration. We note that a CI configuration encapsulates both the CI configuration file (e.g. `.travis.yaml`), as well as some parts of the source code configurations (e.g. library imports and their versions). The process of extracting feature vectors that summarize the CI configuration is detailed in Sec. 8.4.

In the summarization process, we select only information relevant to the CI configuration. Every repository contains a number of files that are not relevant for deriving the properties about the CI configuration. Examples of these files include "readme" files, `.csv` files, or images. We filter out all

such files based on their extensions. For example, for Ruby programs we consider all **.rb*, *Gemfile*, *gemspec* files.

8.4 System Description

To statically check that a CI repository is correctly configured and the build will succeed, we must build a model of correctness. To do this, we analyze a number of existing CI repositories, including both the code and configuration files, and their corresponding CI build results. Once we learn this model of correctness, our system VeriCI can make a judgement on whether a new commit to the repository is likely to break the build. In the case that VeriCI predicts a build failure, VeriCI provides an explanation to the user for the prediction so that the user may proactively fix the build.

For each repository project, we consider a training set of commits from the repository, R. The process of learning the model is an iterative process. As new commits appear in the repository, they are integrated into a refined version of the model.

Existing efforts on prediction of build failures for CI rely on metadata in the learning process [240–242]. As these metrics are not tied to the code itself, approaches using metadata cannot provide the user with any information about which part of the code they should inspect to isolate and fix the root cause of the failure. In our work, in addition to providing build failure prediction, we also provide software developers with an error message describing the suspected cause of failure, so that they can proactively fix the CI configuration.

The goal of producing useful error messages creates two new challenges in this domain. First, we must use a learning strategy that not only classifies the build status, but also provides a justification for output of the classifier. This problem of justification is referred to as *interpretability* [243] and *explainable AI* [244] in machine learning. In order to generate useful justifications, we must restrict ourselves from using any kind of metadata in our learning process. Our goal is to produce a justification that guides users to the root cause of a build failure. As such, the learning process must rely on code feature extraction that only uses a set of features directly tied to the code.

An overview of our learning approach is shown in Fig. 8.4 and consists of three main components:

1. We introduce *Abstraction Based Relabeling* (ABR) as a novel method to reduce noise in the training set $\mathcal{TR}$ of past commits. The underlying dataset is noisy due to non-deterministic

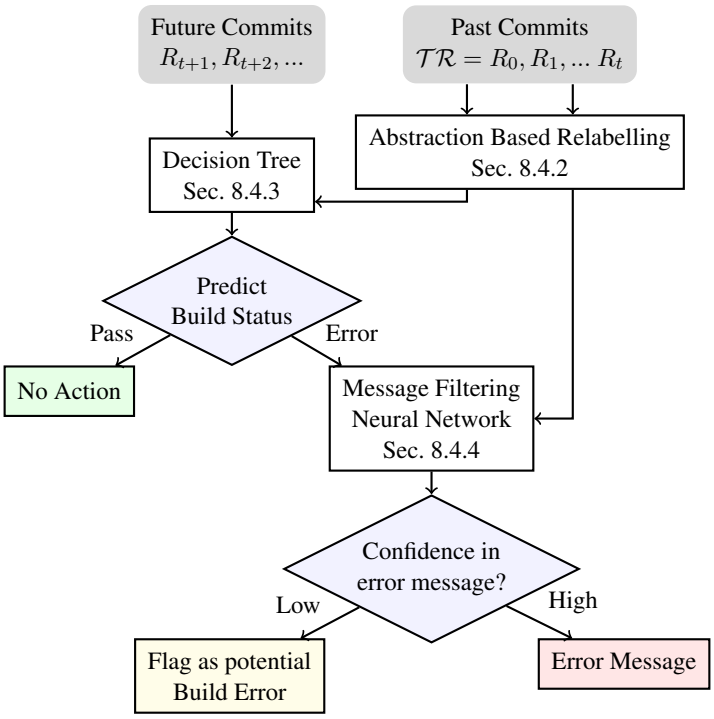


Figure 8.4: VeriCI trains a model to predict build failures on previous commits. If the model predicts a new commit will result in an failure, VeriCI tries to generate an error message. If VeriCI has confidence in the error message, it reports this to the user, and otherwise remains silent.

build outcomes introduced by errors out of the users' control (such as network failures, power outages, or even a hardware failure on the part of the CI provider). We call these types of errors *transient errors*. ABR ensures that if the extracted code feature vectors are the same, so are their build outcomes.

2. Given a pre-processed training set, we use a *decision tree* to construct a model that predicts whether or not a build will fail. In the case that the decision tree predicts a build failure, we generate an error message.

3. Constructing an appropriate error message is difficult without actually running the build. To increase the confidence in providing a helpful message, we train a neural network model that allows us to filter out cases of messages that are likely not contributing in resolving the build failure.

8.4.1 Feature Extraction

The first step in our system design is to pre-process the training set of a timeline of past commits. This list of commits form our labelled dataset, with each past commit $R_0, \ldots, R_t \in \mathcal{TR}$ being tied to a label denoting the corresponding build status, as introduced in Sec. 8.3. In order to leverage machine learning techniques, we must extract a selection of *code features* from these commits to form a summary of the repository state, $\hat{R}_t$ (also called a feature vector) to be used in the learning process.

We focus on extracting what we call *magic constant* code features. This feature tracks the use of hard-coded numerical constants in a repository. Magic constants are especially important in CI configurations as they often specify library version constraints. More generally, the core idea is if a line contains a scalar that we can embed into a feature vector, we build a tuple, where the first element is the context of the scalar, and the second element is the numerical constant value. We give an example of this process in Fig. 8.5.

Listing 8.5: R_1

```
import Tweet V1.0
m = rndMsg(1)
tweet(m)
```

Listing 8.6: R_2

```
import Tweet V2.0
m = rndMsg(1)
sendTweet(m)
```

Listing 8.7: $\hat{R}_1$

```
(import Tweet, 1.0)
(rndMsg, 1.0)
```

Listing 8.8: $\hat{R}_2$

```
(import Tweet, 2.0)
(rndMsg, 1.0)
```

Figure 8.5: Two examples for extracting code features (Listings 8.7 and 8.8) from repository code (Listings 8.5 and 8.6 respectively) at time points 1 and 2 of a commit history.

The number of potential code features when analyzing a repository is intractably large (a complete feature extraction would encode all behavior of the code), thus we must reduce the dimensionality of the code by selecting key features. As with many feature extraction tasks, our code feature extraction uses a set of templates built by an expert, a process known as feature engineering [245]. Automated code feature extraction was studied in [246], but it was in the context of C-like languages. We investigated reusing their techniques, but it was not a good match because their language domain has much more structure than CI configurations.

8.4.2 Abstraction Based Relabeling

During this feature extraction process we may face cases where we end up with two repository summaries that are identical, although the build statuses differ. This can happen for two reasons. First, it is possible that our feature extraction has abstracted too much of the code and we have lost the key difference between a build that succeeds and a build that fails. Second, we must account for the fact that CI builds are not pure in the sense that some builds can fail due to transient errors. This non-determinism of configuration behavior has also been observed in other settings of configuration analysis [234]. Transient errors are problematic as having identical data points with different labels injects noise into the training set. In order to reduce the noise that our learning module needs to handle, we want to ensure that if extracted code feature vectors are the same, their build outcomes are also the same. Restated using our formalism, $\hat{R}_t = \hat{R}_{t'} \Rightarrow S_{ABR}(R_t) = S_{ABR}(R_{t'})$.

We introduce the notion of *Abstraction Based Relabeling* (ABR) to account for the impact of CI build non-determinism and information loss during feature extraction. The goal of ABR is to relabel the training set so that it is self-consistent with our observation (feature extraction) of the training set. In ABR, we examine each repository summary and if at some point the build status changes, but the code features do not, we relabel that failing status as a passing state. Put formally, the relabeling from the original build status $S(R_t)$ to the ABR build status $S_{ABR}(R_t)$, works as follows:

$$\text{If } S(R_{t,t+1}) = PF \wedge \hat{R}_t = \hat{R}_{t+1} \Rightarrow S_{ABR}(R_{t+1}) = P$$
$$\text{If } S(R_{t,t+1}) = FP \wedge \hat{R}_t = \hat{R}_{t+1} \Rightarrow S_{ABR}(R_t) = P$$

To understand the effect of ABR, we provide an example process in Table 8.1. In this example, the build of $S(R_2) = F$ failed (for the sake of demonstration, due to a network failure), but we can see that $S(R_3) = P$ succeeded. During the learning process we do not have information on the root cause of these failures, but we can observe the code features and CI status, so we relabel $S(R_2)$ accordingly.

Table 8.1: An example learning process demonstrating ABR. Source code for $\hat{R}_1$ and $\hat{R}_2$ are listed in Fig. 8.5.

Repository States	$\hat{R}_1$	$\hat{R}_2$	$\hat{R}_3$	$\hat{R}_4$
Original Status : $S(R_t)$	P	F	P	F
Code Features Extraction				
$\hat{R}$				
import Tweet	1.0	2.0	2.0	2.0
rndMsg	1.0	1.0	1.0	2.0
ABR Status : $S_{ABR}(R_t)$	P	P	P	F

8.4.3 Predicting Build Status

After we preprocess the training set with ABR, we build a decision tree classifier to analyze a new commit to a repository and predict the build status. Decision tree learning [247] is a widely used machine learning technique in which we construct a tree consisting of two types of nodes: decision nodes and leaf nodes. Decision nodes contain a Boolean condition, which defines branching depending on if the condition is true or false. The leaf nodes represent the final classification of a data point. For a data point belonging to some leaf node in a decision tree, we extract a *decision path* that describes which decision nodes were used to arrive at that leaf node. This decision path is constructed by recording the traversal of the decision tree - that is, tracking the conditions in decision nodes (or their negations) on the path from the tree root to that leaf. The ability of a decision tree to easily provide a justification for its classification is the key property of decision trees as a machine learning technique that makes it a good fit for our purposes.

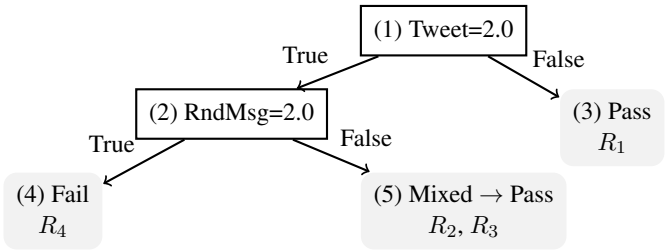


Figure 8.6: An example of a decision tree with white boxes as decision nodes and gray boxes as leaf nodes.

Fig. 8.6 depicts a decision tree generated from the results of ABR shown in Table 8.1. Our tree needs to classify four commits, based on the given two features (`Tweet` and `RndMsg`). In this

example, the learning process results in a decision tree with three leaf nodes. Node (3) contains only $\hat{R}_1$, correctly classifying it as a passing node. Node (4) contains only $\hat{R}_4$ and it is correctly classified as a failing node. We call nodes (3) and (4) pure nodes, since they perfectly predict the commits from the training set.

In contrast, node (5) is a mixed node, since repositories from the training set that are classified by this node are both passing ($S(R_2) = F$) and failing ($S(R_3) = P$). Although these repositories had different build statuses, based on the feature vectors we have extracted, they are equivalent - and thus had equivalent ABR statuses. This leaf node cannot then be split further based on the available features and we cannot correctly classify these repositories.

The reason why we are using decision trees as our classification learning technique to predict build status is because we can also easily extract root causes of the failed build. To do this, we simply need to trace the path traversed through the tree to classify a data point. As an example, we can construct an error message for the classification of R_4 by tracing the decision path and generating the error message: (`Tweet = 2.0` $\wedge$ `Rndmsg = 2.0`).

8.4.4 Filtering Error Messages

VeriCI predicts the build status of a new commit, and extracts a potential error message using the decision tree classification path. This is a key benefit to our technique, as these error messages allow VeriCI to not only pre-emptively tell users the status of their build, but also give them feedback on how to fix it. However, not all our error messages are informative - sometimes we predict the correct status for the wrong reason. An incorrect error message (even if the build is failing) is potentially worse than no error message at all. If we report an incorrect error message, users may waste time searching in the wrong location for the root cause of the build error. However, if we can refrain from reporting bad error messages, even when the build does fail, the user workflow remains at the status quo as it is when not using VeriCI.

Thus, to ensure we minimize the number of misleading error messages, we trained a neural network to predict the effectiveness of our generated error messages. We measure the effectiveness of the error message by checking if the error message we report is contained within the `git diff` of the commit that eventually fixes the build. If any part of the decision tree path for the failing

commit appears as a substring of the fix, we label the error message as Hit, otherwise we label the error message as a Miss.

To label our training set, we “learn from our mistakes" by looking at the error messages we have previously reported. In labeling our training set for the neural network there are two cases we must consider. First, we may have previously reported an error, and have already seen the fixing commit for this failing build. In this case, we can label out error message by checking the ground truth. Second, we may have previously reported an error, but not yet seen a fixing commit for the failing build. In this case, we will construct a label instead from the breaking commit, e.g., the `git diff` between this failing build and the last previous build that succeeded.

In order to encode the error message as a feature vector that can be understood by the neural network, we convert the text to its ASCII representation. As error messages can differ in length, we pad all error messages shorter than a fixed feature vector size with spaces, and truncate all error messages longer than this size. We then use a technique called *feature scaling* to ensure all features (characters of the error message) are represented by a float value 0-1. This is an important step, as neural networks are sensitive to the scale of feature values, whereas decision trees are not. Concretely, we used a multi-layer perceptron neural network with two hidden layers. Finally, to integrate the neural network with the rest of VeriCI, we use the resulting model as a binary classifier. If the neural network predicts an error message to be a Hit, we report both the predicted build status and its corresponding error messages to the user. However, if the neural network predicts a Miss, we only report the predicted build status to the user.

8.5 Evaluation

We implemented VeriCI to statically check repositories that use the open-source continuous integration testing tool, TravisCI. TravisCI is an ideal testbed for analyzing CI builds, as the tool is free for open-source projects and widely used on GitHub. Additionally, a log history of the builds for a number of large, active, and open-source project has been made available through the TravisTorrent dataset [229]. This dataset directly provides us with the labeled, temporally ordered commit data over many repositories. Our evaluation set consists of all repositories, available in TravisTorrent, which have 150 - 200 commits, and whose main language is Ruby (the 35 repositories listed in

Table 8.2). As we require many commits as training data, we wanted to take repositories with the largest number of commits possible. Taking repositories with a commit history of larger than 200 commits slowed down the training process, due to limited network speed and memory needed to clone the repositories.

In the evaluation, our goal is to answer the following questions:

1. Does VeriCI accurately predict the build status of a commit before the build actually executes?
2. Does VeriCI report error messages that correctly identify the root cause of the build failure?
3. Are the error messages VeriCI generates helpful to users?

8.5.1 Accuracy of Prediction

In order to evaluate the accuracy of VeriCI, our evaluation models the way VeriCI would be used in a real world setting. In practice, VeriCI would be trained over a set of commits up until the present moment, and used to predict commits for one day. Then, at the end of the day, VeriCI would build a new model, incorporating the information from that day, so that the model the next day can learn from the new commits and any mistakes that were made. We show this on a timeline in Fig. 8.7.

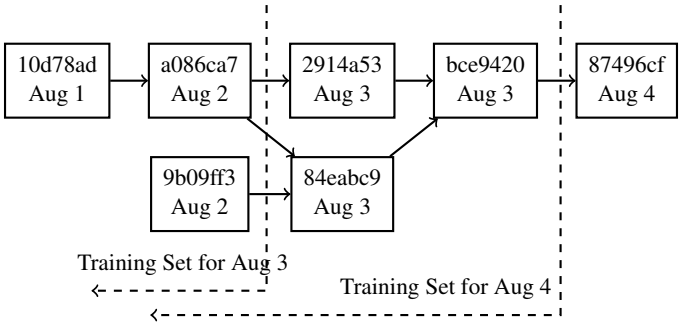


Figure 8.7: VeriCI rebuilds a model at the end of every day to provide better prediction the next day.

The strategy we employed in the evaluation is presented in Table 8.2. For a repository with n total commits, we build the first model with $n/2$ commits, and use that model to evaluate commit $R_{(n/2)+1}$. We then rebuilt the model with a training set of $(n/2) + 1$ commits and used that model to predict the status of the next commit. Using this approach, we found that VeriCI can predict build status with an overall accuracy of 91.0%. This is competitive with other tools for build status prediction

Table 8.2: Classification rates across 150 - 200 commits of various repositories including TP (True Positive - correctly predicted failure), TN (True Negative), FP (False Positive - incorrect predicted failure when actual status was pass), and FN (False Negative).

Repository Name	Accuracy	TP	TN	FP	FN
sferik/rails_admin	93%	14	75	1	6
thoughtbot/shoulda-matchers	98%	2	80	0	2
jnunemaker/flipper	87%	17	68	3	10
pagseguro/ruby	91%	6	80	2	7
activescaffold/active_scaffold	89%	7	51	2	5
twitter/twitter-cldr-rb	92%	20	63	4	3
pivotal/LicenseFinder	96%	9	71	1	2
padrino/padrino-framework	90%	0	27	1	2
toptal/chewy	87%	1	38	2	4
eapache/starscope	100%	0	20	0	0
thoughtbot/clearance	85%	0	70	0	12
spree-contrib/spree_i18n	90%	1	44	0	5
johnbellone/consul-cookbook	95%	17	55	1	3
rspec/rspec-expectations	96%	0	24	1	0
RubyMoney/money	100%	0	15	0	0
rabbit-shocker/rabbit	85%	0	83	0	15
lotus/lotus	92%	0	23	0	2
sensu/sensu-puppet	90%	0	9	0	1
orientation/orientation	82%	2	47	2	9
rspec/rspec-mocks	95%	0	19	0	1
weppos/whois	95%	2	50	1	2
benbalter/gman	84%	3	76	0	15
rspec/rspec-rails	92%	0	55	0	5
ashmckenzie/percheron	95%	0	38	1	1
twilio/twilio-ruby	96%	0	24	1	0
heroku/heroku-buildpack-ruby	90%	0	72	1	7
asciidoctor/asciidoctor	73%	0	11	2	2
ai/autoprefixer-rails	88%	1	21	1	2
reactjs/react-rails	100%	0	10	0	0
asciinema/asciinema.org	100%	0	5	0	0
concerto/concerto	93%	1	41	2	1
railsbridge/docs	100%	0	15	0	0
pothibo/ecrire	100%	0	10	0	0
prawnpdf/prawn	100%	0	15	0	0
plataformatec/devise	95%	0	62	0	3
Average	91.0%	2.943	41.914	0.829	3.629

based on metadata such as the committer's historical rate of correct commits, the size of the commit, sentiment analysis on the commit message, use of emojis, and similar metadata [240–242, 248].

One interesting note is that we should always expect our prediction to have a non-zero number of false positives (predicting pass when $S(R_t) = F$). A recent study investigated the reliability of TravisCI builds [249], and found that roughly 9% of builds have a misleading or incorrect

outcome. These incorrect outcomes manifest from transient errors outside the code base, such as in dependencies or through network failures.

8.5.2 Accuracy of Error Messages

Table 8.3: Accuracy of neural network postprocessing on error messages. N/A indicates we do not report a message to the user.

Prediction	Ground Truth	Number of Occurrences
Hit	Hit	15
Hit	Miss	2
Miss	Hit	5
Miss	Miss	40
Prediction	**Reason for N/A**	**Number of Occurrences**
N/A	Non-Alphanumeric	1
N/A	Insufficient Training Data	24
N/A	No Fixing Commit	16

Table 8.4: Effect of ABR on VeriCI performance.

ABR	Accuracy	Hit Rate	Average Depth
OFF	90.50%	5/10	2.31
ON	90.96%	15/20	1.99

We have shown that VeriCI can predict build status with very high accuracy, but the key innovation in this work is that we also provide an explanation for the classification that our model provides. However, we would specifically like the explanation provided to correspond to a potential change in the code base that could fix the failing build status. To evaluate the rate at which VeriCI provides accurate error messages, we check if any of the keywords we presented in the error message appear in the difference between the failing commit and the next passing commit. If the user has changed a keyword that we suggested in order to fix the repository's build status, it can be seen as evidence that we suggested a correct root cause of the failure.

Using this metric, we found that over the repositories listed in Table 8.2, the error we reported corresponded to the change that the user eventually made to fix the build 88% (15/17) of the time, for the rules we learned. We report the full effect of the error messaging filtering technique described in Sec. 8.4.4 in Table 8.3.

In Table 8.4, we show the results of running VeriCI with and without ABR to measure the impact of this new method. We find that enabling ABR has a positive impact on the accuracy of our error messages. The average number of keywords reported in an error message (the decision path depth) by VeriCI, with all optimizations, was 1.99. This means that 88% of the time we correctly identified a critical keyword in a breaking commit with an average of only 1.99 guesses.

8.5.3 User Study

In Sec 8.5.2, we demonstrate the error messages VeriCI produces are accurate, but this still leaves the question on how useful these messages are to developers. To assess the impact of VeriCI on how developers identify and fix CI build failures, we conducted a user study with a total of 20 professional developers. Our goal was to investigate whether the use of VeriCI increases the correctness of solutions to errors in CI, compared to solely the output of TravisCI. The null hypothesis and alternative hypothesis corresponding to this question can be formulated as follows:

- $H1_0$: Error messages generated by VeriCI *do not* increase the user's ability to correctly find solutions to CI errors.
- H1: Error messages generated by VeriCI increase the user's ability to correctly find solutions to CI errors.

Tasks & Environment. We asked participants to fix two TravisCI build failures from a GitHub repository [250, 251]. Given the failed builds, participants were asked to identify the root cause of the failure and describe a possible fix. Our user study was designed as a between-subjects study. In the control group, participants were required to fix builds with the log file provided by TravisCI itself. In the treatment group, we provided the error messages generated from VeriCI. In both groups, participants had no prior knowledge of the code base and were told to make use of any resources available to them (including internet search), except looking at future commits in the repository of the study. This setting was designed to, as closely as possible, model the information available to developers in real-world CI use cases. Users were asked to identify the root cause and propose a fix in an open response field.

Variables & Analysis. The independent variable in our experiment is the presence of error messages by VeriCI during the tasks. The dependent variable is the correctness of the given answers. To

Table 8.5: Statistical results related to the correctness of how developers identify and fix CI build failures

Task One	**Exact Fix**	**Partial Fix**	**Not Related**
Control Group	1	0	9
Treatment Group	2	4	4
Task Two	**Exact Fix**	**Partial Fix**	**Not Related**
Control Group	6	2	2
Treatment Group	4	4	2

convert the participants' answers into quantitative information for further analysis, we designed a rubric to classify answers into one of three categories (following the methodology by similar user studies in software engineering research [252, 253]). We evaluate participants' results by comparing their answers to the actual subjects' fix provided by the TravisCI users.

1. The participant identifies the root cause or describes a fix with enough detail to directly resolve the build failure.

2. The participant mentions part of the root cause or describes a partial fix, but does not provide enough detail to completely resolve the failure. We use a similar, but more relaxed, criteria for correctness here compared to Sec. 8.5.2 - the user must only mention a substring of any possible fix (not just the eventual fix in this repository).

3. The participant incorrectly identifies the root cause and proposes an incorrect fix.

Subjects. Since our study setting requires knowledge on how to identify and fix production build errors, we recruited 20 subjects with professional software development experience to participate in our study, 10 for both control and treatment conditions. We also collected the subjects' levels of expertise on different topics (continuous integration, TravisCI, Ruby, and GitHub) through a self-reported assessment on a Likert-scale. We performed a Mann-Whitney U test [254] to assess whether both groups' expertise is comparable. The results did not show any significant differences between expertise levels in the groups (results were overall not significant at $p < .05$ with critical value of $U > 27$.

Results. We asked each participant to fix two failed TravisCI builds. We report the distribution of their coded answers into Table 8.5. We combined the category one and two into a new category

of category "at least partially correct" and use Barnard's test[2] for significance. We found strong evidence ($p = 0.0227$) that rejects the null hypothesis (i.e. we showed that VeriCI helped participants to correctly identify the root cause or find a fix for the build failure). This experiment strongly indicates that VeriCI successfully helped users to find the root cause of the failed build or even a fix. For the second task, we found no significant differences in both control and treatment conditions. Of the participants, 80% (eight out of ten) achieved at least a partially correct solution.

We examined the second task more carefully to form a hypothesis for this inconclusive result. When checking the Travis log file of this specific build, we found that the root cause of this build failure was already contained in the Travis log file. This observation leads us to the theory that if the root cause is easily extractable from the log file, the generated error message by VeriCI does not yield any significant differences to find the fix.

8.6 Discussion

We discuss the challenges of building learning-based static analyzers, and how we mitigate possible threats to the validity of our evaluation.

8.6.1 Developer Use in Learning-based Analyzers

We envision VeriCI being used by a developer, or team of developers, in their everyday CI workflow.

Uncertainty and Explainability. In deploying probabilistic program analysis tools, one of the most important metrics is that the tool has a very low false positive rate. In the context of VeriCI, it quantifies the number of times developers are falsely led to believe there is a build failure that they need to investigate. Often in practice, this can be the single metric that prevents adoption, as users will quickly ignore tools with too many false alarms. Our evaluation results (Table 8.2) have shown that the average number of false positive reports to average number of correct classifications are at only 1.8% (FP/TP+TN). For the remaining uncertainty, our use of decision trees for learning provides developers with a comprehensible set of rules (see Fig. 8.6) that led to the classification. This allows developers to reason about the outcome more quickly and disqualify false positives and continue to run the build on the server.

2. Barnard's test is a more conservative version of Chi-Square specialized to small values [255].

Table 8.6: Training time for VeriCI over a single repository with variable number of commits (training set size).

Num of Commits	1	20	40	60	80
Training Times (s)	0.049	1.067	2.887	4.824	7.623

We hypothesize that, as the use of VeriCI to check a CI build would occur much less frequently than, for example compiler errors, users may have a relatively higher tolerance for false positives. However, such a statement would require a user study, and we leave such explorations to future work.

Scalability. We find that VeriCI scales roughly linearly (dependent on the size and complexity of the repositories in the training set) and list training times in Table 8.6. All the experiments in this section are conducted on a MacBook Pro equipped with Haswell Quad Core i7-4870HQ 2.5 GHz CPU, 16GB memory, and PCIe-based 512 GB SSD hard drive. Although our results with the repositories in Table 8.2 already provide 91.0% accuracy of prediction and 88% hit rate for correct error messages, it is generally the nature of machine learning to return better results with more data, which requires fast training times. As a point of implementation, decreasing the constant factor cost of VeriCI could improve running times so that we can scale to a truly large scale, and leverage the thousands of repositories with a total of more than 2.6 million commits available in the TravisTorrent dataset [229]. We leave this to future work, noting that this first version of VeriCI focused entirely on correctness of implementation and not on any optimizations.

8.6.2 Negative Results

Adding more features does not necessarily lead to better results. In order to refine the decision trees at points of mixed nodes, a natural next step would be to imagine a way to extract new features, and re-run ABR. To test this idea, we also implemented an abstraction refinement loop that iteratively added new features to the feature vector of the repository summaries. We extracted a *code diff feature*, that encoded information on the key lines that changed between commits in the feature vector. However, in our evaluation this approach faced two issues. First, most build failures in practice involve constants we can extract from the code with our initial abstraction, and adding further features actually decreases the accuracy of prediction. Second, a more complex feature extraction algorithm can be computationally expensive. While our initial abstraction of our evaluation is completed in roughly one hour, with the complex feature extraction, evaluation took roughly days.

8.6.3 Threats to Validity

We briefly revisit potential limitations of our study concerning threats to validity.

Internal Validity.. There is a possibility that our learned model is not the most ideal within the space of potential hypotheses that classify build outcomes. In particular, there may be more effective ways to do code feature extraction. To this end, we also experimented with incorporating more code features via a refinement loop (c.f. Sec 8.6.2), but did not find any benefit with this approach. This is an additional indication that our chosen model and feature extraction strategy is able to properly capture the complexities of successful and failed builds. Overall, given our high accuracy and low false positive rate, we are confident that we have found a sufficiently good model and leave optimization on this for future exploration. Threats in the user study relate to the chosen study population and tasks. We opted to recruit participants with professional software development experience to suit the complex nature of our tasks (identifying and fixing production build errors in CI). While developers dealing with these issues are well-versed with the respective code base, our participants did not have any familiarity. We took this constraint into consideration when designing the study tasks and environment. The tasks and their complexity were carefully chosen from an active open source project that exposes challenging CI build errors that as closely as possible reflect the practical circumstances, but at the same time are not too specific to the underlying code base.

External Validity.. Our evaluation samples from open-source projects using Ruby with TravisCI. There is a possibility that this setting does not generalize to other contexts, such as repositories that focus on different core programming languages. However, the features that we use for learning are agnostic to the specific languages and technologies used. We intended for our *magic constant* feature extraction to be easily applied to different programming and configuration languages. To test this, we ran VeriCI on 22 Java repositories (selected using the same criteria described in Sec. 8.5) and found VeriCI to have 90.6% predication accuracy, but leave a more through investigation to future work. In terms of generalizing to other CI frameworks beyond Travis, we note that the information required from TravisCI to build our model is also available in other continuous integration frameworks.

Construct Validity.. Our measurement of the correctness of our error messages by checking for the use of overlap in fixing commits is only a rough proxy measure of true positives. This metric over-approximates our success in that it is possible that although the user changed a keyword we

flagged, that keyword was not in fact very important. At the same time, this metric also under-approximates our success int that there may be multiple correct fixes for a problem, and while we suggested a correct one, the user decided to take another fix. Overall the average path depth of VeriCI is an encouraging result that shows that it is indeed possible to pinpoint the root cause of a CI misconfiguration from a learning approach.

8.7 Related Work

Configuration verification and validation has been considered as a promising way to tackling software failures resulting from misconfiguration [164]. Nevertheless, the strategy for generating models and checking configuration settings still remains an open problem.

Continuous Integration Build Prediction. The increasing prevalence of CI as a core software development tool has inspired significant work on the topic. Some work has included predicting the status of a commit based on metadata such as the previous commit and history of the committer [240–242]. Natural language processing and sentiment analysis has also been used to predict build status [248]. However neither of these approaches provide the user with information that they are able to change that could actually fix the build. For example, if a user pushes a commit with a commit message that is "negative" (*e.g.* , "annoying hack to get javascript working"), changing the commit message cannot change the build status.

In contrast, VeriCI specifically predicts the build status based on direct code features that users have control over, so that in addition to a predicted build status, users can to address the issues with the justification of the classification that VeriCI provides.

Learning-based configuration verification. Several machine learning based misconfiguration detection efforts also have been proposed [176, 178, 256–259]. EnCore's [176] learning process is guided by a set of predefined rule templates that enforce learning to focus on patterns of interest. In this way, EnCore filters out irrelevant information and reduces false positives; moreover, the templates are able to express system environment information that other machine learning techniques (*e.g.* , [178]) cannot handle.

Compared to prior work, VeriCI targets configuration files for CI, which is a fundamentally different task from the above efforts. EnCore, ConfigC, and ConfigV are proposed to verify configuration

files for database system setup. Additionally, because these systems target configuration files for database systems, their model is specific to the key-value assignment representation, which is a more structured schema representation then CI configurations.

Language-support misconfiguration checking. There have been several language-based efforts proposed for specifying the correctness of system-wide configurations. For example, in the datacenter network management field, the network administrators often produce configuration errors in their routing configuration files. PRESTO [208] automates the generation of device-native configurations with configlets in a template language. Loo *et al.* [209] adopt Datalog to reason about routing protocols in a declarative fashion. COOLAID [210] constructs a language abstraction to describe domain knowledge about network devices and services for convenient network management. In software configuration checking area, Huang *et al.* [177] proposed a specification language, ConfValley, for the administrators to write their specifications, thus validating whether the given configuration files meet administrators' written specifications. Compared with the above efforts, VeriCI focuses on the configuration files used for the software building process. In contrast, an important aspect of our work is to automate configuration verification process. The above efforts can only offer language representations and still require the administrators to manually write specifications, which is an error-prone and labor-intensive process.

8.8 Conclusions

This chapter has presented our tool, VeriCI, that automatically checks for errors in CI configurations before the build process. Driven by the insight that repositories in CI environments are already labeled with build status histories, our approach automatically generates specifications for correct CI configurations. We evaluated VeriCI on real world data from GitHub and find that we have 91.0% accuracy of predicting a build failure. We also ran a user study that shows VeriCI helps users to more accurately identify build issues, but we still require further investigation to understand the situations in which VeriCI is the most beneficial.

Part IV ❖ Synthesis for Impact

At the crux of the development of new synthesis techniques is the goal of having a positive impact on the developer experience. Achieving such a goal necessitates that we thoughtfully engage with developers and ask critical questions about the role of synthesis in typical development workflows. In this part of the thesis, we first look at programming by example in the domain of shell scripting. We find that common measurements of usability and impact of synthesis tools (i.e. time taken to complete tasks) do not tell a complete story of the user experience.

On a more meta level, we look at how research itself in synthesis is a powerful tool of engaging with early stage researchers and encouraging the development of a computing identity. We present a mentoring framework designed to help introduce and integrate new students to computer science research through projects on synthesis.

Chapter 9

Programming by Example: Efficient, but Not "Helpful"

Work presented in this chapter was completed in collaboration with Drew Goldman, Allison Wesley, and Ruzica Piskac. Sections of this work have been previously published [22], and are reproduced here, at times in their original form.

Programming by example (PBE) is a powerful programming paradigm based on example driven synthesis. Users can provide examples, and a tool automatically constructs a program that satisfies the examples. To investigate the impact of PBE on real-world users, we built a study around StriSynth, a tool for shell scripting by example, and recruited 27 working IT professionals to participate. In our study we asked the users to complete three tasks with StriSynth, and the same three tasks with PowerShell, a traditional scripting language. We found that, although our participants completed the tasks more quickly with StriSynth, they reported that they believed PowerShell to be a more helpful tool.

9.1 Introduction

Scripting languages, such as PowerShell and bash, help IT professionals to more efficiently complete tedious and repetitive tasks. Those tasks can include file manipulations and organizing data, where a simple error can destroy users' data. As an example, consider the disastrous attempt to remove all backup emacs files with the command `rm * ~`. Additionally, small errors in the scripts can lead

to malicious behavior, such as data loss [260]. Scripts can be difficult for users to write by hand, requiring users to have extensive experience with regular expressions, programming and domain expertise in the scripting language of their choice. Depending on the application, a user may need to be able to write a very complicated regular expression for a relatively simple task. Furthermore, users may not have access to their scripting language of choice, depending on the operating system and software policies used by their employer.

For these reasons, many end-users search for help on online forums when they need to write a script [261–263]. When users seek help in writing a script on forums, they will often provide a few illustrative examples that convey the goal of the script. This observation was the basis of StriSynth [264], a research tool that was proposed to make scripting easier and more efficient by allowing users to program scripts by example. While scripting is a challenging task, especially for novice programmers, providing examples of the intended behavior is a more natural interface for scripting. StriSynth supports various types of functions, such as transformations, filters, partitions, and merging strings.

In this work, we explore how scripting by example, specifically with StriSynth, is received by the real-world target end-users. We designed a user study around StriSynth and recruited 27 IT professionals to participate in the study. In our study we asked users to complete three tasks with StriSynth, and the same three tasks with PowerShell, a traditional scripting language. When using StriSynth, users were statistically significantly faster at completing tasks as compared with PowerShell. However, in a post-study survey when users were asked which tool they perceived to be more "helpful", users statistically significantly reported that PowerShell, with the traditional scripting paradigm, was more helpful. This was a counter-intuitive result, as we expected that a faster should be considered to be more helpful by users. While the formal methods community has largely taken efficiency of task completion to be an indicator of a good language design, we explore our results here that show this is in fact a more complex issue.

9.2 Background

Programming by example (PBE) [2, 3, 121, 122] is a form of program synthesis. It works by automatically generating programs that coincide with the given examples. In this way, the examples

can be seen as an incomplete, but easily readable and understandable specification. However, even if the synthesized program satisfies all the provided examples, it might not correspond to user's intentions, due to this incompleteness in the specification. In this case, a user must provide further examples to the synthesis tool.

To address this issue, StriSynth was implemented as a live programming environment [265] for PBE. In this way, a synthesized script can be refined with every new provided example, and thus yields more interactive experience for the user. Interactive PBE allows end-users to provide a single example at a time, rather than guessing at the full example set that is necessary for synthesis.

In order to compare the PBE paradigm to more traditional scripting languages, we have chosen to use the tool StriSynth [264]. StriSynth is an existing tool for automating file manipulation tasks, in a similar style to Flash Fill's [266] synthesis of spreadsheet manipulations. While the use of scripting language such as sed, awk, Bash or PowerShell requires a certain level of expertise, many tasks can be easily described using natural language or through examples.

9.2.1 StriSynth example

To give some context for how StriSynth compares to traditional scripting language paradigms, we give an example task that can be easily completed with StriSynth. This task comes from a StackOverflow post, where the users discuss challenging regular expressions [261]. The user asked for a script that will create a link from every item in a directory. To better illustrate the goal of the script, the user provided two examples transformations:

```
Document1.docx      ⇨    <A HREF='Document1.docx'>Document1</A>
Document2.docx           <A HREF='Document2.docx'>Document2</A>
```

To accomplish this transformation, other users on the forum suggested a solution based on regular expressions in `sed`:

```
sed/\(^[a-zA-Z0-9]+\)\.\([a-z]+\)/\<a href\=\'\1\.\2\' \>\1\<\/a\>/g
```

While it was very easy for the user to express the goal of the script by providing examples, the resulting script is arguably less readable, even for such a simple problem. In contrast, to solve this problem in StriSynth, a user provides an example showing what a script should do:

```
> NEW
> "f.docx" ==> "<a href='f.docx'>a</a>"
> val F = TRANSFORM
```

The keyword `NEW` denotes the start for learning of a new script, after which the user provides an example of the scripts desired behavior. Based on the provided example, StriSynth learns a string transformer, and the user saves it with the next command. Every learned function can be saved using the command `val name = ...` which creates a reference, `name`, to the learned script. The user may then check how F works on different examples to confirm the learned function is correct.

```
> F("Document1.docx")
<A HREF='Document1.docx'>Document1</A>
> F("Document2.docx")
<A HREF='Document2.docx'>Document2</A>
```

We observe that the learned transformer F is a function that exactly does what the user asked initially. However, it only takes a single string as input, while the user wanted a script that operates on a list of strings. To extend the learned transformer to work over a list, the user can use the `as map` function.

```
> val finalScript = F as map
```

If a function G has a type signature $G : T_1 \rightarrow T_2$, then applying the postfix operator `as map` will result in G `as map` : List(T_1) $\rightarrow$ List(T_2). With `as map`, the user creates the final script which takes as input a list of file names and creates a list of HTML links.

Beyond the string transformation used above, StriSynth can also learn other types of functions from examples. StriSynth supports a *filter* function that takes a list of strings as input and removes some elements based on the filtering criterion. Similarly, StriSynth also supports learning a *partition* function takes as input a list of strings, and divides them into groups based on the partitioning criterion. Those groups are then returned as a list of lists of strings. This functions can be used in any way by the user, but are particularly useful for scripting tasks that require operations on certain types of files, or files matching some naming pattern.

In addition, StriSynth can learn a *reduce* function that merges the elements in a list into a single string. StriSynth's *split* function does the opposite: it returns a list of strings from the input string. These types of functions are especially useful for scripting tasks that apply operations to collections of files.

9.3 Methodology

A recent survey of the key challenges facing formal methods cites the need for more user studies, especially on real-world users [267]. To test the impact PBE on real users, we recruited 27 IT professionals, all of whom were 18 years of age or older. All materials for the study, as well as the raw data results from the study are available open source at `https://github.com/santolucito/StriSynthStudy`.

Our study design consisted of four stages:

1. A tutorial on both PowerShell and StriSynth that introduced the paradigm and syntax
2. Complete three scripting tasks (*Extract filenames* from a directory listing, *Move files* with `*.png` to `imgs/`, *Printing pdfs* from a list of various file types) in PowerShell
3. Complete the same three scripting tasks in StriSynth
4. A post-study survey

In the study, participants were told that they would be using the tools StriSynthA and StriSynthB instead of StriSynth and PowerShell to avoid bias from participants' prior experience. The participants were randomly split into two groups, group A and group B, where the two groups switched the order of steps 2 and 3 of the study to account for any potential bias in earlier exposure to the tasks. Group A completed the tasks with PowerShell first (N=12) and group B completed the tasks with StriSynth first (N=15). The entire study generally took each participant 50 minutes, and the study was conducted in-person with a researcher present. The scripting tasks were completed on the researcher's laptop, which was preloaded before each study with directories and files needed for the scripting tasks.

While each user was participating in the study, the researcher present recorded the overall time that was used to complete each task. Following the completion of the six tasks, each user was given

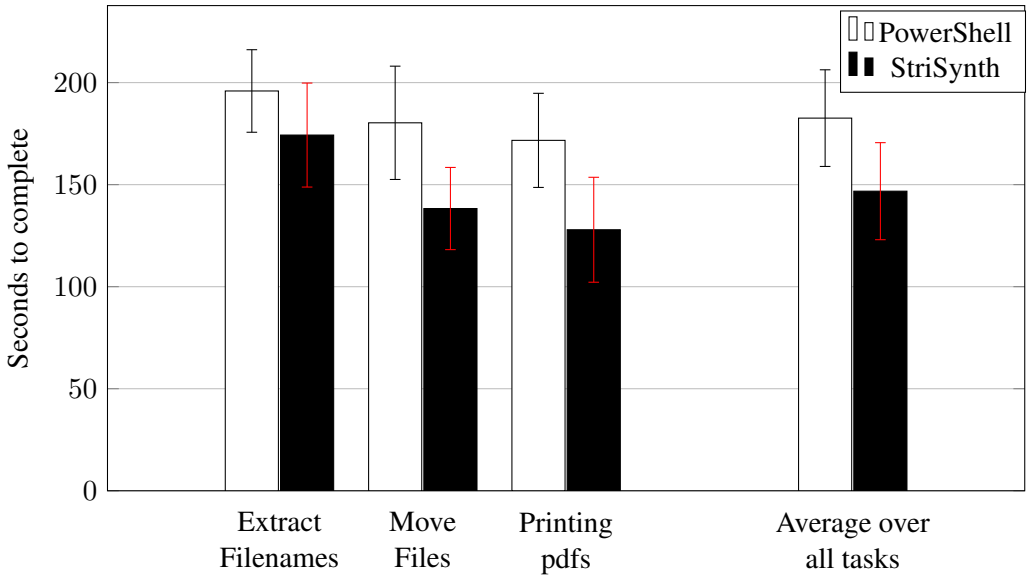


Figure 9.1: The amount of the time each task took, as well as the average time over all tasks for all users (N=27). The smaller bars indicate standard error.

a questionnaire. The questionnaire measured various responses: prior coding experience, perceived helpfulness of each program as a whole, and perceived helpfulness of each program for each specific task they completed.

9.4 Results

In this section we present the results of the user study described in Sec. 9.3. Overall, users completed the tasks more quickly when using StriSynth as opposed to PowerShell. This is good evidence that StriSynth is an efficient tool, especially as none of our users had used StriSynth before this study, while some already had experience with PowerShell. However, despite this concrete measure of efficiency for StriSynth, users said that they believe that PowerShell is a more helpful tool.

9.4.1 Time to complete the user study tasks

To estimate the usefulness of the programming by example tool StriSynth, we recorded the time it took for users to complete each task with both StriSynth and PowerShell. The results are shown in Fig. 9.1. In addition, Fig. 9.1 also contains standard error, depicted with line bars.

In the case of the first task (extracting filenames), from in Fig. 9.1 the standard error bars give us the intuition that true mean of the time it takes for overall population to complete this task using PowerShell is between 170 and 210 seconds. The smaller the standard error, the more likely is that we have achieved the exact, true value of the mean time, which it takes for the entire population of IT professionals to complete the tasks.

We can see in Fig. 9.1 that overall the users took less time to complete the tasks with StriSynth. However, our sample size was relatively small (N=27). Therefore, we wanted to measure the confidence that our observations are reflective of the larger IT population beyond our small sample size. To do this we ran a paired sample *t*-test [268].

When running the paired sample *t*-test, we are checking the null hypothesis that the difference between the paired observations in the two samples is zero. Without going into the details of statistical methods, we need to compute the *p*-value. Any *p*-value of less than .05 is called *statistically significant*, indicating we have met a generally accepted threshold of confidence in our results [268].

By running these tests on our samples, we learn that a statistically significant difference was found in the *Move Files* ($p = .03$) and *Printing pdfs* ($p = .02$) tasks. The *p*-value of .03 means that, assuming StriSynth does *not* actually have any impact on time to complete the *Move Files* task, there is only a 3% chance that we could have observed the timing difference (or even some larger difference) between StriSynth and PowerShell presented Fig. 9.1. In other words, given these low *p*-values, we can be confident that using StriSynth does in fact have an impact on time to complete the task.

All together, our results support the claim that, for small scripting tasks of the type we presented to our users, PBE can be a more efficient programming paradigm. This is the expected result that is inline with the literature.

9.4.2 Reported helpfulness

Reaching beyond traditional measures for PBE, at the end of the study we also asked users to report how "helpful" they found both StriSynth and PowerShell. At this point, users did not know how long they took to complete the tasks with each of the tools. Users were asked the rate the helpfulness only based on their experience of using the tools during the study. The exact questions asked were "The following program was helpful for scripting/completing *Extract Filenames*/etc...", and users

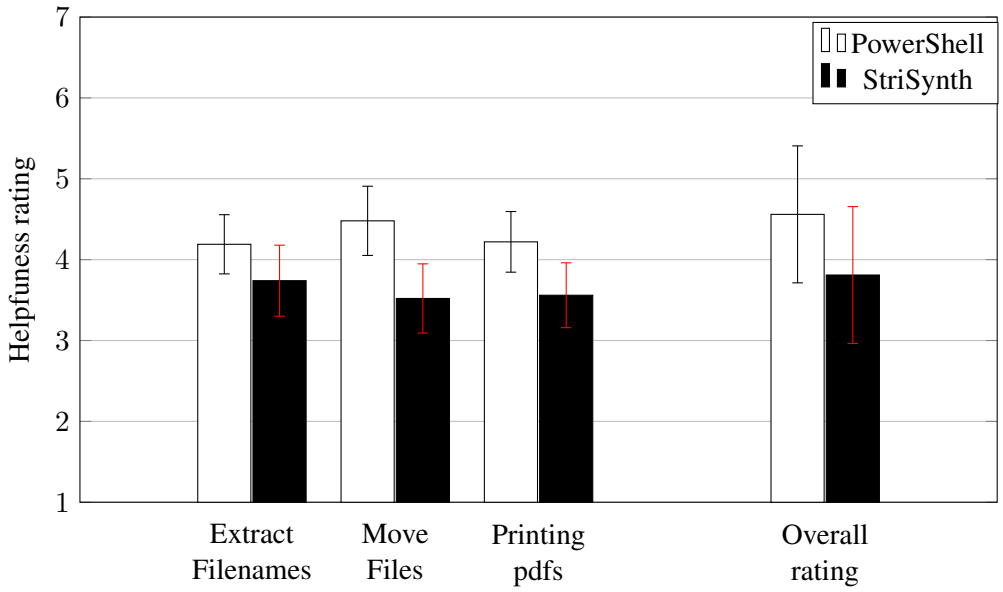


Figure 9.2: Users' (N=27) self reported measure of the helpfulness of each tool with standard error bars.

were asked to respond on a scale from 1 (strongly disagree) - 7 (strongly agree). We show the results from this survey question in Fig. 9.2, again with standard error bars. Users rated PowerShell as more helpful in all three tasks, with the *Move Files* task showing the most significant difference ($p < .01$).

The results in Fig. 9.2 show the surprising insight that, despite the efficiency of StriSynth as demonstrated in Fig. 9.1, users perceived PowerShell to be the more helpful tool. Unfortunately, as we did not anticipate such unexpected results, our study design did not include a more detailed definition of helpfulness, or ask users to give a more detailed description of their interpretation of what it means for a tool to be helpful. However, we can at least surmise from the results presented here, that efficiency is not a complete proxy measure for helpfulness of a tool.

9.4.3 Impact of prior user experience

The phenomena of familiarity of a tool as a stronger indicator of impact on user preference than efficiency was noted in other studies on the development of programming languages [269]. Our study asked users to self-report their prior experience with scripting languages in a post-study survey to understand the impact of user familiarity. The survey used a seven-point Likert scale for users assess the users' prior experience. Fig. 9.3 shows the distribution of experience in three categories for all users.

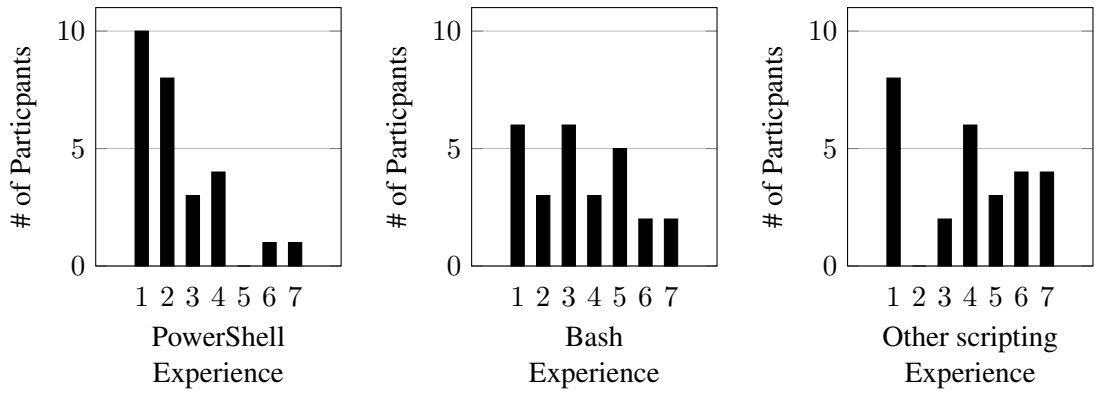


Figure 9.3: Users' (N=27) self reported prior experience with various scripting languages from 1 (Unfamiliar) to 7 (Expert User).

To understand the impact of prior experience on how the users interacted with StriSynth, we split our user population into two categories. We have the inexperienced user group, which is the users who rated their prior experience with PowerShell as a 1 (unfamiliar), and the complement set of users as the experienced user group, who rated their prior experience with PowerShell as (≥ 2). In Fig. 9.4, we show how these two groups performed in the study.

Fig. 9.4 shows that both groups of users completed the tasks faster with StriSynth. A more subtle and interesting insight is that inexperienced users had a greater relative speedup in task completion when using StriSynth. That is, inexperienced users benefited more from using StriSynth as compared to the benefit to experienced users. This provides evidence for the widely stated perception that programming by example is a domain well-suited for novice programmers.

9.4.4 Threats to Validity

In a usability study, it is important to avoid any possible selection bias in the call for participants. Selection bias can be an issue if the set of users selected systematically differs from the target population. The results we have presented are from a set of users that work as professional IT support specialists. We do not believe that we have any selection bias here because in this work, we specifically wanted to explore the impact synthesis can have in the real-world on such professionals.

A further potential threat to the validity of our results is in the social desirability bias, or need-to-please phenomena, whereby users will subconsciously try to produce the results they expect the researcher would like to see. This potential bias can occur when users are asked to compare a tool

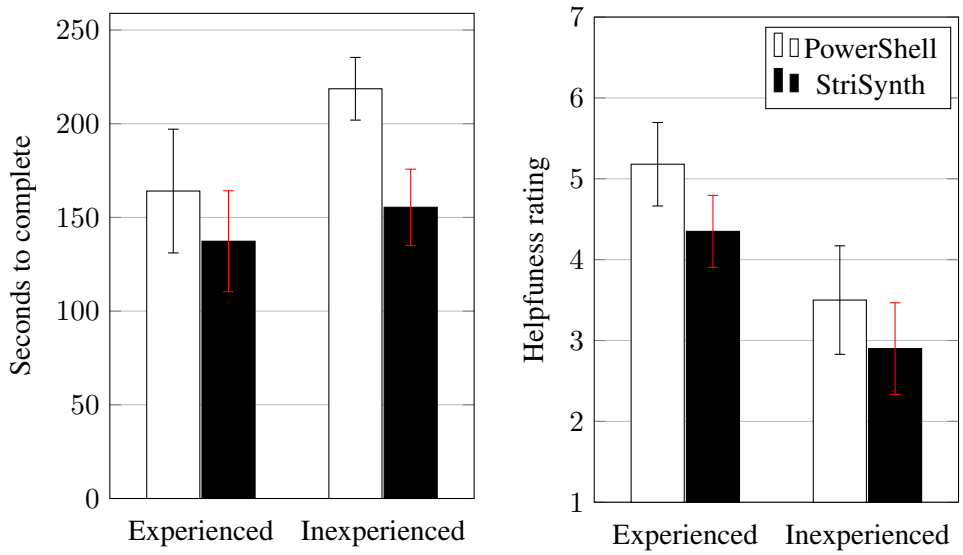


Figure 9.4: We grouped users as Experienced (PowerShell experience≥2, N=17) and Inexperienced (PowerShell experience=1, N=10). We report average time to complete the tasks, and self reported helpfulness of the tools, as separated by these two groups.

that is a known standard with an alternative that the user knows to be developed by the researcher. To do combat this issue, we presented StriSynth and PowerShell as tools named StriSynthA and StriSynthB. In this way, we framed the study as a comparison between two different tools that we had developed, eliminating the potential need-to-please bias. This was a critical component to our study design that allowed us to observe the disconnect between efficiency and users' perceived helpfulness of each tool.

9.5 Discussion

A key question from these results remains - how are efficiency and helpfulness different in the eyes of the users, and why is this difference manifest in our study? One tempting explanation is that is the result of two vastly different interfaces. PowerShell is an industrially developed tool, while StriSynth is a research prototype. However, both are command line utilities with a qualitatively similar user experience. Another possible point of departure is in the ability of the user to understand the function of a synthesized script from StriSynth. Trust in the result of program synthesis is a direction that needs further exploration, but StriSynth is unique in this respect, in that it provides an English text explanation of the synthesis result. Another possible interpretation may be tied to the expressivity of

the paradigm - StriSynth and other PBE tools are generally limited in their ability to directly work with a traditional programming language and use familiar concepts such as variables and loops. This may make a language seem less helpful for new users.

Finally, the results from our user study are specifically targeted at the impact of programming by example systems for scripting in IT professional populations. We must also consider how our results can be interpreted and extended to other PBE domains and program synthesis more generally.

9.5.1 Application to Related Work

Gulwani et al. [270] show that PBE is an effective paradigm for industrial application in spreadsheet manipulation, such as string transformations [266, 271], table transformations [272] and database look-ups [273]. Another approach is based on the abstraction of 'topes' [274], which lets users create abstractions for different data present in a spreadsheet. With topes, a programmer uses a GUI to define constraints on the data, and to generate a context-free grammar that is used to validate and reformat the data. These application domains of PBE are focused on a similar population of non-expert programmers, and so it may be possible to observe a similar efficiency vs helpfulness phenomena.

Unlike programming by example, in which the user provides input-output examples, programming by demonstration is characterized by the user providing a complete trace demonstration leading from the initial to the final state. There are several programming by demonstration systems [121], such as Simultaneous Editing [275] for string manipulation, SMARTedit [276] for text manipulation and Wrangler [277] for table transformations. As programming by demonstration requires intermediate configurations instead of just input and output examples, this paradigm is usually less flexible [278] than programming by example, but the synthesis problem is easier. Based on our results here, it is possible that this reduced flexibility may indicate users would rate programming by demonstration even less helpful (but possible more efficient) than PBE in certain domains.

The Myth [2] and Λ^2 [3] systems support PBE for inductively defined data-types in functional languages. In contrast to StriSynth which focuses on scripting tasks, these tools are focused on synthesis for more general purpose programming languages. The results from our study may be cautiously extrapolated other domains - while the theme of PBE is the same, interaction preference for users may differ when looking at general purpose languages.

Instead of providing specification in terms of examples or demonstrations, specification can also be given in more formal and complete ways. InSynth [4, 5], CodeHint [279] and the C# code snippets on demand [280] are systems that aim to provide code snippets based on context such as the inferred type or the surrounding comments. Leon [281] and Comfusy [6, 7] synthesize code snippets based on complete specifications, which are written in the same language that is used for programming. Sketch [8] takes as input an incomplete program with holes, and synthesizes code to complete the so that it meets the specification. These techniques provide a more nuanced interface that may seem, from a perspective of helpfulness, to be more similar to a traditional language paradigm.

9.6 Conclusions

Our study shows that users do not always correlate an efficient programming paradigm with a helpful paradigm. A more thorough exploration of this finding requires a follow up study, in particular to discover the definition of helpfulness that participants are using. A key question to answer would be whether users had erroneously perceived PowerShell to be more efficient and therefore helpful, or if users consciously have other metrics in mind that constitute the helpfulness of programming paradigm.

Chapter 10

Building Computing-Identity through Synthesis for Early-Stage Researchers

Work presented in this chapter was completed in collaboration Ruzica Piskac. Sections of this work have been previously published [282], and are reproduced here, at times in their original form.

The field of formal methods relies on a large body of background knowledge that can dissuade researchers from engaging with younger students, such as undergraduates or high school students. However, we have found that formal methods can be an excellent entry point to computer science research - especially in the framing of Computing Identity-based Mentorship. In this chapter, we report on our experience in using a cascading mentorship model to involve early stage researchers in formal methods, covering our process with these students from recruitment to publication. We present case studies (N=12) of our cascading mentorship and how we were able to integrate formal methods research with the students' own interests. We outline some key strategies that have led to success and reflect on strategies that have been, in our experience, inefficient.

10.1 Introduction

Engaging in early mentorship of students in computer science has been shown to be an effective means to improve outcomes in diversity, retention, and performance [283]. In this paper we report on our experiences in mentoring early stage researchers (such as high school students and undergraduate students). In our report, the mentors are based in Yale University, however the mentored students

came from various schools, even including from abroad. We introduced the students to research in formal methods, which is a subfield of computer science focused on improving software reliability using formal mathematical techniques. Traditionally, research in program verification and formal methods is considered a part of computer science with a high entry bar [284, 285]. Conducting research in formal methods requires practical expertise in programming languages and systems, as well as training in theoretical mathematical foundations and logic. In addition, courses in formal methods are usually taught only at universities and on a graduate level. Therefore, we first needed to introduce the students to the field of formal methods, and after that, additionally find a suitable project for each student.

We report on our experiences mentoring students in formal methods, with a focus on program synthesis, using the lens of Computing Identity-based Mentoring [286]. Computing Identity-based Mentorship posits that by engaging students' own backgrounds and interests we can help students build a sense of identity around computing. By engaging student with their own interests, it is possible to build a computing identity that helps to further engage and retain students in the field [287]. Through a number of case studies of our mentorship experiences, we provide anecdotal evidence of how the field of formal methods can be leveraged to engage students in their own interests, while also building a common community of practice [288]. Encouraging communities of practice in computer science has been shown to lead to stronger student outcomes [289–291].

We frame our mentorship model as a type of *cascading mentorship* [292], implemented similarly to prior mentorship models for early stage computer scientists [293]. The cascading mentorship model reimagines the mentor-mentee roles as interchangeable - asking mentees to act as mentors through mentoring younger students (as in [292]), or through peer mentorship.

Our mentorship model, derived from the work of Tashakkori et al [293], is shown in Fig. 10.1. The faculty mentor oversees the graduate students, who themselves closely work with a small number of early stage researchers. In addition to the graduate student's direct mentees, every graduate student also engages to some degree with the mentees of other graduate students as well. Additionally, the younger students are also put in a position to act as peer mentors to other early stage researchers. All the while, the faculty gives an opportunity to the younger students to more formally present their progress. We report in our case studies how, for some students, we have seen this strongly connected social graph within the lab environment lead to a stronger sense of community.

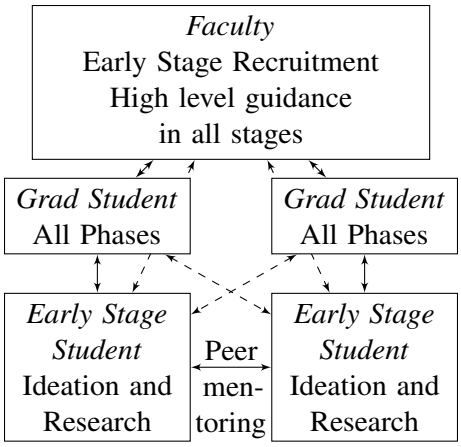


Figure 10.1: Our mentorship model engages all participants with each other.

In our presentation of our case studies (N=12), we break down the mentorship into three phases. First, we explain the recruiting phase to give insight into how the opportunity for mentorship began. Second, we look at the ideation stage of the research, where the faculty member, the graduate students, and early stage mentees collaborate to define and scope an appropriate project for the allotted time. Third, we look at the research stage itself, where the mentee is actively involving in producing novel results. Finally, in the presentation of each case study, we reflect on the outcomes of the mentorship, in terms of concrete benefits both for the mentees and the mentors. In many cases, the research resulted in publication.

When examining the positive outcomes of the mentorship experiences, we frame the student experience's in the context of the Thomas principles of mentoring success [294]. For the sake of clarity and concision, we in particular focus on the following principles:

1. *Identity Development* - forming and reinforcing a Computing Identity.

2. *Academic Support* - ensuring efficient access to expertise within the lab setting for guiding mentees and ensuring continued progress.

3. *Sense of Belonging* - building communities of practice in our research field of formal methods.

4. *Leadership Development* - developing mentees' sense of leadership and local expertise within our community of practice encourages further engagement.

10.2 Formal Methods as Introductory Research

One of the key insights of prior work was that students seeing a clear impact of their work can have a positive impact on retention [287]. In this report, we focus on our own research programs in the field of formal methods and the potential for students to see impact.

The main goal of research in formal methods is to gain better insight into code by using formal mathematical reasoning. Research in formal methods began with development of computer science: for instance, works of Alan Turing already talk about formal proofs of program correctness. However, only recently we have seen formal methods being applied to industrial software. Today, formal methods are used in many major software companies, including Google [295], Amazon [296], Facebook [297] and Microsoft [298].

The range of applications where formal methods are used covers almost all aspects of computer science: we can obtain guarantees that servers are secure [299], ensure that data centers will not crash [300], assist student programmers with bugs [301, 302], or write programs that write programs [12] - to name a few applications.

Taking into account the increasing need for scientists working in formal methods, there has been significant community effort in developing programs to assist early stage researchers in covering the basics of the field [303, 304]. These programs are also designed to help lower the barrier to entry for formal methods research. This is in part because, within the field itself, formal methods is seen as having too high an entry bar to make it an appropriate field for students' initial experience with research without significant preparation.

While this perceived high barrier to entry may at first seem to imply it is not a good fit for younger students with limited programming experience, we have found the opposite to be true. In the subsequent sections we describe a number of case studies of our collaborations with high school and undergraduate students in our lab. All these projects were defined after initial discussions with the students and tailor-made to fit around the student's application domain of interest.

Of all the fields of formal methods, we found that students were particularly interested in software synthesis and its applications. The goal of software synthesis is to automatically derive code based on given specifications. A specification describes *what* the program should do without going into details of *how* it should be implemented. There are various ways of providing the specification: one of the

most commonly used approaches is to take a set of examples that clearly illustrative the intended behavior of the code.

Based on these examples, a synthesis tool should automatically generate corresponding code. This branch of synthesis is known as "programming by example" [305]. While program synthesis might have looked like an unreachable goal a decade ago, today – due to the development of formal methods – the class of programs that can be automatically synthesized has dramatically increased. Program synthesis is even used in industrial software: it is vital part of the Flash Fill feature in Microsoft's Excel [143].

10.3 Research Conducted by High School Students

In this section we describe the projects that were conducted by high school students ($n = 4$). All these projects are related to software synthesis, as the concept of program synthesis was the easiest for us to describe to high school students, and they could immediately see the impact of their work.

Of the four projects we present here, one used the research experience as part of a course credit, while the other three included formalized summer internship experiences. We conducted semi-structured exit interviews with these three students. The projects described here were largely defined in content and in scope by us, but the technical solutions were driven by the students. In all cases, the project topic was found as an intersection between our group's ongoing work on program synthesis, and the students' background and existing interests in computing.

10.3.1 Synthesis + HCI

Recruiting: Our first high school research intern came to us from a cold email (as the student entered 11th grade). This student reached out, as part of a research course at their high school, to seek out opportunities for collaboration. The student mentioned our publications on program synthesis in their email, and was interested in the potential benefit of synthesis for new programmers. The student had prior research experience, having done a Human Computer Interaction (HCI) study as a school project.

Ideation: In initial discussions, the student demonstrated interest in HCI. We had an existing program synthesis tool implemented to apply the programming-by-example paradigm to PowerShell

scripts. While we had already developed the tool itself, we were lacking any usability studies. To complement the students' existing interests, we planned to develop a user study to learn about what users expect and want from a programming-by-example engine.

Research: We had previously developed a user study that was focused on measuring the time it took participants to complete a number of tasks with and without the programming-by-example engine. The pilot was limited in scope, so the high school student took on the task of redesigning and extending this study. In developing the user study, the student made a point to include both timing information (how quickly participants complete the tasks) as well as how helpful the participants found the tool. To our surprise, we found that while participants completed tasks more quickly with programming-by-example, they found manually writing code to be more helpful [22].

Outcomes: We published these results with the high school student and his teacher. While we largely handled the writing ourselves, the student drove the analysis and interpretation of the results with the help of his teacher. This allowed the student to actively participate in the publication process, without the prohibitive overhead of learning scientific writing. This helped the student develop a leadership role as the local expert on user study analysis.

We first presented the work at a workshop without proceedings, and had a graduate student present a talk. We later published the work at another workshop, and the high school student used a recording of the graduate student's talk as guide in the preparation of their own presentation. In having the student present the work, we helped form a sense of computing identity in the student, such that the student is now majoring in Computer Science at college.

10.3.2 Synthesis + Machine Learning

Recruiting: One high school student was recruited (while the student was in 11th grade) during one of our outreach activities, at a hackathon in Bermuda [306]. At this hackathon, we were running a workshop on machine learning techniques for local Bermudian high school students with prior experience with programming. We contextualized the machine learning workshop within our research in order to both deliver the content and expose the students to novel applications of the material. One student was particularly interested in the application of machine learning to program synthesis, so we offered to stay in contact to begin a research collaboration.

Ideation: During the hackathon, the student had already discussed possible applications of their own expertise, machine learning, to our work on program synthesis. Through the research stage, we iteratively refined the topic such that this became its own research project that was not reliant on any of our existing work. The project was still framed in such a way so that it directly helped our ongoing work.

Research: We worked with this student remotely after the hackathon in December for four months, and in May invited the student to join us to work in person for a four week internship in the summer. Being both underage and an international student, the process of formalizing the internship presented challenges. The best solution we found in the end was to have this student's stay in New Haven accompanied by their parent. The student's work was largely self-directed in technique, while we provided guidance in scoping the project appropriately. We were aiming for a scope such that we could have a measured impact, but also complete an initial evaluation by the end of the students' visit to campus.

Outcomes: The student was able to develop a tool with strong results and subsequently published a paper [16]. In our exit interview with the student, we saw that the student connected to the impact of their work in the larger research community.

> "If you're very interested in the topic while you're in high school, I think [research experience] is beneficial due to the fact that it allows you to really experience how real world problem solving happens in a very real environment."

We also saw the student developed a sense of belonging that was conducive to productive research conversations. When asked about the benefits of the internship experience, the students responded:

> "Just being around the office and having fun conversations about things I'm interested in, and that being okay, just being able to have fun in what you're interested in."

In regards to academic support, we saw the student had developed the critical skill of asking questions.

> "There is no point just sitting around and wondering whether a question is a dumb question or not, just ask the question. You'll be surprised how much you learn just being curious."

From the first day of the internship, we stressed the importance of proactively asking questions. This was especially helpful as there were a number of technical challenges the student faced a new researcher to formal methods that could be quickly answered by more senior mentors. This was also an opportunity for us to leverage the cascading mentorship model, as the majority of the questions could initially be directed to the graduate student rather than the faculty member.

10.3.3 Synthesis + Apps

Recruiting: Another high school student came to us from a cold email (as the student entered 12th grade) with an interest in developing mobile applications. This student reached out on the recommendation of friends to email professors to find research opportunities. The student reported reaching out roughly 100 professors, and reported hearing back at all from about five, including our group. We were able to find intersection with our research on reactive systems synthesis [12].

Ideation: In our initial Skype sessions, we worked with the student to design the high level goal of the project and layout the way in which we could find an overlap with the student's existing interests. The student had sent a resume which mentioned a course that involved VeriLog, a programming language popular in the 'reactive synthesis" subfield of formal methods As such we searched for projects that overlapped with our own work on reactive synthesis The goal of this student's project was to reuse the existing program synthesis infrastructure we had built for general reactive synthesis problems, and adapt it to synthesize an Android app in Kotlin.

Research: We initially worked with the student over Skype, then invited the student to work with us over for four weeks over the summer. Again, as an underage international student, this student's stay in New Haven was accompanied by their parent. This project was a good fit for the student, who had limited programming experience, as it the programming aspect mostly involved editing existing parsing code, and following predefined patterns to generate program code. The student had a stronger math background, and was able to explore more of the theoretical problems with reactive synthesis for Kotlin. We did not anticipate having the student address these theory issues, but the student in fact took the project in this direction themselves after completing the programming aspect of the project.

Outcomes: The majority of this student's research experience was during an onsite internship that was overlapping with the student from Sec. 10.3.2. Organizing overlapping internships was

recommended a key lesson learned in organizing internships from prior work [283]. We confirmed that this is a productive insight in our own experience. Throughout their time on campus, the two students worked closely together on both projects - helping each other with both technical and conceptual issues. This directly leveraged the peer mentorship aspect of our mentorship model in Fig. 10.1. In addition to seeking help from the student's peers, the student also discovered the importance of leverage the larger academic support network.

> "I'm very comfortable and the environment is such that I can go out and say 'Hey, do you think you could help me with this problem' and everyone is super open to helping me."

10.3.4 Synthesis + Hardware

Recruitment: We recruited one high school student (as the student entered 10th grade) from the local New Haven area. This student had been working on writing a compiler for Basic as a side project and reached out for some guidance on ways to get more involved in computer science.

Ideation: We worked with this student for roughly a year - mostly over email with occasional in-person meetings. Because this student was interested in low-level language details, we designed a project working with intermittent computing and program synthesis. The goal of this project was to synthesis code that is able to run on specialized hardware devices that run on harvested ambient (e.g. radiowaves) energy.

Research: The student's project has two components - a hardware side focused setting up a test framework on the specialized hardware - and the software component of program synthesis. The student first setup the hardware and gained basic familiarity with work in this domain. The student's first approach to synthesis was to utilize their existing work on writing a compiler. While this was not an efficient solution in the end, it gave the student exposure to the technical details necessary for working in this domain. With that background, the student was able to reframe and rescope their work into a new direction.

Outcomes: While the hardware setup was less of a novel research project, it gave the student a strong sense of local expertise with these devices. The student worked primarily with one graduate

student, but was able to get feedback from another graduate student with a background in electrical engineering.

As a local student, the collaboration is logistically simple to facilitate as an ongoing project. We also hosted this student for a four week internship during the same time period as the students in Sec. 10.3.2 and Sec. 10.3.3. One negative results we found with this setup was that, while the students occasionally worked together, the sense of peer mentorship was not as strong with this student. We hypothesis the age gap may have contributed to this, as well as the student's prior local commitments decreasing the amount of interaction time with the other two interns.

10.4 Research Conducted by Undergraduate Students

In this section we describe projects involving undergraduate students ($n = 8$). While these projects might not seem more demanding or complex than the projects conducted by high school students, undergraduate students were given more freedom in defining their research agenda. Their projects were initially less clearly defined. They were given a general description of a problem and led the discussion about possible research directions. In this way, they helped to outline the project, based on their own research interests, which made the students additionally motivated to participate in the project. Their research interests were often defined either by courses that they had taken until that point, or by their extracurricular activities. All the projects presented in this section are either published at a top venue, or are currently under a submission to a top tier conference. The students conducted their research either as a part of a summer research internship, or received course credits, or were working on these projects in their own free time.

10.4.1 Synthesis + HCI

Recruitment: One student (a rising senior at the time of recruitment) came to us at the end of the school year looking to gain research experience, with the intention of applying to graduate school. The student expressed an interest in topics of HCI, but did not have a concrete project in mind. As much of our group's work in formal methods focuses on applications intended to help developers, we are nearly always in need of further user studies.

Ideation: We presented a number of our projects to this student, and asked the student to identify a preference and direction within the HCI space. The student was interested in our work on synthesis in a "live" environment, whereby program synthesis runs in realtime to assist developers as they are writing code. We had some initial work on developing the tooling for this project, but lacked any user studies to drive our interface decision making.

Research: The student worked in a 12-week onsite internship to design and implement an online version of the live synthesis interface in order to then deploy the user study to a wider audience. The student also designed the user study, and conducted numerous pilot studies with early iterations of the online interface and the study design.

Outcomes: At the time of writing, the study is currently being deployed, and the student is continuing to work with us to prepare a paper submission. This student particularly developed a strong sense of belonging in the field. Initially, the student did not have prior experience, in coursework or otherwise, in formal methods, but was able to quickly adapt to the field. Through working on the project, the student developed local expertise within our community of practice and when reflecting on the experience, said:

> "I think one reason that I find going into academia, as a Chicano, so important is that I want to see other people like me in these fields and sometimes it's difficult to see yourself in these places, see yourself in academia, going to grad school when you don't see many leaders like that. But through this experience I found there is a lot of great and supportive people within the field and through that I feel so much more confident about getting into academia, applying to grad school, and cranking out a thesis one day"

10.4.2 Synthesis + Systems

Recruitment: Two undergraduate students were working on this project. One student approached us after taking the course (in their 3rd year) given by our group and asked to work on a verification-related project. The other student was a personal friend of group members so initially became interested in the project through discussions with group members. As the project progressed, this student (in their 4th year) became more involved and took a lead for certain parts of the project. Both students were supervised by a PhD student, using the cascading mentorship model.

Ideation: Our group had previously worked on configuration file analysis. Incorrectly setting up a configuration file has been found to cause more server outages than bugs in the code. These students worked on learning specifications of correctness of configurations by analyzing a large number of existing configuration files. From that corpus we learned properties about configuration files, and used them as a specification, enabling us to formally verify configuration files and detect previously unknown bugs.

Outcomes: This work resulted in a paper at a top venue [21]. Both students were exposed to the formal methods community either at a conference, or by attending summer schools. While both students worked as software engineers upon graduating and completing this project, one of them has recently returned to a non-CS graduate school at Yale.

10.4.3 Synthesis + Music

Recruitment: One non-traditional application field we have found rich collaborations from is computer music. We have had three students (at the time of their recruitment, a 4th year, and two 2nd years) work with us on a project combining program synthesis with music. One student participated in a 12-week internship, one participated in this project as their senior project, and one is currently participating as an extracurricular activity.

Ideation: We initially started this project because one PhD student was interested in the intersection of formal methods and music. The goal of the project was to build a programming-by-example engine for audio files. The engine takes two audio wave files, and automatically synthesizes digital signal processing programs that transform the input to the output audio file. From a technical perspective, this work focused on combining machine learning techniques with formal methods for application in music.

Research: The first student worked on this over the summer and was mainly focused on implementation, building much of the core codebase of the project that handles the machine learning components. The senior thesis project focused on developing a theory of how to apply formal methods techniques to the project. Currently, the most recent undergraduate researcher is working on implementing and extending this theory.

Outcomes: The work of two of the students on this project was published at a specialized computer music workshop that focuses on programming language design [15]. In the vein of

institutional academic support, one student was able to use this project to gain credit for their senior thesis.

10.4.4 Symbolic Execution Engine

Recruiting: This student started to attend our group meetings immediately after joining Yale (a freshman at the time of recruitment) in order to learn more about research.

Ideation: Due the opportunity of getting this student involved early, we gave the student an especially challenging project of developing a symbolic execution engine for Haskell programming language. The student was directly supervised by a PhD student from the group, but also worked independently, developing their own research agenda.

Research: A symbolic execution engine takes as input a program, and instead of executing the program on some concrete values, it runs the program using symbolic values. The result is a mathematical formula that describes the program's behavior. Although symbolic execution engines are well studied, languages like Haskell, based on lazy semantics, had no efficient symbolic execution engine. The problem required a deep understanding of Haskell's semantics: the research combined some foundational theoretical problems, but was also implementation-intensive.

Outcomes: While the student kept working with our group throughout their undergraduate program, they also developed other research interests at the intersection of mathematical reasoning and computer science. Nevertheless, the symbolic execution engine work resulted in papers accepted and presented at two top ACM sponsored venues [302, 307]. In addition, we helped the student explore other research directions, and the student did three research internships: one at a different university, one at an international research institute, and one at a research lab of a large company. The student presented work at various international meetings and volunteered at conferences, developing a strong sense of helping the community. The student also received the 2019 NSF Graduate Research Fellowship, was accepted to several PhD programs, and is currently attending one of them.

10.4.5 Program Repair

Recruiting: The student took a course (during their 3rd year) from our research group and asked directly about a possibility to conduct an independent research study. The student worked directly with a professor, as the group had no graduate students at the time.

Ideation: The student was doing a double major in math and computer science, and was interested in finding a way to leverage both these backgrounds.

Research: When writing code, a user might be sure about what they want to write, but we are not sure about the right ordering of all arguments when invoking library functions. When programming, a user might write code that does not compile but clearly outlines their intentions. The student used their expertise in graph algorithms to develop a tool that repaired these errors.

Outcomes: This work was published and presented by the student at a top conference [308]. The student also presented a poster at the ACM Student Research Competition, and received second place. The student was accepted to several PhD programs, and is currently attending one of these schools.

10.5 Lessons Learned

Reflecting back on our experiences with using formal methods research as an entry point to computer science research and a way to build a computing identity, we explore here some key lessons we have learned.

10.5.1 Recruitment

Recruiting students for collaboration has been one of the most important steps of our process. At Yale University, we are privileged to have a large pool of talented undergraduates to draw from - however the main challenge is awareness. There is some existing infrastructure in place - as in many universities - to encourage Computer Science majors to complete a senior thesis as a research collaboration with a lab. While this is effective, as such a thesis is completed in the students' final year, the student then leaves just as they begin to be particularly productive from a research perspective. Involving students at early stages in their academic career not only has helped the students themselves, but also has increased the long term quality of our collaboration.

One of the most successful recruiting resources we have developed is an institutional memory among undergraduates. A number of our undergraduate research interns have come to us recommended by a previous intern. This has allowed us to increase the number of students we can work with, and also yields students who have a better idea of what to expect from formal methods research.

Unfortunately, due to the time constraints on faculty members, proactive recruiting dedicated to high school students was not a practical strategy in our situation. The cold email is an increasingly common strategy for high school students to get involved in research. Students reach out to a large number of professors (in the case of Sec. 10.3.3, as many as 100) in hopes that one will respond and allow them participate in an unpaid internship in their lab. While the volume of these emails can be overwhelming, we have had success by forwarding these students to graduate students, following the cascading mentorship model. In this way, the faculty member provides mentorship training opportunities to the graduate student, and the graduate student can utilize the students' assistance.

In terms of selecting high school students, while the high school students' resumes provided some clues as to their prior experience, we found the students to be too young for their resume to be a useful predictor of their success in research. Generally, demonstrating some prior experience and interest in programming was sufficient. Beyond this, students largely self-selected when presented with concrete research tasks. However, from another perspective, this is a potentially negative result. While we have had a number of successful high school interns, as listed in Sec. 10.3, a number of students have also dropped contact with us after a short time. Investigating effort-effective strategies to stay engaged with more students who initially reach out for collaboration is a space for further research.

In recruiting students, we tried to limit the extent to which the student's demonstrated pre-existing computing identity played a role in our selection process. However, it is likely that our own bias to seek out student's with some computing identity guided our recruitment. This was certainly the case in Sec. 10.3.2, where the student had already developed a strong computing identity, and so was able to immediately engage with us in discussions of potential research collaborations. It was in part because of this prior technical experience that the student's internship was so successful. In our future recruiting efforts, we hope to find the right balance between seeking students with the requisite level of experience, but also finding students where we can help further nurture a computing identity that may not develop as well without outside intervention.

10.5.2 Ideation

We learned to ensure that student projects are *noncritical paths* along our larger research vision, but still contribute significant value, which encourages the students' sense of computing identity. Additionally, granting students the latitude to guide their project scope and how their project integrates into the larger research vision encourages the development of a stronger computing identity, with the student as the leader, or "local expert", of their topic. In a way, this allows the students to drive their work in such a way as to create their own critical path.

The idea of developing student's into local experts also helps to build a community of practice around formal methods in our lab. Rather than focusing on developing a complete understanding of the field, which would be untenable given the time constraints, we encourage student interns to dive deep into their project and build expertise in that domain. In our experience, this has helped to build a stronger sense of belonging and a more effective community of practice, as the students feel empowered to contribute unique perspective to the group's work.

The benefits of mentorship go beyond the mentees, especially with the cascading mentorship model. For graduate students, the process of advising students and defining scope of projects is valuable experience. In our experience, undergraduate research is generally fairly regulated, such as being formalized as a course, a summer internship, or a campus job. In contrast, working with high school students allows graduate students to take more risks with mentorship, which in turn creates more learning opportunities. Especially for graduate students who are planning to go on the academic job market, the experience of running a 'micro-lab' environment has been particularly useful.

10.5.3 Research

The first and most important guideline for our research programs has been to provide academic support to early stage students - especially in stressing the importance of asking the right questions. Students have tended to ask too few questions, and waste time on technical challenges that can be answered quickly by the mentor. This is dangerous to the success of the collaboration, as it can cause students to lose interest in the project.

Consequentially, we have found it to be important that the mentor makes sure the student feels comfortable asking questions. However, sometimes a question will be more appropriate for the student to discover on their own. This is especially important for developing computing identity, as we have seen this provide students the confidence needed to solve problems on their own.

We also found that the academic support and mentorship model helped with building a sense of belonging, as more clearly demonstrated by the student described in Sec. 10.4.1. In future internships, we plan to continue to stress the important of academic support and building a sense of belonging. So far, we have had limited organized social gatherings with the set of all interns. We predict that organizing such events may even further help to strengthen the sense of belonging in all students, not only to computer science itself, but also on a more local level, to our lab.

In working to minimize the interruptions to the workflow of graduate students (as mentoring students was *not* the graduate students' primary responsibility), we found that having multiple students working on projects at the same time and *in the same space* encouraged peer mentoring. By connecting the students, they can lend their expertise to each other and progress more quickly. This worked particularly well with the high school students, but had not worked well with the undergraduates. We suspect that this was due to the great flexibility the undergraduates had in the physical workspace. We plan to further investigate strategies to increase informal peer mentoring among undergraduate researchers.

A challenge we found with international high schools students was their ability to travel for an internship alone. Our solution was to require parental supervision, but this introduces a socioeconomic bias, potentially restricting access for some students. This is a critical challenge we face moving forward. Our strategies so far have involved using video chat and frequent emails as our main collaboration tool for student interns that predominately work remotely. While this has been effective, the extent to which this approach can build a sense of belonging to the lab has been limited in our experience.

10.6 Publishing

In our experience, early stage students have had limited scientific writing experience, so the majority of the initial drafts were written by graduate students and faculty. However, we have been able to

keep students engaged during the publication process by assigning other critical tasks, for example, in the analysis of data.

When possible, ensuring that publication happens before January of the students' senior (final) year of high school will deliver the most value to the student, as the publication can be included in their college application (for the American college application cycle). This acts as a good motivator for them to complete their projects by a hard deadline. We have found both workshops and full conference papers to be possible to publish with high school student work.

One challenge we have found with the computer science conference model is the challenge of travel. Publishing in conferences that are geographically far from the students has presented challenges in allowing the student to fully participate in the research experience. While we have sometimes been able to fund undergraduate researchers during their time at the undergraduate institution, if a student graduates before the conference takes places, it becomes more difficult to find the resources to allow the student to attend. In the case of one of the student co-authors in Sec. 10.4.2, the student was able to use funding from their company to attend the conference where we had published the work, as the work was also aligned with the goals of their post-graduation employer. However, this problem is even exasperated in the case of high school students. Our current solution is to intentionally target conferences that are relatively local to allow the student to attend as well.

When we have had papers with a high school students accepted we have had more senior coauthors present the work as high school students' presentation skills are, expectedly, underdeveloped. In one case, we first presented the work at a workshop without proceedings and video recorded the presentation. For the second presentation of the work (for which there were published proceedings), the high school coauthor then had material on which to model their own presentation.

Chapter 11

Conclusions

In this dissertation we have examined the role synthesis can play through three lenses of software - design, implementation, and deployment. Through combining abstraction techniques from programming languages with temporal logics, we introduced Temporal Stream Logic, a new logic for the synthesis of reactive programs. With Temporal Stream Logic, we were able to apply synthesis to domains which were previously out of scope for existing tools. One of the key aspects of Temporal Stream Logic was the abstraction from data transformation functions. Thus, in the second part of this dissertation, we looked at programming by example, and ways that data-driven techqniues can be leveraged to build more effective tools for implementation level synthesis. Finally, we looked at the deployment stage of software development, and how we can leverage synthesis to assit developers with the infrastructure aspects of their work.

So far, we have addressed synthesis techniques that target individual stages of the software development process. Our techniques point to many opportunities for combinations of these methods. Temporal Stream Logic uses function abstractions, the implemtation of which can then be completed with, for example, the programming-by-example technqiues we introduced in the second part of the disseration. However, there are still many steps to take in this direction. Eventually, all three of these directions should be merged together to build a complete synthesis-based development pipeline. The vision of this dissertation is that program synthesis can be used to help developers skip over the tedious, difficult problems in the software development process, and instead focus on the reasons we love programming - solving the *interesting*, difficult problems.

Bibliography

[1] B. G. T. (Firm)(US). Beyond point and click: the expanding demand for coding skills. 2016.

[2] P.-M. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *PLDI*, 2015.

[3] J. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *PLDI*, 2015.

[4] T. Gvero, V. Kuncak, and R. Piskac. Interactive synthesis of code snippets. In *CAV*, pages 418–423, 2011.

[5] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac. Complete completion using types and weights. In *PLDI*, pages 27–38, 2013.

[6] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, pages 316–329, 2010.

[7] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Comfusy: A tool for complete functional synthesis. In *CAV*, 2010.

[8] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[9] R. Alur, D. Fisman, S. Padhi, A. Reynolds, R. Singh, and A. Udupa. The 6th competition on syntax-guided synthesis. `https://sygus.org/comp/2019/results-slides.pdf`, 2019. Accessed: 2019-11-20.

[10] A. Nötzli, A. Reynolds, H. Barbosa, A. Niemetz, M. Preiner, C. Barrett, and C. Tinelli. Syntax-guided rewrite rule enumeration for smt solvers. *SAT*, 2019.

[11] M. Nye, L. Hewitt, J. Tenenbaum, and A. Solar-Lezama. Learning to infer program sketches. *arXiv preprint arXiv:1902.06349*, 2019.

[12] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Temporal stream logic: Synthesis beyond the bools. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, 2019.

[13] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Vehicle platooning simulations with functional reactive programming. In *Safe Control of Autonomous Vehicles Workshop at CPSWeek*, 2017.

[14] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Synthesizing functional reactive programs. In *Haskell Symposium*, October 2019.

[15] M. Santolucito, K. Rogers, A. Lombardo, and R. Piskac. Programming-by-example for audio: Synthesizing digital signal processing programs. In *Function Art and Music (FARM) at ICFP*, 2018.

[16] K. Morton, B. Hallahan, E. Shum, R. Piskac, and M. Santolucito. Grammar filtering for syntax-guided synthesis. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence (AAAI)*, 2020.

[17] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[18] A. Breland. Fcc: Over 12,000 callers couldn't reach 911 during at&t outage. http://thehill.com/policy/technology/325510-over-12000-callers-couldnt-reach-911-during-att-outage, March 2017.

[19] Z. Yin et al. An empirical study on configuration errors in commercial and open source systems. In *Symposium on Operating Systems Principles*, 2011.

[20] M. Santolucito, E. Zhai, and R. Piskac. Probabilistic automated language learning for configuration files. In *International Conference on Computer Aided Verification (CAV)*, 2016.

[21] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac. Synthesizing configuration file specifications with association rule learning. *Proc. ACM Program. Lang.*, (OOPSLA), October 2017.

[22] M. Santolucito, D. Goldman, A. Weseley, and R. Piskac. Programming by example: efficient, but not "helpful". In *Evaluation and Usability of Programming Languages and Tools (PLATEAU) at OOPSLA*, 2018. Also presented at SYNT 2018.

[23] N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. In *VMCAI*, 2006.

[24] C. Helbling and S. Z. Guyer. Juniper: A functional reactive programming language for the arduino. FARM 2016, New York, NY, USA, 2016. ACM.

[25] I. Perez. GALE: a functional graphic adventure library and engine. In *Proceedings of the 5th ACM SIGPLAN International Workshop on Functional Art, Music, Modeling, and Design, FARM@ICFP 2018, Oxford, UK, September 9, 2017*, pages 28–35, 2017.

[26] G. Jing, T. Tosun, M. Yim, and H. Kress-Gazit. An end-to-end system for accomplishing tasks with modular robots. In Hsu et al. [309].

[27] R. Bloem, S. Jacobs, and A. Khalimov. Parameterized synthesis case study: AMBA AHB. In Chatterjee et al. [310], pages 68–83.

[28] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for guis. In Boehm and Flanagan [311], pages 411–422.

[29] M. Santolucito, D. Quick, and P. Hudak. Media modules: Intermedia systems in a pure functional paradigm. In *ICMC 2015, Denton, TX, USA*, 2015.

[30] R. Bloem, B. Jobstmann, N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of reactive(1) designs. *J. Comput. Syst. Sci.*, 78(3):911–938, 2012.

[31] B. Finkbeiner and S. Schewe. Bounded synthesis. *STTT*, 15(5-6):519–539, 2013.

[32] S. Jacobs, N. Basset, R. Bloem, R. Brenguier, M. Colange, P. Faymonville, B. Finkbeiner, A. Khalimov, F. Klein, T. Michaud, G. A. Perez, J.-F. Raskin, O. Sankur, and L. Tentrup. The

4th reactive synthesis competition (SYNTCOMP 2017): Benchmarks, participants and results. In *SYNT 2017*, volume 260 of *EPTCS*, pages 116–143, 2017.

[33] T. Wongpiromsarn, U. Topcu, and R. M. Murray. Synthesis of control protocols for autonomous systems. *Unmanned Systems*, 1(01):21–39, 2013.

[34] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Vehicle platooning simulations with functional reactive programming. In *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, April 21, 2017* [312], pages 43–47.

[35] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, January 1980.

[36] E. L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 04 1946.

[37] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[38] C. Elliott and P. Hudak. Functional reactive animation. In Jones et al. [313], pages 263–273.

[39] H. Apfelmus. Reactive-banana. Haskell library available at `http://www.haskell.org/haskellwiki/Reactive-banana`, 2012.

[40] C. Baaij. *Digital circuit in CλaSH: functional specifications and type-directed synthesis*. PhD thesis, University of Twente, 1 2015. eemcs-eprint-23939.

[41] A. Courtney, H. Nilsson, and J. Peterson. The yampa arcade. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 7–18. ACM, 2003.

[42] I. Perez, M. Bärenz, and H. Nilsson. Functional reactive programming, refactored. In Mainland [314], pages 33–44.

[43] Z. Shan, T. Azim, and I. Neamtiu. Finding resume and restart errors in android applications. In Visser and Smaragdakis [315], pages 864–880.

[44] B. Finkbeiner, F. Klein, R. Piskac, and M. Santolucito. Temporal stream logic: Synthesis beyond the bools. *CoRR*, abs/1712.00246, 2019.

[45] G. H. Mealy. A method for synthesizing sequential circuits. *The Bell System Technical Journal*, 34(5):1045–1079, 1955.

[46] A. Church. Edward f. moore. gedanken-experiments on sequential machines. automata studies, edited by ce shannon and j. mccarthy, annals of mathematics studies no. 34, litho-printed, princeton university press, princeton1956, pp. 129–153. *The Journal of Symbolic Logic*, 23(1):60–60, 1958.

[47] A. Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977* [316], pages 46–57.

[48] A. Pnueli and R. Rosner. On the synthesis of an asynchronous reactive module. In Ausiello et al. [317], pages 652–671.

[49] H. Liu, E. Cheng, and P. Hudak. Causal commutative arrows. *J. Funct. Program.*, 21(4-5):467–496, 2011.

[50] H. Liu and P. Hudak. Plugging a space leak with an arrow. *Electr. Notes Theor. Comput. Sci.*, 193:29–45, 2007.

[51] J. Yallop and H. Liu. Causal commutative arrows revisited. In *Proceedings of the 9th International Symposium on Haskell*, pages 21–32. ACM, 2016.

[52] S. Lindley, P. Wadler, and J. Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. *Electronic Notes in Theoretical Computer Science*, 229(5):97–117, 2011.

[53] A. van der Ploeg and K. Claessen. Practical principled FRP: forget the past, change the future, frpnow! In Fisher and Reppy [318], pages 302–314.

[54] C. Helbling and S. Z. Guyer. Juniper: a functional reactive programming language for the arduino. In *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design*, pages 8–16. ACM, 2016.

[55] R. Trinkle. Reflex-frp. `https://github.com/reflex-frp/reflex`, 2017.

[56] T. E. Murphy. A livecoding semantics for functional reactive programming. In Janin and Sperber [319], pages 48–53.

[57] D. Winograd-Cort. *Effects, Asynchrony, and Choice in Arrowized Functional Reactive Programming*. PhD thesis, Yale University, December 2015.

[58] P. Faymonville, B. Finkbeiner, and L. Tentrup. Bosy: An experimentation framework for bounded synthesis. In Majumdar and Kuncak [320], pages 325–332.

[59] B. Finkbeiner. Synthesis of reactive systems. In J. Esparza, O. Grumberg, and S. Sickert, editors, *Dependable Software Systems Engineering*, volume 45 of *NATO Science for Peace and Security Series, D: Information and Communication Security*, pages 72–98. IOS Press, 2016.

[60] P. Madhusudan. Synthesizing reactive programs. In Bezem [321], pages 428–442.

[61] C. Gerstacker, F. Klein, and B. Finkbeiner. Bounded synthesis of reactive programs. In *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*, pages 441–457, 2018.

[62] A. Jeffrey. LTL types FRP: linear-time temporal logic propositions as types, proofs as functional reactive programs. In Claessen and Swamy [322], pages 49–60.

[63] W. Jeltsch. Towards a common categorical semantics for linear-time temporal logic and functional reactive programming. *Electr. Notes Theor. Comput. Sci.*, 286:229–242, 2012.

[64] A. Cave, F. Ferreira, P. Panangaden, and B. Pientka. Fair reactive programming. In Jagannathan and Sewell [323], pages 361–372.

[65] N. R. Krishnaswami. Higher-order functional reactive programming without spacetime leaks. In Morrisett and Uustalu [324], pages 221–232.

[66] R. Ehlers, S. A. Seshia, and H. Kress-Gazit. Synthesis with identifiers. In McMillan and Rival [325], pages 415–433.

[67] R. Bloem, G. Hofferek, B. Könighofer, R. Könighofer, S. Ausserlechner, and R. Spork. Synthesis of synchronization using uninterpreted functions. In *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014* [326], pages 35–42.

[68] R. Alur, S. Moarref, and U. Topcu. Counter-strategy guided refinement of GR(1) temporal logic specifications. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013* [327], pages 26–33.

[69] M. T. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. *STTT*, 15(5-6):413–431, 2013.

[70] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Comfusy: A tool for complete functional synthesis. In Touili et al. [328], pages 430–433.

[71] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In Grove and Blackburn [329], pages 619–630.

[72] A. Solar-Lezama. Program sketching. *STTT*, 15(5-6):475–495, 2013.

[73] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In Grove and Blackburn [329], pages 229–239.

[74] X. Wang, I. Dillig, and R. Singh. Synthesis of data completion scripts using finite tree automata. *PACMPL*, 1(OOPSLA):62:1–62:26, 2017.

[75] T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In Jagannathan and Sewell [323], pages 221–234.

[76] R. Dimitrova and B. Finkbeiner. Counterexample-guided synthesis of observation predicates. In Jurdzinski and Nickovic [330], pages 107–122.

[77] K. Hsu, R. Majumdar, K. Mallik, and A.-K. Schmuck. Multi-layered abstraction-based controller synthesis for continuous-time systems. In *Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, pages 120–129. ACM, 2018.

[78] K. Mallik, A.-K. Schmuck, S. Soudjani, and R. Majumdar. Compositional abstraction-based controller synthesis for continuous-time systems. *arXiv preprint arXiv:1612.08515*, 2016.

[79] J. Hughes. Generalising monads to arrows. *Science of computer programming*, 37(1):67–111, 2000.

[80] S. Gélineau. FRPzoo - comparing many FRP implementations by reimplementing the same toy app in each. `https://github.com/gelisam/frp-zoo`, 2016.

[81] H. Apfelmus. Threepenny-gui. `https://wiki.haskell.org/Threepenny-gui`, 2013.

[82] A. van der Ploeg. Monadic functional reactive programming. *ACM SIGPLAN Notices*, 48(12):117–128, 2014.

[83] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, pages 159–187. Springer, 2003.

[84] R. Paterson. A new notation for arrows. *ACM SIGPLAN Notices*, 36(10):229–240, 2001.

[85] N. Polikarpova, I. Kuraj, and A. Solar-Lezama. Program synthesis from polymorphic refinement types. In Krintz and Berger [331], pages 522–538.

[86] R. Paterson. A new notation for arrows. In *International Conference on Functional Programming*, pages 229–240. ACM Press, September 2001.

[87] G. Patai. Efficient and compositional higher-order streams. In Mariño [332], pages 137–154.

[88] K. Sawada and T. Watanabe. Emfrp: a functional reactive programming language for small-scale embedded systems. In Fuentes et al. [333], pages 36–44.

[89] D. Winograd-Cort and P. Hudak. Settable and non-interfering signal functions for frp: How a first-order switch is more than enough. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 213–225, New York, NY, USA, 2014. ACM.

[90] N. Sculthorpe and H. Nilsson. Keeping calm in the face of change - towards optimisation of FRP by reasoning about change. *Higher-Order and Symbolic Computation*, 23(2):227–271, 2010.

[91] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia. Reactive synthesis from signal temporal logic specifications. In Girard and Sankaranarayanan [334], pages 239–248.

[92] D. Cyrluk and P. Narendran. Ground temporal logic: A logic for hardware verification. In Dill [335], pages 247–259.

[93] G. Mainland. Why It's Nice to be Quoted: Quasiquoting for Haskell. In *Proceedings of the ACM SIGPLAN workshop on Haskell workshop*, pages 73–82. ACM, 2007.

[94] R. Alur, S. Moarref, and U. Topcu. Compositional synthesis with parametric reactive controllers. In *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*, pages 215–224, 2016.

[95] T. Wongpiromsarn, S. Karaman, and E. Frazzoli. Synthesis of provably correct controllers for autonomous vehicles in urban environments. In *14th International IEEE Conference on Intelligent Transportation Systems, ITSC 2011, Washington, DC, USA, October 5-7, 2011*, pages 1168–1173, 2011.

[96] V. Raman, A. Donzé, D. Sadigh, R. M. Murray, and S. A. Seshia. Reactive synthesis from signal temporal logic specifications. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*, pages 239–248, 2015.

[97] P. Caspi, D. Pilaud, N. Halbwachs, and J. Plaice. Lustre: A declarative language for programming synchronous systems. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987* [336], pages 178–188.

[98] G. Berry and L. Cosserat. The ESTEREL synchronous programming language and its mathematical semantics. In Brookes et al. [337], pages 389–448.

[99] O. S. Initiative et al. Ieee standard systemc language reference manual. *IEEE Computer Society*, pages 1666–2005, 2006.

[100] D. Harel and P. Thiagarajan. Message sequence charts. In *UML for Real*, pages 77–105. Springer, 2003.

[101] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[102] S. Frankau, D. Spinellis, N. Nassuphis, and C. Burgard. Commercial uses: Going functional on exotic trades. *Journal of Functional Programming*, 19(01):27–45, 2009.

[103] L. Cardelli. Type systems. *ACM Comput. Surv.*, 28(1):263–264, March 1996.

[104] M. Broy and K. Stølen. *Specification and Development of Interactive Systems - Focus on Streams, Interfaces, and Refinement*. Monographs in Computer Science. Springer, 2001.

[105] M. Broy. Engineering cyber-physical systems: Challenges and foundations. In Aiguier et al. [338], pages 1–13.

[106] L. Pike, P. Hickey, J. Bielman, T. Elliott, T. DuBuisson, and J. Launchbury. Programming languages for high-assurance autonomous vehicles. In *Proceedings of the ACM SIGPLAN 2014 Workshop on Programming Languages meets Program Verification*, pages 1–2. ACM, 2014.

[107] B. Wymann, E. Espie, and C. Guionneau. Torcs: The open racing car simulator, v1.3.4. `http://torcs.sourceforge.net/index.php`, 2017.

[108] M. R. Bonyadi, S. Nallaperuma, D. Loiacono, and F. Neumann. Simulated car racing championship. `http://cs.adelaide.edu.au/~optlog/SCR2015/index.html`, 2015.

[109] Z. Xu, J. Jiang, and Y. Liu. Experimental research of vehicle-platoon coordination control based on torcs platform. In *Control Conference (CCC), 2016 35th Chinese*, pages 7404–7409. IEEE, 2016.

[110] E. Onieva, D. A. Pelta, J. Alonso, V. Milanés, and J. Pérez. A modular parametric architecture for the TORCS racing engine. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Milano, Italy, 7-10 September, 2009*, pages 256–262, 2009.

[111] L. Cardamone, D. Loiacono, and P. L. Lanzi. Learning drivers for TORCS through imitation using supervised methods. In *Proceedings of the 2009 IEEE Symposium on Computational Intelligence and Games, CIG 2009, Milano, Italy, 7-10 September, 2009*, pages 148–155, 2009.

[112] J. Muñoz, G. Gutiérrez, and A. Sanchis. A human-like TORCS controller for the simulated car racing championship. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG 2010, Copenhagen, Denmark, 18-21 August, 2010*, pages 473–480, 2010.

[113] M. Kamali, L. A. Dennis, O. McAree, M. Fisher, and S. M. Veres. Formal verification of autonomous vehicle platooning. *arXiv preprint arXiv:1602.01718*, 2016.

[114] Z. Kazemi and A. M. Cheng. A scratchpad memory-based execution platform for functional reactive systems and its static timing analysis. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*, pages 176–181. IEEE, 2016.

[115] X. Zou, A. M. Cheng, and Y. Jiang. P-frp task scheduling: A survey. In *Declarative Cyber-Physical Systems (DCPS), CPSWeek Workshop on*, pages 1–8. IEEE, 2016.

[116] A. Voellmy and J. Wang. Scalable software defined network controllers. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 289–290. ACM, 2012.

[117] D. Loiacono, L. Cardamone, and P. L. Lanzi. Simulated car racing championship: Competition software manual. *CoRR*, abs/1304.1672, 2013.

[118] J. McCartney. Rethinking the computer music language: Supercollider. *Computer Music Journal*, 26(4):61–68, 2002.

[119] R. C. Boulanger et al. *The Csound book: perspectives in software synthesis, sound design, signal processing, and programming*. 2000.

[120] M. S. Puckette et al. Pure data. In *ICMC*, 1997.

[121] A. Cypher and D. Halbert. *Watch what I Do: Programming by Demonstration*. MIT Press, 1993.

[122] H. Lieberman. *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[123] S. Gulwani. Synthesis from examples: Interaction models and algorithms. *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012. Invited talk paper.

[124] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012.

[125] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai. A machine learning framework for programming by example. In *ICML (1)*, pages 187–195, 2013.

[126] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[127] P. Osera and S. Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015.

[128] T. E. Murphy. Vivid synth. `vivid-synth.org`, 2018.

[129] P. Masri and A. Bateman. Improved modelling of attack transients in music analysis-resynthesis. In *ICMC*, 1996.

[130] J. Engel, L. Hantrakul, C. Gu, and A. Roberts. Ddsp: Differentiable digital signal processing, 2020.

[131] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, 2013.

[132] Req: Room eq wizard. `http://www.roomeqwizard.com`, 2018. Accessed: 2018-07-08, Version 5.18.

[133] M. A. Casey, R. Veltkamp, M. Goto, M. Leman, C. Rhodes, and M. Slaney. Content-based music information retrieval: Current directions and future challenges. *Proceedings of the IEEE*, 96(4), 2008.

[134] A. Wang. An industrial strength audio search algorithm. ISMIR, 2003.

[135] A. Donzé, R. Valle, I. Akkaya, S. Libkind, S. A. Seshia, and D. Wessel. Machine improvisation with formal specifications. 2014.

[136] S. J. Wright. Coordinate descent algorithms. *Mathematical Programming*, 151(1):3–34, 2015.

[137] T. S. Freeman and F. Pfenning. Refinement types for ML. In Wise [339], pages 268–277.

[138] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *ACM SIGPLAN Notices*, volume 49, pages 269–282. ACM, 2014.

[139] M. Santolucito. Trumpet is to trombone as violin is to ? `https://soundcloud.com/mark-santolucito/sets/trumpet-is-to-trombone-as-violin-is-to`, 2018.

[140] R. Alur, R. Bodik, E. Dallal, D. Fisman, P. Garg, G. Juniwal, H. Kress-Gazit, P. Madusudan, M. Martin, M. Raghothman, S. Saha, S. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and E. Udupa. Syntax-guided synthesis. *Dependable Software Systems Engineering, NATO Science for Peace and Security Series*, 2014.

[141] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama. Sygus-comp 2017: Results and analysis. *CoRR*, abs/1711.11438, 2017.

[142] A. Church. Application of recursive arithmetic to the problem of circuit synthesis. *Journal of Symbolic Logic*, 28(4):289–290, 1963.

[143] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *PoPL'11, January 26-28, 2011, Austin, Texas, USA*, January 2011.

[144] codata - ai completions for your java ide. `https://www.codota.com/`, 2019. Accessed: 2019-09-03.

[145] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow. Deepcoder: Learning to write programs. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*, 2017.

[146] R. Alur, R. Bodik, G. Juniwal, M. M. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[147] M. Raghothaman and A. Udupa. Language to specify syntax-guided synthesis problems. `https://sygus.org/assets/pdf/SyGuS-IF.pdf`, 2019. Accessed: 2019-11-20.

[148] M. Santolucito, W. T. Hallahan, and R. Piskac. Live programming by example. In *Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, CHI 2019, Glasgow, Scotland, UK, May 04-09, 2019.*, 2019.

[149] S. Padhi, T. D. Millstein, A. V. Nori, and R. Sharma. Overfitting in synthesis: Theory and practice. In *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, pages 315–334, 2019.

[150] R. Shin, N. Kant, K. Gupta, C. Bender, B. Trabucco, R. Singh, and D. Song. Synthetic datasets for neural program synthesis. In *International Conference on Learning Representations*, 2019.

[151] R. Alur, D. Fisman, R. Singh, and A. Solar-Lezama. Sygus-comp 2017: Results and analysis. *arXiv preprint arXiv:1711.11438*, 2017.

[152] M. Andrychowicz and K. Kurach. Learning efficient algorithms with hierarchical attentive memory. *arXiv preprint arXiv:1602.03218*, 2016.

[153] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A.-r. Mohamed, and P. Kohli. Robustfill: Neural program learning under noisy i/o. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 990–998. JMLR. org, 2017.

[154] A. Graves, G. Wayne, and I. Danihelka. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

[155] A. Joulin and T. Mikolov. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in neural information processing systems*, pages 190–198, 2015.

[156] Ł. Kaiser and I. Sutskever. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

[157] X. Chen, C. Liu, and D. Song. Learning neural programs to parse programs. *CoRR*, abs/1706.01284, 2017.

[158] J. Devlin, R. R. Bunel, R. Singh, M. Hausknecht, and P. Kohli. Neural program meta-induction. In *Advances in Neural Information Processing Systems*, pages 2080–2088, 2017.

[159] C. Wang, P.-S. Huang, A. Polozov, M. Brockschmidt, and R. Singh. Execution-guided neural program decoding. *arXiv preprint arXiv:1807.03100*, 2018.

[160] R. Bunel, M. Hausknecht, J. Devlin, R. Singh, and P. Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. *arXiv preprint arXiv:1805.04276*, 2018.

[161] A. Kalyan, A. Mohta, O. Polozov, D. Batra, P. Jain, and S. Gulwani. Neural-guided deductive search for real-time program synthesis from examples. *arXiv preprint arXiv:1804.01186*, 2018.

[162] X. Si, Y. Yang, H. Dai, M. Naik, and L. Song. Learning a meta-solver for syntax-guided program synthesis. 2018.

[163] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy. An empirical study on configuration errors in commercial and open source systems. In *23rd ACM Symposium on Operating Systems Principles (SOSP)*, October 2011.

[164] T. Xu and Y. Zhou. Systems approaches to tackling configuration errors: A survey. *ACM Comput. Surv.*, 47(4):70, 2015.

[165] T. Xu, J. Zhang, P. Huang, J. Zheng, T. Sheng, D. Yuan, Y. Zhou, and S. Pasupathy. Do not blame users for misconfigurations. In *24th ACM Symposium on Operating Systems Principles (SOSP)*, November 2013.

[166] T. Xu, L. Jin, X. Fan, Y. Zhou, S. Pasupathy, and R. Talwadker. Hey, you have given me too many knobs!: understanding and dealing with over-designed configuration in system software. In *10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, August 2015.

[167] J. Ryall. Facebook, Tinder, Instagram suffer widespread issues. `http://mashable.com/2015/01/27/facebook-tinder-instagram-issues/`, 2015.

[168] M. Attariyan and J. Flinn. Automating configuration troubleshooting with dynamic information flow analysis. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2010.

[169] Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating systems. In *21st ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.

[170] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: Finding the needle in the haystack. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[171] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy. Early detection of configuration errors to reduce failure damage. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, November 2016.

[172] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning - 16th International Conference, LPAR-16*, pages 348–370, 2010.

[173] R. Piskac, T. Wies, and D. Zufferey. Automating separation logic with trees and data. In *Proceedings of the 26th International Conference on Computer Aided Verification CAV 2014*, pages 711–728, 2014.

[174] F. Bobot, J. Filliâtre, C. Marché, and A. Paskevich. Let's verify this with why3. *STTT*, 17(6):709–727, 2015.

[175] M. Santolucito, E. Zhai, and R. Piskac. Probabilistic automated language learning for configuration files. In *28th Computer Aided Verification (CAV)*, July 2016.

[176] J. Zhang, L. Renganarayana, X. Zhang, N. Ge, V. Bala, T. Xu, and Y. Zhou. Encore: Exploiting system environment and correlation information for misconfiguration detection. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2014.

[177] P. Huang, W. J. Bolosky, A. Singh, and Y. Zhou. Confvalley: A systematic configuration validation framework for cloud services. In *10th European Conference on Computer Systems (EuroSys)*, April 2015.

[178] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, December 2004.

[179] T. Xu. Misconfiguration dataset. `https://github.com/tianyin/configuration_datasets`, March 2017.

[180] R. Agrawal, T. Imieliński, and A. Swami. Mining association rules between sets of items in large databases. In *Acm sigmod record*, volume 22, pages 207–216. ACM, 1993.

[181] A. Zeller. *Why Programs Fail: A Guide to Systematic Debugging*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[182] Stack Overflow. `http://stackoverflow.com/`, March 2017.

[183] PHP CLI Segmentation Fault with pgsql. `http://linux.m2osw.com/php_cli_segmentation_fault_with_pgsql`, March 2017.

[184] Fine-grained value correlation error, March 2017. http://serverfault.com/questions/628414/my-cnf-configuration-in-mysql-5-6-x.

[185] Type Error Example. https://github.com/thekad/puppet-module-mysql/blob/master/templates/my.cnf.erb, March 2017.

[186] The issue for slow query log. http://forum.directadmin.com/showthread.php?t=47547, March 2017.

[187] Fatal Error: Cannot allocate memory for the buffer pool. http://dba.stackexchange.com/questions/25165/intermittent-mysql-crashes-with-error-fatal-error-cannot-allocate-memory March 2017.

[188] J. Han, H. Cheng, D. Xin, and X. Yan. Frequent pattern mining: current status and future directions. *Data Mining and Knowledge Discovery*, 15(1):55–86, 2007.

[189] P. Langley and H. A. Simon. Applications of machine learning and rule induction. *Communications of the ACM*, 38(11):54–64, 1995.

[190] T. Lei, R. Barzilay, and T. Jaakkola. Rationalizing neural predictions. *arXiv preprint arXiv:1606.04155*, 2016.

[191] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from big code. In *ACM SIGPLAN Notices*, volume 50, pages 111–124. ACM, 2015.

[192] A. N. Kolmogorov. Three approaches to the definition of the concept "quantity of information". *Problemy peredachi informatsii*, 1(1):3–11, 1965.

[193] mysetup. https://raw.githubusercontent.com/kazeburo/mysetup/99ba8656f54b1b36f4a7c93941e113adc2f05f70/mysql/my55.cnf, March 2017.

[194] container. https://www.dropbox.com/s/5alc0zs0qp5i529/ybh8r3n2avj7sqd1rcmx0orzry23bopl.cnf?dl=0, March 2017.

[195] isucon2-summer-ruby. https://raw.githubusercontent.com/co-me/isucon2-summer-ruby/1f633384f485fb7282bbbf42f2bf5d18410f7307/config/database/my.cnf, March 2017.

[196] vitroot2. https://www.dropbox.com/s/qcfmsx12i4pjjtd/missing.cnf?dl=0, March 2017.

[197] Aymargeddon. https://raw.githubusercontent.com/bennibaermann/Aymargeddon/b85d23c0690b1c6a48a045ea45f4c8b19b036fa5/var/my.cnf, March 2017.

[198] mini-2011. https://raw.githubusercontent.com/funtoo/experimental-mini-2011/083598863a7c9659f188d31e15b39e3af0f56cab/dev-db/mysql/files/my.cnf, March 2017.

[199] Stats-analysis. https://raw.githubusercontent.com/NCIP/stats-analysis/ec7a1a15b0a5a7518a061aedd2d601ea7cc2dfca/cacoresdk203.2.1/conf/download/my.cnf, March 2017.

[200] evansims. https://raw.githubusercontent.com/evansims/scripts/715e4f4519bbff8bab5ab26a15256d79796c923a/config/mysql/my-2gb.cnf, March 2017.

[201] evansims-script. https://raw.githubusercontent.com/evansims/scripts/715e4f4519bbff8bab5ab26a15256d79796c923a/config/mysql/my-1gb.cnf, March 2017.

[202] Stats-analysis. https://raw.githubusercontent.com/NCIP/stats-analysis/ec7a1a15b0a5a7518a061aedd2d601ea7cc2dfca/cacoresdk203.2.1/conf/download/my.cnf, March 2017.

[203] vitroot. https://raw.githubusercontent.com/vitroot/configs/90441204dbae37521912eaaeedd3574db07b8ae4/my.cnf, March 2017.

[204] vit-analysis. https://www.dropbox.com/s/09joln8kacu9ceq/ekqjat6m1j5nv9ihjhua9q89sid77cso.cnf?dl=00, March 2017.

[205] vps. https://raw.githubusercontent.com/rarescosma/vps/7d0b898bb30eecac65158f704b43bb4d1ca06dbe/_config/mysql/my.cnf, March 2017.

[206] containerization. https://raw.githubusercontent.com/billycyzhang/containerization/78c6e8fefbafb89de8c28296e83a2f6fefe03879/enterprise-images/mariadb/my.cnf, March 2017.

[207] puppet. https://raw.githubusercontent.com/a2o/puppet-modules-a2o-essential/9e48057cc1320de52548ff019352299bc4bd5069/modules/a2o_essential_linux_mysql/files/my.cnf, March 2017.

[208] W. Enck, P. D. McDaniel, S. Sen, P. Sebos, S. Spoerel, A. G. Greenberg, S. G. Rao, and W. Aiello. Configuration management at massive scale: System design and experience. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2007.

[209] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative routing: Extensible routing with declarative queries. In *ACM SIGCOMM (SIGCOMM)*, August 2005.

[210] X. Chen, Y. Mao, Z. M. Mao, and J. E. van der Merwe. Declarative configuration management for complex and dynamic networks. In *ACM CoNEXT (CoNEXT)*, November 2010.

[211] D. Yuan, Y. Xie, R. Panigrahy, J. Yang, C. Verbowski, and A. Kumar. Context-based online configuration-error detection. In *USENIX Annual Technical Conference (USENIX ATC)*, June 2011.

[212] M. Attariyan, M. Chow, and J. Flinn. X-ray: Automating root-cause diagnosis of performance anomalies in production software. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, October 2012.

[213] N. R. Mabroukeh and C. I. Ezeife. A taxonomy of sequential pattern mining algorithms. *ACM Computing Surveys (CSUR)*, 43(1):3, 2010.

[214] J. Ayres, J. Flannick, J. Gehrke, and T. Yiu. Sequential pattern mining using a bitmap representation. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 429–435. ACM, 2002.

[215] M. Santolucito, J. Zhang, E. Zhai, and R. Piskac. Statically verifying continuous integration configurations. *CoRR*, abs/1805.04473, 2018.

[216] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[217] Travis CI. `https://travis-ci.org/`, August 2019.

[218] CircleCI. `https://circleci.com/`, August 2019.

[219] Jenkins. `https://jenkins.io/`, August 2019.

[220] GitLab CI. `https://about.gitlab.com/`, August 2019.

[221] Codefresh. `https://g.codefresh.io/signup?ref=BJV2J4zib`, August 2019.

[222] TeamCity. `https://www.jetbrains.com/teamcity/`, August 2019.

[223] M. Hilton, T. Tunnell, K. Huang, D. Marinov, and D. Dig. Usage, costs, and benefits of continuous integration in open-source projects. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 426–437. ACM, 2016.

[224] R. Pham, L. Singer, O. Liskin, F. Figueira Filho, and K. Schneider. Creating a shared understanding of testing culture on a social coding site. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 112–121. IEEE, 2013.

[225] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov. Quality and productivity outcomes relating to continuous integration in github. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 805–816. ACM, 2015.

[226] G. Gousios, M.-A. Storey, and A. Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 285–296. IEEE, 2016.

[227] M. Leppänen, S. Mäkinen, M. Pagels, V.-P. Eloranta, J. Itkonen, M. V. Mäntylä, and T. Männistö. The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72, 2015.

[228] B. Heim. Supercollider travisci build. TravisCI buildlog, January 2018.

[229] Z. A. Beller M, Gousios G. Oops, my tests broke the build: An analysis of travis ci builds with github. PREPRINT, 2016.

[230] SuperCollider. [sc-dev] 3.9 delayed to tomorrow. Mailing List, January 2018.

[231] S. Baset, S. Suneja, N. Bila, O. Tuncer, and C. Isci. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track*, pages 29–35. ACM, 2017.

[232] M. Raab. Elektra: universal framework to access configuration parameters. *The Journal of Open Source Software*, 1(8):1–2, 2016.

[233] A. Weiss, A. Guha, and Y. Brun. Tortoise: interactive system configuration repair. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 625–636, 2017.

[234] R. Shambaugh, A. Weiss, and A. Guha. Rehearsal: a configuration verification tool for puppet. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 416–430, 2016.

[235] J. M. González-Barahona, A. Hindle, and L. Tan, editors. *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*. IEEE Computer Society, 2017.

[236] d. bbenezech. Railsadmin. `https://github.com/sferik/rails_admin`, 2016. 3fd3b32551e6d2d41cbbc623c99d08656a80be07.

[237] mshibuya. Rails admin. `https://github.com/sferik/rails_admin`, 2016. dca8911f240ea11ebb186c33573188aa9e1b031d.

[238] scambra. Active scaffold. `https://github.com/activescaffold/active_scaffold`, 2016. 10d78ad6ac45a0a55e3c15e12c39d2019aff5146.

[239] scambra. Active scaffold. `https://github.com/activescaffold/active_scaffold`, 2016. 87496cfa09a49d071817ba3da0fa6364c92a5191.

[240] A. Ni and M. Li. Cost-effective build outcome prediction using cascaded classifiers. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 455–458. IEEE, 2017.

[241] T. Wolf, A. Schroter, D. Damian, and T. Nguyen. Predicting build failures using social network analysis on developer communication. In *Proceedings of the 31st International Conference on Software Engineering*, pages 1–11. IEEE Computer Society, 2009.

[242] A. E. Hassan and K. Zhang. Using decision trees to predict the certification result of a build. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 189–198. IEEE, 2006.

[243] G. Forman. An extensive empirical study of feature selection metrics for text classification. *Journal of machine learning research*, 3(Mar), 2003.

[244] D. Gunning. Explainable artificial intelligence (XAI). *Defense Advanced Research Projects Agency (DARPA), nd Web*, 2017.

[245] S. Scott and S. Matwin. Feature engineering for text classification. In *16th International Conference on Machine Learning (ICML)*, June 1999.

[246] K. Chae, H. Oh, K. Heo, and H. Yang. Automatically generating features for learning program analysis heuristics for c-like languages. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):101, 2017.

[247] L. Rokach and O. Maimon. *Data Mining with Decision Trees: Theroy and Applications*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2008.

[248] K. V. R. Paixão, C. Z. Felício, F. M. Delfim, and M. de A. Maia. On the interplay between non-functional requirements and builds on continuous integration. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 479–482, Piscataway, NJ, USA, 2017. IEEE Press.

[249] K. Gallaba, C. Macho, M. Pinzger, and S. McIntosh. Noise and heterogeneity in historical build data: An empirical study of travis ci. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 87–97, New York, NY, USA, 2018. ACM.

[250] jnunemaker. Flipper. `https://github.com/jnunemaker/flipper`, 2019. f4d68e4eb923d1bd6273aa452fbc4dd7146db75f.

[251] jnunemaker. Flipper. `https://github.com/jnunemaker/flipper`, 2019. 9221c50d83e7214fad8f971e5d8d6b76453ebd4f.

[252] B. Cornelissen, A. Zaidman, A. Van Deursen, and B. Van Rompaey. Trace visualization for program comprehension: A controlled experiment. In *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 2009.

[253] R. Wettel, M. Lanza, and R. Robbes. Software systems as cities: A controlled experiment. In *2011 33rd International Conference on Software Engineering (ICSE)*, pages 551–560. IEEE, 2011.

[254] H. B. Mann and D. R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[255] G. A. Barnard. A new test for $2\times$ 2 tables. *Nature*, 156, 1945.

[256] M. Santolucito, E. Zhai, and R. Piskac. Probabilistic automated language learning for configuration files. In *28th Computer Aided Verification (CAV)*, July 2016.

[257] M. Santolucito, E. Zhai, R. Dhodapkar, A. Shim, and R. Piskac. Synthesizing configuration file specifications with association rule learning. *PACMPL*, 1(OOPSLA):64:1–64:20, 2017.

[258] B. Amand, M. Cordy, P. Heymans, M. Acher, P. Temple, and J.-M. Jézéquel. Towards learning-aided configuration in 3d printing: Feasibility study and application to defect prediction. In *Proceedings of the 13th International Workshop on Variability Modelling of Software-Intensive Systems*. ACM, 2019.

[259] O. Tuncer, N. Bila, C. Isci, and A. K. Coskun. Confex: An analytics framework for text-based software configurations in the cloud. Technical report, Tech. Rep. RC25675 (WAT1803-107), IBM Research, 2018.

[260] K. Mazurak and S. Zdancewic. Abash: Finding bugs in bash scripts. In *In ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*, 2007.

[261] Stack Overflow: What is the most difficult/challenging regular expression you have ever written? `http://goo.gl/LLJe0r`. Accessed: 2015-03-24.

[262] Stack Overflow: Auto increment a variable in regex. `http://goo.gl/GPuZP3`. Accessed: 2015-03-25.

[263] Super User: How to batch combine jpeg's from folders into pdf's? `http://goo.gl/LnGYH7`. Accessed: 2015-05-13.

[264] S. Gulwani, M. Mayer, F. Niksic, and R. Piskac. Strisynth: Synthesis for live programming. In *ICSE*, 2015.

[265] S. Burckhardt, M. Fähndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's alive! continuous feedback in ui programming. In *PLDI*, pages 95–104, 2013.

[266] Flash Fill (Microsoft Excel 2013 feature), 2013. `http://research.microsoft.com/users/sumitg/flashfill.html`.

[267] R. Hähnle and M. Huisman. 24 challenges in deductive software verification. In G. Reger and D. Traytel, editors, *ARCADE 2017. 1st International Workshop on Automated Reasoning: Challenges, Applications, Directions, Exemplary Achievements*, volume 51 of *EPiC Series in Computing*. EasyChair, 2017.

[268] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye. *Probability and statistics for engineers and scientists*, volume 5. Macmillan New York, 1993.

[269] J. Sunshine. *Protocol Programmability*. PhD thesis, Pittsburgh, PA, USA, 2013.

[270] S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Commun. ACM*, 55(8):97–105, 2012.

[271] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[272] S. Gulwani. Synthesis from examples: Interaction models and algorithms. In *SYNASC*, 2012.

[273] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5(8), 2012.

[274] C. Scaffidi, B. A. Myers, and M. Shaw. Topes: reusable abstractions for validating data. In *ICSE*, 2008.

[275] R. C. Miller and B. A. Myers. Interactive simultaneous editing of multiple text regions. In *USENIX*, 2001.

[276] T. A. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, 2003.

[277] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Wrangler: interactive visual specification of data transformation scripts. In *CHI*, 2011.

[278] T. Lau. Why programming-by-demonstration systems fail: Lessons learned for usable ai. *AI Magazine*, 30(4):65–67, 2009.

[279] J. Galenson, P. Reames, R. Bodík, B. Hartmann, and K. Sen. CodeHint: dynamic and interactive synthesis of code snippets. In *ICSE*, 2014.

[280] Y. Wei, Y. Hamadi, S. Gulwani, and M. Raghothaman. C# code snippets on-demand, 2014. `http://codesnippet.research.microsoft.com`.

[281] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *Commun. ACM*, 55(2):103–111, 2012.

[282] M. Santolucito and R. Piskac. Formal methods and computing identity-based mentorship for early stage researchers. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2020. To Appear.

[283] M. Fryling, M. Egan, R. Y. Flatland, S. Vandenberg, and S. Small. Catch 'em early: Internship and assistantship cs mentoring programs for underclassmen. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, SIGCSE '18, New York, NY, USA, 2018. ACM.

[284] J. A. Davis, M. A. Clark, D. D. Cofer, A. Fifarek, J. Hinchman, J. A. Hoffman, B. W. Hulbert, S. P. Miller, and L. G. Wagner. Study on the barriers to the industrial adoption of formal methods. In *Formal Methods for Industrial Critical Systems - 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings*, 2013.

[285] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, October 2009.

[286] K. E. Boyer, E. N. Thomas, A. S. Rorrer, D. Cooper, and M. A. Vouk. Increasing technical excellence, leadership and commitment of computing students through identity-based mentoring. In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, SIGCSE '10, New York, NY, USA, 2010. ACM.

[287] J. Burg, V. P. Pauca, W. Turkett, E. Fulp, S. S. Cho, P. Santago, D. Cañas, and H. D. Gage. Engaging non-traditional students in computer science through socially-inspired learning and sustained mentoring. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, SIGCSE '15, New York, NY, USA, 2015. ACM.

[288] E. Wenger, R. A. McDermott, and W. Snyder. *Cultivating communities of practice: A guide to managing knowledge*. Harvard Business Press, 2002.

[289] D. Gürer and T. Camp. An acm-w literature review on women in computing. *ACM SIGCSE Bulletin*, 2002.

[290] B. DiSalvo, M. Guzdial, A. Bruckman, and T. McKlin. Saving face while geeking out: Video game testing as a justification for learning computer science. *Journal of the Learning Sciences*, 2014.

[291] A. J. Ko and K. Davis. Computing mentorship in a software boomtown: Relationships to adolescent interest and beliefs. In *Proceedings of the 2017 ACM Conference on International Computing Education Research*, ICER '17, New York, NY, USA, 2017. ACM.

[292] Y. Kafai, J. Griffin, Q. Burke, M. Slattery, D. Fields, R. Powell, M. Grab, S. Davidson, and J. Sun. A cascading mentoring pedagogy in a cs service learning course to broaden participation and perceptions. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, New York, NY, USA, 2013. ACM.

[293] R. Tashakkori, J. T. Wilkes, and E. G. Pekarek. A systemic mentoring model in computer science. In *Proceedings of the 43rd Annual Southeast Regional Conference - Volume 1*, ACM-SE 43, New York, NY, USA, 2005. ACM.

[294] N. Thomas. Mentoring | stars computing corps. `https://www.starscomputingcorps.org/mentoring`. Accessed: 2019-08-28.

[295] D. Babic, S. Bucur, Y. Chen, F. Ivancic, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang. FUDGE: fuzz driver generation at scale. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019.*, 2019.

[296] J. Backes, P. Bolignano, B. Cook, C. Dodge, A. Gacek, K. Luckow, N. Rungta, O. Tkachuk, and C. Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, 2018.

[297] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn. Scaling static analyses at facebook. *Commun. ACM*, 2019.

[298] C. Hawblitzel, J. Howell, M. Kapritsos, J. R. Lorch, B. Parno, M. L. Roberts, S. T. V. Setty, and B. Zill. Ironfleet: proving safety and liveness of practical distributed systems. *Commun. ACM*, 2017.

[299] L. C. Paulson. Inductive analysis of the internet protocol TLS. *CoRR*, 2019.

[300] E. Zhai, R. Piskac, R. Gu, X. Lao, and X. Wang. An auditing language for preventing correlated failures in the cloud. *PACMPL*, (OOPSLA), 2017.

[301] L. D'Antoni, D. Kini, R. Alur, S. Gulwani, M. Viswanathan, and B. Hartmann. How can automatic feedback help students construct automata? *ACM Trans. Comput.-Hum. Interact.*, 2015.

[302] W. T. Hallahan, A. Xue, M. T. Bland, R. Jhala, and R. Piskac. Lazy counterfactual symbolic execution. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019*, 2019.

[303] A. Ignatiev, A. Morgado, N. Narodytska, and V. Manquinho. Sat/smt/ar summer school 2019. `https://reason.di.fc.ul.pt/ssa-school-2019/`. Accessed: 2019-08-28.

[304] C. Enea, V. Manquinho, and R. Piskac. Vmcai winter school 2019. `http://vmcaischool19.tecnico.ulisboa.pt/`. Accessed: 2019-08-28.

[305] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[306] J. Eve. `http://www.code441.com/`, 2019. Accessed: 2019-11-30.

[307] W. T. Hallahan, A. Xue, and R. Piskac. G2Q: haskell constraint solving. In *Proceedings of the 12th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2019, Berlin, Germany, August 18-23, 2019*, 2019.

[308] A. Reinking and R. Piskac. A type-directed approach to program repair. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA*, 2015.

[309] D. Hsu, N. M. Amato, S. Berman, and S. A. Jacobs, editors. *Robotics: Science and Systems XII, University of Michigan, Ann Arbor, Michigan, USA, June 18 - June 22, 2016*, 2016.

[310] K. Chatterjee, R. Ehlers, and S. Jha, editors. *Proceedings 3rd Workshop on Synthesis, SYNT 2014, Vienna, Austria, July 23-24, 2014*, volume 157 of *EPTCS*, 2014.

[311] H. Boehm and C. Flanagan, editors. *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. ACM, 2013.

[312] ACM. *Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, SCAV@CPSWeek 2017, Pittsburgh, PA, USA, April 21, 2017*. ACM, 2017.

[313] S. L. P. Jones, M. Tofte, and A. M. Berman, editors. *Proceedings of the 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP '97), Amsterdam, The Netherlands, June 9-11, 1997*. ACM, 1997.

[314] G. Mainland, editor. *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*. ACM, 2016.

[315] E. Visser and Y. Smaragdakis, editors. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 - November 4, 2016*. ACM, 2016.

[316] IEEE. *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977.

[317] G. Ausiello, M. Dezani-Ciancaglini, and S. R. D. Rocca, editors. *Automata, Languages and Programming, 16th International Colloquium, ICALP89, Stresa, Italy, July 11-15, 1989, Proceedings*, volume 372 of *Lecture Notes in Computer Science*. Springer, 1989.

[318] K. Fisher and J. H. Reppy, editors. *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. ACM, 2015.

[319] D. Janin and M. Sperber, editors. *Proceedings of the 4th International Workshop on Functional Art, Music, Modelling, and Design, FARM@ICFP 2016, Nara, Japan, September 24, 2016*. ACM, 2016.

[320] R. Majumdar and V. Kuncak, editors. *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*, volume 10427 of *Lecture Notes in Computer Science*. Springer, 2017.

[321] M. Bezem, editor. *Computer Science Logic, 25th International Workshop / 20th Annual Conference of the EACSL, CSL 2011, September 12-15, 2011, Bergen, Norway, Proceedings*, volume 12 of *LIPIcs*. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2011.

[322] K. Claessen and N. Swamy, editors. *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA, January 24, 2012*. ACM, 2012.

[323] S. Jagannathan and P. Sewell, editors. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014.

[324] G. Morrisett and T. Uustalu, editors. *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*. ACM, 2013.

[325] K. L. McMillan and X. Rival, editors. *Verification, Model Checking, and Abstract Interpretation - 15th International Conference, VMCAI 2014, San Diego, CA, USA, January 19-21, 2014, Proceedings*, volume 8318 of *Lecture Notes in Computer Science*. Springer, 2014.

[326] IEEE. *Formal Methods in Computer-Aided Design, FMCAD 2014, Lausanne, Switzerland, October 21-24, 2014*. IEEE, 2014.

[327] *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*. IEEE, 2013.

[328] T. Touili, B. Cook, and P. B. Jackson, editors. *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*. Springer, 2010.

[329] D. Grove and S. Blackburn, editors. *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. ACM, 2015.

[330] M. Jurdzinski and D. Nickovic, editors. *Formal Modeling and Analysis of Timed Systems - 10th International Conference, FORMATS 2012, London, UK, September 18-20, 2012. Proceedings*, volume 7595 of *Lecture Notes in Computer Science*. Springer, 2012.

[331] C. Krintz and E. Berger, editors. *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. ACM, 2016.

[332] J. Mariño, editor. *Functional and Constraint Logic Programming - 19th International Workshop, WFLP 2010, Madrid, Spain, January 17, 2010. Revised Selected Papers*, volume 6559 of *Lecture Notes in Computer Science*. Springer, 2011.

[333] L. Fuentes, D. S. Batory, and K. Czarnecki, editors. *Companion Proceedings of the 15th International Conference on Modularity, Málaga, Spain, March 14 - 18, 2016*. ACM, 2016.

[334] A. Girard and S. Sankaranarayanan, editors. *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*. ACM, 2015.

[335] D. L. Dill, editor. *Computer Aided Verification, 6th International Conference, CAV '94, Stanford, California, USA, June 21-23, 1994, Proceedings*, volume 818 of *Lecture Notes in Computer Science*. Springer, 1994.

[336] *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*. ACM Press, 1987.

[337] S. D. Brookes, A. W. Roscoe, and G. Winskel, editors. *Seminar on Concurrency, Carnegie-Mellon University, Pittsburg, PA, USA, July 9-11, 1984*, volume 197 of *Lecture Notes in Computer Science*. Springer, 1985.

[338] M. Aiguier, Y. Caseau, D. Krob, and A. Rauzy, editors. *Complex Systems Design & Management, Proceedings of the Third International Conference on Complex Systems Design & Management CSD&M 2012, Paris, France, December 12-14, 2012*. Springer, 2013.

[339] D. S. Wise, editor. *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI), Toronto, Ontario, Canada, June 26-28, 1991*. ACM, 1991.