DANIEL NEIDER

INTELLIGENT FORMAL METHODS

COMBINING DEDUCTIVE AND INDUCTIVE REASONING TO BUILD RELIABLE SYSTEMS

HABILITATION
JUNE 2022

UNIVERSITY OF KAISERSLAUTERN

Vom Fachbereich Informatik der Technischen Universität Kaiserslautern im Rahmen des Habilitationsverfahrens akzeptierte Habilitationsschrift

To my parents and partner, my greatest supporters

CONTENTS

1 INTRODUCTION

Information technology has become an indispensable part of our daily lives, with a significant proportion of our everyday activities relying on the safe and reliable operation of computer systems. However, building safe and reliable systems is notoriously difficult. It involves a host of different tasks, ranging from formalizing the system's requirements, to designing and implementing the system, to verifying that it indeed meets its specification. Each of these tasks is highly complex and—even worse—prone to error.

A promising approach to ensuring safety and reliability of computer systems is the use of so-called *formal methods*, a broad range of rigorous, mathematical techniques for specifying, developing, and verifying hardware, software, cyber-physical systems, and artificial intelligence (e.g., see Huisman, Gurov, and Malkis [85] for an overview). Unlike more traditional quality assurance approaches, such as testing, formal methods offer the unique ability to provide formal proof of the absence of errors, a trait particularly desirable in the context of today's ubiquitous safety-critical systems. However, this advantage comes at a cost: formal methods require extensive training, often assume idealized or limited settings, and typically demand substantial computational resources (see corresponding textbooks for an overview [26, 42, 74]). Unfortunately, this cost can outweigh the expected benefits [147].

This work reports on our research to fundamentally simplify the use of formal methods. Similar to the vision of artificial intelligence, our overall goal is to automate formal methods and dramatically expand their applicability. To achieve this goal, we develop an innovative, novel type of formal methods, which combines inductive techniques from the area of machine learning and deductive techniques from the area of logic. We name this new type ***intelligent formal methods***.

In the remainder of this work, we develop intelligent formal methods for three of the most critical challenges in the development of safe and reliable systems:

- In Chapter 2, we address the problem of software verification [42, 67, 82], which is the task of proving that a given piece of software satisfies its specification. In particular, we develop novel methods to learn correctness proofs (in the form of invariants and method contracts) for a wide range of software, including recursive programs, concurrent programs, and programs that use dynamically allocated data structures.
- In Chapter 3, we address the problem of synthesizing both hardware and software from formal specifications. More specifically, we develop learning-based algorithms

for two important synthesis settings: reactive synthesis [51] (i.e., the task of synthesizing systems that continuously interact with their environment) as well as functional synthesis [18] (i.e., the task of synthesizing implementations of functions, including loop-free code). Additionally, we develop a general framework that provides the vocabulary needed to better understand the similarities and differences between alternative learning-based synthesis algorithms.

- In Chapter 4, we develop novel techniques that fundamentally improve the way engineers write formal specifications. In particular, we show how machine learning can be used to infer specifications expressed in Linear Temporal Logic [125] or the Property Specification Language [61] from examples describing the desired (and undesired) behavior of a system.

We conclude in Chapter 5 with a summary of our work, a discussion of open challenges in the area of (intelligent) formal methods, and an outlook on directions for future research.

Finally, it is important to mention that this document serves as an extended abstract summarizing our research in the area of intelligent formal methods. It is not meant to be an entirely original work but a unified compilation of a long series of our research papers. For the sake of conciseness, we have omitted proofs and all empirical evaluations, which show that all of our techniques are highly competitive. To provide the necessary context, we regularly refer to specific results in our original research papers and encourage the reader to consult this additional material for further details. Moreover, Chapters 2 to 4 conclude with notes on related work. For the reader's convenience, the bibliography of this work is partitioned into a primary bibliography, listing the papers compiled into this document, and a secondary bibliography, containing all other references.

2 INTELLIGENT SOFTWARE VERIFICATION

Automated program verification dates back to the 1960s when Floyd [67] and Hoare [82] pioneered the idea of assigning a formal meaning to programs, thereby providing an effective way to reason about computer programs rigorously. Since then, the field of automated verification has made significant progress, and a host of different verification techniques have been developed. Examples include abstract interpretation [54], deductive verification (which builds upon Floyd and Hoare's work as well as the weakest pre-condition calculus by Dijkstra [55]), software model checking [36, 84], and symbolic execution [97], to name but a few.

In this work, we focus on *deductive verification*. To illustrate this approach, consider a prototypical program P, given by

```
assume α;
while (g) {
    s;
}
assert β;
```

where α and β are formulas over the configurations[1] of the program, g is a Boolean expression (the *loop guard*), and s is a program snippet (i.e., a sequence of statements) representing the *loop body*. The assume and assert statements serve as the specification of P: the assume statement represents a *pre-condition* (i.e., a condition that has to hold before the code is executed), while the assert statement corresponds to a *post-condition* (i.e., a condition that has to hold after the code is executed). The goal of deductive verification is now to prove that every execution of P that satisfies the pre-condition and terminates also satisfies the post-condition. This task is referred to as *proving the program correct* or *program correctness* for short.

On a technical level, the core idea of deductive verification is to translate a program and its specification into a set of logic formulas, called *verification conditions (VCs)*, such that the verification conditions are valid if and only if the program satisfies the specification. Subsequently, highly optimized constraint solvers, such as CVC4/

[1] To avoid too much notational overhead, we assume a program configuration to be a vector that records the values of the program's variables, the content of the heap, and (for technical reasons that become apparent later) the current location in the code.

CVC5 [28, 31] or Z3 [114], can be used to check the validity of the verification conditions in a fully or semi-automated fashion (by checking (un-)satisfiability of the negated verification conditions). Moreover, if the program does not satisfy its specification, modern constraint solvers are able to return inputs to the program that witness a specification violation. Programmers can then use these inputs to locate and fix the detected errors.

In general, however, deductive verification requires additional guidance. Take the program P from above as an example: since the exact number of loop iterations is not known in general and might change depending on the program's input, the user needs to provide a so-called loop invariant, which summarizes the effect of the loop and abstracts from the actual number of times the loop is executed. More formally, a *loop invariant* is a formula γ over the configurations of a program that has to satisfy three properties:

1. the pre-condition α implies γ;
2. the invariant γ and the negation of the loop guard b imply the post-condition β; and
3. the invariant is *inductive*, meaning that if γ and the loop guard hold at the beginning of the loop, then γ also needs to hold at the end of the loop (i.e., after executing the loop body).

Once a loop invariant is provided, a constraint solver (or theorem prover) can be used to establish whether the program P is correct.

Unfortunately, the generation of loop invariants remains a highly challenging, manual task. Although programmers often have adequate invariants in mind while writing their code, formalizing them in a logical formalism is error-prone and requires substantial training and expertise. In fact, the lack of automated means to generate loop invariants is one of the significant hurdles that prevent a widespread adaptation of deductive verification in practice. As a consequence, intensive research has been devoted to this topic, with varying degrees of automation and success (we refer the reader to Garg et al. [69, 3] for an in-depth discussion of related work).

One of the most promising developments towards a fully automated generation of loop invariants is the use of machine learning, specifically a framework called *ICE learning* [69].[2] To make this framework precise, let us fix a program P, and let C_P denote the set of all configurations that can arise while executing P. Given a formula φ and a program configuration $c \in C_P$, we then write $c \models \varphi$ if c satisfies φ, which is defined in the usual way (e.g., see Bradley and Manna [42]). To ease the presentation in

[2]The name "ICE learning" arises from so-called implication counterexamples, which are the defining feature of the framework.

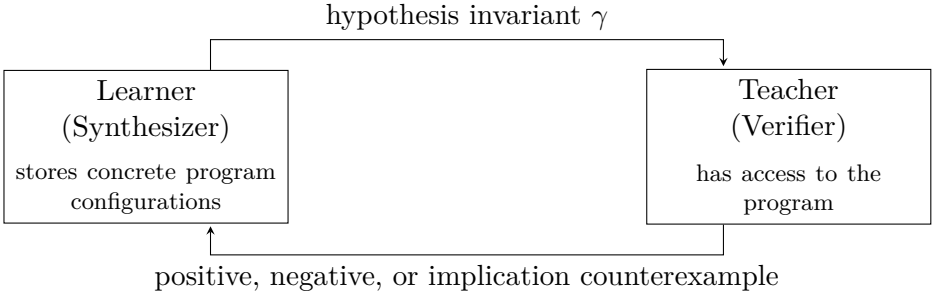


Figure 2.1: The ICE learning framework [69].

the remainder, we assume that the satisfaction of a formula φ can either be determined directly from the program configurations or that the program is instrumented with ghost variables tracking its satisfaction at relevant places in the program (e.g., at the loop header and immediately after the loop). Moreover, we lift this notation to formulas involving Boolean program expression (e.g., we write $c \models \varphi \wedge g$ if c satisfies φ and the loop guard g).

The general workflow of ICE learning is shown in Figure 2.1. It follows the principle of *counterexample-guided inductive synthesis (CEGIS)* [142] and consists of a feedback loop with two entities: a *learner* (or synthesizer), who is agnostic to the program under consideration, and a *teacher* (or verifier), who is able to reason symbolically about the program.[3]

In each round of the feedback loop, the learner proposes a candidate invariant γ based on the information it has gathered so far. The teacher receives this hypothesis and checks whether it is an adequate invariant that is sufficient to prove the program correct. To this end, the teacher can employ any off-the-shelf deductive verifier, such as Boogie [29], SeaHorn [80], or why3 [65]. If the verification succeeds, the feedback loop terminates, and the teacher reports that the program is correct. If the verification fails, however, the teacher is required to return a so-called *counterexample*, which guides the learner towards an invariant that proves the program correct. The type of counterexample depends on which of the three defining properties of an invariant is violated:

1. If the pre-condition α does not imply γ, the teacher returns a so-called *positive counterexample* $c \in C_P$ such that $c \models \alpha \wedge \neg\gamma$. The meaning of such a counterexample is that any future hypothesis must satisfy c.

[3]The ideas underlying CEGIS date back to the 1970s, when Biermann and Feldman [37] proposed a similar concept in the context of automata learning.

2. If the hypothesis γ and the negation of the loop guard g do not imply the post-condition β, the teacher returns a so-called *negative counterexample* $c \in C_P$ such that $c \models \gamma \wedge \neg g \wedge \neg\beta$. The meaning of such a counterexample is that any future hypothesis must not satisfy c.

3. If the hypothesis H is not inductive, then the teacher returns a so-called *implication counterexample* $c \to c' \in C_P \times C_P$ such that $c \models \gamma \wedge g$, the program state c' is reached from c by executing the loop body, and $c' \not\models \gamma$. The meaning of such a counterexample is that any future hypothesis that satisfies c must also satisfy c'. In contrast to positive and negative counterexamples, however, the classification of neither c nor c' is fixed; instead, the learner is free to choose their classifications as long as the implication constraint is satisfied.

This process continues until the learner has learned an invariant that is sufficient to prove the program correct. Note that counterexamples consist of concrete program configurations, which provide a generic interface to data-driven machine learning techniques.

In contrast to many other approaches that employ machine learning for verification, ICE learning has three outstanding features, which contribute significantly to its success. First, ICE learning provides formal correctness guarantees in that the inferred invariants indeed prove that the given programs satisfy their specifications. Second, ICE learning is guaranteed to terminate in many practical settings (e.g., when the class of potential invariants is finite); once this happens, the feedback loop either returns a suitable invariant or reports that no invariant exists. Third, ICE learning allows the reuse of existing verification infrastructure, thus benefiting from any advances in deductive verification.

However, despite these advantages and its general success, ICE learning also has various drawbacks:

- Although ICE learning—in general—permits synthesizing loop invariants at multiple locations in a program, these invariants must be independent of each other (which essentially amounts to synthesizing one invariant at a time). In particular, this restriction excludes programs with nested loops and programs in which several loops occur in sequence. Moreover, the ICE learning framework cannot synthesize contracts for recursive functions (i.e., their pre-conditions and post-conditions), which is required for fully automated verification.

- Due to the combinatorial nature of implication counterexamples, the original ICE learning framework [69] uses constraint solving to implement the learner. Since constraint solving is computationally hard, this approach is significantly less scalable than statistical machine learning algorithms.

- To implement a teacher and extract counterexamples, ICE learning requires that the verification conditions fall into a decidable logic. However, this is often not the case, specifically when programs manipulate dynamically allocated data structures, such as arrays, lists, and trees.

In this work, we address all three of these limitations. More specifically, Section 2.1 generalizes the ICE learning framework to a fully automated verification setting where both loop invariants and function contracts can be synthesized simultaneously. This generalization, named Horn-ICE learning, also provides an effective way to synthesize assume-guarantee contracts for the verification of concurrent programs [2], which the original ICE learning cannot. In Section 2.2, we then present two novel Horn-ICE learning algorithms, which build on top of popular machine learning techniques and provide a substantially increased performance compared to the original, constraint-based approach. Finally, Section 2.3 shows how to automatically generate invariants and method contracts for programs whose verification conditions do not fall into a decidable logic. To this end, we develop a modification of the Horn-ICE framework that permits interfacing with sound-but-incomplete verification engines. We conclude this chapter in Section 2.4 with a discussion of related work.

2.1 Horn-ICE Learning

In this section, we present a generalization of the ICE learning framework, named *Horn-ICE learning* [2], that allows synthesizing multiple, interdependent *annotations* (i.e., loop invariants and function contracts). To illustrate the need for such a generalization, let us consider the following program P

```
int func1(int x)
requires α1;
ensures β1;
{
    s1;
    int y = func2(x);
    s2;
    return y;
}
```

```
int func2(int x)
requires α2;
ensures β2;
{
    s3;
    return x;
}
```

where α_1, α_2 and β_1, β_2 are the pre-conditions and post-conditions of the two functions `func1` and `func2`, respectively, and s_1, s_2, s_3 are program snippets (which modify the

variables in the current scope and might or might not contains loops). The annotations $\alpha_1, \alpha_2, \beta_2, \beta_2$ serve as contracts for both functions and enable us to verify each function individually (by replacing each function call with a sequence of statements that assert the pre-condition and then assume the post-condition). Note that such a compositional approach to deductive verification is routinely used in practice because it reduces the size of the individual verification conditions and permits the verification of multiple functions in parallel. Moreover, compositional verification is essential in situations where function calls cannot be inlined (say due to recursion or deep nesting) or where the program involves concurrency (in which case reasoning in the form of rely-guarantee contracts is typically used [152]).

Let us now assume a fully automated verification setting, where the function contracts are not given by the user but need to be synthesized. To understand why this is a hard problem in general, consider one of the functions in P, say `func2`. For the sake of simplicity, let us moreover suppose that `func2` is a leaf function in the sense that it does not call other functions. Already in this simple situation, it is hard to determine what contract to generate for `func2`, especially without looking at its clients. There are many trivial contracts (e.g., $\alpha_2 = \beta_2 = \mathit{true}$), while other may be useless for the client to prove its assertions. In general, there is also no "most useful" contract or, even if it exists, it may be inexpressible in the logic or too expensive and unnecessary for the program's verification. These observations show that the correctness properties being verified (i.e., the assertions in the program) should somehow dictate the granularity of the contracts.

To capture such interdependencies between different parts of a program, the concept of constrained Horn clauses has emerged as a general and robust formalism for expressing verification conditions [75, 80]. Consider, for instance, the task of verifying the function `func1` of the program P above. Then, one of the verification conditions that we want to be valid is

$$\big[\alpha_1(c_1) \wedge \mathit{trans}_{s_1}(c_1, c_2) \wedge \beta_2(c_2) \wedge \mathit{trans}_{s_2}(c_2, c_3)\big] \rightarrow \beta_1(c_3)$$

where $\mathit{trans}_s(c, c')$ is a binary relation capturing the semantics of a snippet s in terms of how it transforms a program configuration c into the configuration c'. Note that this verification condition is indeed in the form of a Horn clause: the antecedent of the implication is a conjunction of positive (i.e., unnegated) literals, while the consequent is a single positive literal.[4] Moreover, Horn clauses are implicitly universally quantified in the sense that they are required to be satisfied for all possible program configurations.

[4] Alternatively, one can define Horn clauses as disjunctions of literals with at most one positive literal (i.e., all literals, except for at most one, appear negated). Throughout this work, however, we view Horn constraints as implications since this is a more natural way to represent verification conditions.

Since we are interested in a fully automated verification, all four formulas $\alpha_1, \alpha_2, \beta_1, \beta_2$ need to be synthesized. When a learner proposes concrete formulas, the teacher checking the above verification condition may find it to be invalid and produces concrete programs configurations $c_1, c_2, c_3 \in C_P$ that violate the Horn clause. Thus, the teacher has to inform the learner of the reason why the verification failed and needs to convey that

$$\text{if } c_1 \models \alpha_1 \text{ and } c_2 \models \beta_2, \text{ then } c_3 \models \beta_1$$

has to hold. Notice, however, that this information cannot be formulated as an implication counterexample, and the most natural constraint to return to the learner is indeed in the form of a Horn clause. This observation has motivated us to develop a generalization of the ICE learning framework, named Horn-ICE learning, in which implication counterexamples are replaced with Horn clauses. In fact, a very similar modification has been proposed independently by Champion et al. [48] as a means to discover refinement types for higher-order functional programs.

The Horn-ICE framework [2] is shown in Figure 2.2. It follows ICE learning closely but has two distinct differences:

- The learner now has to be able to synthesize multiple, interdependent annotations simultaneously.
- Implication counterexamples are replaced by *Horn counterexamples* $(c_1 \wedge \cdots \wedge c_n) \rightarrow c$ where $c_1, \ldots, c_n, c \in C_P$ are program configurations. To simplify our presentation, we do not explicitly record the information which function contracts or invariants a Horn counterexample relates. Instead, we include a program counter in every program configuration that tracks which location in the code and, hence, to which invariant or function contract a specific configuration refers. The omitted information can then easily be recovered from the configurations themselves.

Note that the Horn-ICE learning framework is, in fact, a generalization of ICE learning since every implication counterexample is a Horn counterexample with a single program configuration occurring in the antecedent. Like in implication counterexamples, the program configurations in Horn counterexamples are not labeled (i.e., they are neither positive nor negative), and the learner has to determine suitable labels in a way that satisfies the Horn constraint. These logical constraints make Horn-ICE learning substantially more challenging than the classification tasks usually considered in machine learning.

Finally, let us briefly discuss the termination of the Horn-ICE framework. Since every non-trivial semantic property of a program is undecidable in general (which is a famous result by Rice [131]), we can, of course, not hope that the feedback-loop in Figure 2.2

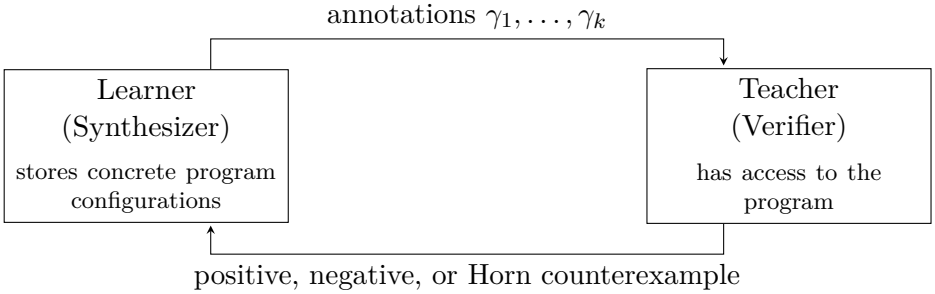


Figure 2.2: The Horn-ICE learning framework [2].

always terminates. However, we discuss several practically relevant settings in the remainder of this chapter in which our framework is guaranteed to terminate. It then either returns an invariant or reports that no invariant exists.

In the next section, we present two novel learning algorithms that are specifically tailored to the Horn-ICE learning framework. In contrast to earlier ICE learning algorithms, which predominantly rely on constraint solving [69], both algorithms draw their inspiration from modern machine learning and, hence, can handle large amounts of data (i.e., counterexamples) effectively.

2.2 Two Efficient Horn-ICE Learning Algorithms

The distinguishing feature of Horn-ICE learning, which sets it apart from many other machine learning settings, is that it introduces the need for symbolic reasoning into the learning process: remember that the program states appearing in Horn counterexamples are unlabeled, and a learning algorithm has to label them such that the Horn constraints are satisfied. Thus, the two learning algorithms we design in the course of this section combine inductive and deductive reasoning to account for this requirement. Both algorithms rely on common concepts, which we introduce first. To simplify our presentation, let us assume that the learner has to synthesize a single formula γ. All challenges that arise for Horn-ICE learning already manifest in this simplified setting, and both learning algorithms can be lifted to the case of multiple, interdependent annotations in a straightforward (yet technical) manner.

Given a program P, we assume that the learner stores the counterexamples in a tuple $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$ where S_+ is a finite set of positive examples, S_- of negative examples, and $S_\Rightarrow$ is a finite set of Horn counterexamples. We call this tuple a *Horn-ICE sample* and drop the prefix "Horn-ICE" if it is clear from the context.

We denote the membership of a program configuration c in a Horn-ICE sample $\mathcal{S}$ by $c \in \mathcal{S}$, meaning that c is an element of S_+ or S_-, or it appears in a Horn counterexample in $S_\Rightarrow$. Moreover, we define the *size* of a sample by $|\mathcal{S}| = |S_+|+|S_-|+\sum_{(c_1\wedge\cdots\wedge c_n)\to c\in S_\Rightarrow} n+1$. Note that a Horn-ICE sample is always a finite object since every individual program configuration is finite, and every iteration of the feedback loop returns a finite number of configurations. For this section and the next, we encourage the reader to think of programs over numeric data types, where a program configuration is a finite vector of numeric values.

Given a Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$, the task of the learner is to generate a formula γ that is *consistent with* $\mathcal{S}$ in the sense that

- $c \models \gamma$ for each positive counterexample $c \in S_+$;
- $c \not\models \gamma$ for each negative counterexample $c \in S_-$; and
- each Horn counterexample $(c_1 \wedge \cdots \wedge c_n) \to c \in S_\Rightarrow$ is satisfied, meaning that $c_i \models \gamma$ for each $i \in \{1, \ldots, n\}$ implies $c \models \gamma$.

Note that this learning task has two important differences to the standard machine learning setting. First, our goal is to learn an object that can be used as an annotation for deductive verification (e.g., a formula or any other object that can be translated into one). Neural networks, or other statistical machine learning models, are typically too complex to reason about algorithmically and, hence, are not suited for this purpose. Second, unlike in machine learning, where the goal is typically to minimize the number of errors, we cannot allow the formula γ to make even a single mistake. If we allowed mistakes, the teacher could repeatedly return the same counterexample, thereby preventing the learner from making any progress.

The remainder of this section presents two novel Horn-ICE learning algorithms in detail. In Section 2.2.1, we develop *Sorcar*, an algorithm for learning invariants in the form of conjunctions over a fixed set of predicates. Although this class of invariants seems restrictive, it turns out that generating conjunctive invariants is highly effective in practice and used in various industry-strength verification tools, such as Microsoft's Static Driver Verifier [102]. In Section 2.2.2, we then present ICE-DT. This algorithm is based on decision trees and can generate general formulas (in disjunctive normal form).

2.2.1 Sorcar

Our first Horn-ICE learning algorithm, named Sorcar [10], is designed to learn conjunctive invariants over a fixed set $\mathcal{P}$ of predicates. This design choice is motivated by two observations: first, for a wide range of domains of programs and types of specifications,

it is possible to identify classes of candidate predicates that are typically involved in invariants (e.g., based on the code of the programs and typical properties of the specification); second, a conjunction over these predicates is often sufficient to prove a given program correct. In fact, various industry-strength verification tools use this exact approach, including Microsoft's Static Driver Verifier [102] (which is build on top of Corral [103]) and GPUVerify [33], a tool for checking race-freedom of GPU kernels.

To ease the presentation throughout this section, we use conjunctions $p_1 \wedge \cdots \wedge p_n$ of predicates over $\mathcal{P}$ and the corresponding sets $\{p_1, \ldots, p_n\} \subseteq \mathcal{P}$ interchangeably. In particular, for a subset $X = \{p_1, \ldots, p_n\} \subseteq \mathcal{P}$ of predicates and a program configuration $c \in C_P$, we write $c \models X$ if and only if $c \models p_1 \wedge \cdots \wedge p_n$. Moreover, we call a set X of predicates *consistent* with a Horn-ICE sample $\mathcal{S}$ if and only if the conjunction $\bigwedge_{p \in X} p$ is consistent with $\mathcal{S}$.

Sorcar uses a modification of the well-known *elimination algorithm* [95], which we have adapted to the Horn-ICE framework as described next. Given a Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$, the modified elimination algorithm first initializes a set $X = \mathcal{P}$ containing all predicates. Then, it repeats the following three steps until a fixed point is reached:

1. The modified elimination algorithm removes all predicates p from X that violate a positive counterexample (i.e., there exists a positive counterexample $c \in S_+$ such that $c \not\models p$). The result is the unique largest set X of predicates that is consistent with S_+ (i.e., $c \models X$ holds for each $c \in S_+$).
2. The modified elimination algorithm checks whether all Horn counterexamples are satisfied. If a Horn counterexample $(c_1 \wedge \cdots \wedge c_n) \rightarrow c \in S_\Rightarrow$ is not satisfied, it means that each program configuration on the left-hand side satisfies X, but the one on the right-hand side does not. Note, however, that X corresponds to the semantically smallest set of program configurations expressible by a conjunctive formula consistent with S_+. Moreover, $c_1, \ldots, c_n$ satisfy X. Thus, the right-hand side c necessarily has to satisfy X as well (otherwise X would not satisfy this Horn counterexample). The elimination algorithm adds c as a new positive counterexample to S_+ to account for this.
3. The elimination algorithm repeats Steps 1 and 2 until a fixed point is reached. Once this happens, X is the unique largest set of predicates that is consistent with S_+ and $S_\Rightarrow$.

Finally, the modified elimination algorithm checks whether each negative counterexample violates X (i.e., $c \not\models X$ for each $c \in S_-$). If this is the case, then X is the largest set of predicates that is consistent with $\mathcal{S}$; otherwise, no consistent conjunction over $\mathcal{P}$ exists.

It is not hard to verify that the runtime of the modified elimination algorithm is polynomial in the number of predicates and the size of the Horn-ICE sample, provided predicates can be evaluated in constant time. Moreover, when used as a learner in the context of the Horn-ICE framework, it converges to a conjunctive invariant in at most $|\mathcal{P}|$ iterations (since at least one predicate is removed in each iteration of the feedback loop) or reports that no conjunctive invariant over $\mathcal{P}$ exists. Note that the guarantee to converge in at most $|\mathcal{P}|$ rounds is of great importance in practice: on the one hand, each interaction with the teacher is computationally expensive because it involves one (or more) invocations of a contraint solver; on the other hand, the search space of potential invariants consists of $2^{|\mathcal{P}|}$ semantically distinct conjunctions, and the set $\mathcal{P}$ can contain hundreds (or even thousands) of predicates for real-world programs. Thus, it is essential to keep the number of iterations as small as possible.

The idea of using the elimination algorithm to synthesize conjunctive invariants is not new and can be traced back to the popular Houdini algorithm by Flanagan and Leino [66]. However, a major disadvantage of just using the modified elimination algorithm (or Houdini for that matter) is that it is not *property-driven*: it generates the largest conjunction, independent of negative counterexamples, and, hence, independent of the assertions and specifications in the program. Consequently, a significant amount of time may be spent finding the tightest invariant (involving many predicates), although a simpler and weaker invariant suffices to prove the program correct. This observation has motivated us to develop Sorcar, which is property-driven (i.e., it also considers the assertions in the program) and has a bias towards learning conjunctions with a smaller number of predicates than Houdini. In fact, the set of predicates learned by Sorcar is always a subset of those synthesized by the modified elimination algorithm and Houdini.

The salient feature of Sorcar is that it learns invariants involving what we call relevant predicates, which are predicates that have shown some evidence to affect the assertions in the program. More precisely, we say that a predicate is *relevant* if it evaluates to false on some negative counterexample or on a program configuration appearing on the left-hand side of a Horn counterexample. This definition indicates that at least some relevant predicates must be part of an invariant because not assuming any leads to an assertion violation or the invariant not being inductive. In general, however, naively choosing relevant predicates leads to an exponential number of rounds. Thus, we have designed Sorcar to select relevant predicates carefully, requiring at most $2|\mathcal{P}|$ rounds to converge to an invariant. Note that this is only twice the number of rounds that the modified elimination algorithm and Houdini guarantee.

Algorithm 1 (on Page 15) presents the Sorcar algorithm in pseudo code. It is divided into a passive part (`Sorcar-Passive`) and an iterative part (`Sorcar-Iterative`), the latter being invoked in every round of the Horn-ICE framework. The passive part of

Sorcar maintains a state in the form of a set $R \subseteq \mathcal{P}$, which is empty in the beginning and used to accumulate relevant predicates (Line 1). The exact choice of relevant predicates, however, is delegated to an external function `Relevant-Predicates`. We treat this function as a parameter for the Sorcar algorithm and discuss four possible implementations later in this section.

Given a Horn-ICE sample $\mathcal{S}$ and a set $R \subseteq \mathcal{P}$ of relevant predicates, `Sorcar-Passive` first constructs the largest conjunction $X \subseteq \mathcal{P}$ that is consistent with $\mathcal{S}$ using the modified elimination algorithm (Line 7). Since X is the largest set of predicates consistent with $\mathcal{S}$, it represents the smallest consistent set of program configurations expressible as a conjunction over $\mathcal{P}$. Consequently, any subset of X, in particular $X \cap R$, is necessarily consistent with S_+. However, $X \cap R$ might not be consistent with S_- or $S_\Rightarrow$. To address this problem, `Sorcar-Passive` collects all inconsistent negative counterexamples in a set N and all inconsistent Horn counterexamples in a set H (Lines 9 to 16). Based on these two sets, it then computes a set of relevant predicates, which it adds to R (Line 17). As mentioned above, the exact computation of relevant predicates is delegated to a function `Relevant-Predicates`, which we consider to be a parameter. The result of this function is a new set $R' \subseteq \mathcal{P}$ of predicates that needs to contain at least one new predicate that is not yet present in R. Once such a set has been computed and added to R, the process repeats (R grows monotonically larger) until a consistent conjunction is found. Then, `Sorcar-Passive` returns both the conjunction $X \cap R$ together with the new set R of relevant predicates.

The condition of the loop in Line 8 immediately shows that the set $X \cap R$ is consistent with the Horn-ICE sample $\mathcal{S}$ once `Sorcar-Passive` terminates. The termination argument, however, is less obvious. To argue termination, we first observe that X is consistent with each positive counterexample and, hence, $X \cap R$ remains consistent with all positive counterexamples during the run of `Sorcar-Passive`. Next, we observe that the termination argument is independent of the exact choice of predicates added to R—in fact, the predicates need not even be relevant to prove termination of `Sorcar-Passive`. More precisely, since the function `Relevant-Predicates` must return a set $R' \subseteq \mathcal{P}$ that contains at least one new (relevant) predicate not currently present in R, we know that R grows strictly monotonically. In the worst case, the loop in Lines 8 to 18 repeats $|\mathcal{P}|$ times until $R = \mathcal{P}$; then, $X \cap R = X$, which is guaranteed to be consistent with $\mathcal{S}$ by construction of X (see Line 7). However, depending on the implementation of `Relevant-Predicates`, `Sorcar-Passive` can terminate early with a much smaller consistent set $X \cap R \subsetneq X$. Since the time spent in each iteration of the loop in Lines 8 to 18 is proportional to $|\mathcal{P}| \cdot |\mathcal{S}| + f(|\mathcal{S}|)$, where the function f captures the complexity of `Relevant-Predicates`, the overall runtime of `Sorcar-Passive` is in $\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathcal{S}| + |\mathcal{P}| \cdot f(|\mathcal{S}|))$. In total, we obtain the following result.

Algorithm 1: The Sorcar algorithm [10]

```
1  static R ← ∅;                    // Stores relevant predicates across rounds
2  Procedure Sorcar-Iterative(S):
3  |  (Y, R) ← Sorcar-Passive(S, R);
4  |  return Y;
5  end
6  Procedure Sorcar-Passive(S = (S+, S−, S⇒), R):
7  |  Run the modified elimination algorithm to compute the largest conjunction
   |    X ⊆ P that is consistent with S (abort if no such formula exists);
8  |  while X ∩ R is not consistent with S do
9  |  |  N ← ∅;              // Stores inconsistent negative counterexamples
10 |  |  H ← ∅;                 // Stores inconsistent Horn counterexamples
11 |  |  foreach negative counterexample c ∈ S− not consistent with X ∩ R do
12 |  |  |  N ← N ∪ {c};
13 |  |  end
14 |  |  foreach Horn counterexample (c1 ∧ ··· ∧ cn) → c ∈ S⇒ not consistent with
   |  |    X ∩ R do
15 |  |  |  H ← H ∪ {(c1 ∧ ··· ∧ cn) → c};
16 |  |  end
17 |  |  R ← R ∪ Relevant-Predicates(N, H, X, R);
18 |  end
19 |  return (X ∩ R, R);
20 end
21 Function Relevant-Predicates(N, H, X, R):
22 |  return a set of R′ ⊆ P of relevant predicates such that R′ \ R ≠ ∅;
23 end
```

Theorem 2.1 (cf. Neider et al. [10, Theorem 1])**.** *Given a Horn-ICE sample $\mathcal{S}$ and a set $R \subseteq \mathcal{P}$ of relevant predicates, the passive Sorcar algorithm learns a consistent set of predicates (i.e., a consistent conjunction over $\mathcal{P}$) in time $\mathcal{O}(|\mathcal{P}|^2 \cdot |\mathcal{S}| + |\mathcal{P}| \cdot f(|\mathcal{S}|))$ where f is a function capturing the complexity of the function* `Relevant-Predicates`*.*

In each round of the Horn-ICE framework, the learner invokes `Sorcar-Iterative` with two arguments: a Horn-ICE sample $\mathcal{S}$, which contains all counterexamples that the learner has received thus far, and a set $R \subseteq \mathcal{P}$ of relevant predicates. Internally, `Sorcar-Iterative` calls `Sorcar-Passive`, updates the set R, and returns a new conjunctive formula, which the learner then proposes as a new hypothesis invariant to the teacher. If the computation of X in Line 7 of `Sorcar-Passive` fails and the algorithm aborts, so does `Sorcar-Iterative`.

A careful analysis of Sorcar's updates of X and R shows that in each round of the Horn-ICE framework, either $|X|$ decreases by at least one or $|R|$ increases by at least one. Since $R \subseteq \mathcal{P}$ and $X \subseteq \mathcal{P}$, this can happen at most $2|\mathcal{P}|$ times before Sorcar either finds a conjunctive invariant or aborts. In the latter case, the correctness of the Houdini algorithm implies that no conjunctive invariant over $\mathcal{P}$ exists that proves the given program correct. These results are summarized in the theorem below.

Theorem 2.2 (cf. Neider et al. [10, Theorem 2])**.** *Let P be a program and $\mathcal{P}$ a finite set of predicates over the program configurations C_P. When embedded in the Horn-ICE framework, the iterative Sorcar algorithm learns an inductive invariant (in the form of a conjunction over $\mathcal{P}$) that proves the program correct in at most $2|\mathcal{P}|$ rounds, or it reports that no such invariant exists.*

It is left to show how to select relevant predicates (i.e., how to implement the function `Relevant-Predicates`). To this end, we have proposed four different methods.

Relevant-Predicates-Max The function *Relevant-Predicates-Max* computes the maximal set of relevant predicates with respect to the sets N and H. To this end, it simply accumulates all predicates that evaluate to false on a negative counterexample in N or on a program configuration appearing on the left-hand side of a Horn counterexample in H. Although the resulting set of predicates can be large, this function performed best in our empirical evaluation.

Relevant-Predicates-First The function *Relevant-Predicates-First* tries to select a smaller set of relevant predicates than *Relevant-Predicates-Max* while giving the user some control over which predicates to choose. To this end, *Relevant-Predicates-First* selects for each negative counterexample and each Horn counterexample only one relevant predicate. The exact choice is determined by a total

	Relevant-Predicates-			
	Max	*First*	*Min*	*Greedy*
Complexity	$\mathcal{O}(\lvert\mathcal{P}\rvert \cdot \lvert\mathcal{S}\rvert)$	$\mathcal{O}(\lvert\mathcal{P}\rvert \cdot \lvert\mathcal{S}\rvert)$	$\mathcal{O}(2^{\lvert\mathcal{P}\rvert} + \lvert\mathcal{S}\rvert)$	$\mathcal{O}(\lvert\mathcal{P}\rvert \cdot \lvert\mathcal{S}\rvert^2)$

Table 2.1: Complexity of the functions to select relevant predicates.

ordering over the predicates, which reflects a preference among predicates and which we assume to be given by the user.

Relevant-Predicates-Min The function *Relevant-Predicates-Min* computes a (not necessarily unique) minimum set of relevant predicates with respect to N and H by (i) reducing it (in polynomial-time) to the well-known minimum hitting set problem [94] and (ii) using a constraint solver to find an optimal solution. Note that our reduction implies that finding a minimum set of relevant predicates is computationally expensive since the hitting set problem is NP-complete.

Relevant-Predicates-Greedy The function *Relevant-Predicates-Greedy* replaces the exact computation of a minimum hitting set with a polynomial-time approximation algorithm, which is based on the well-known greedy algorithm for the minimum set cover problem [52]. This function guarantees that the resulting set of predicates is at most logarithmically larger than a minimal one.

The time complexity of all four methods is shown in Table 2.1. Note again that the runtime of the learner in each iteration of the Horn-ICE framework is less of a concern compared to the total number of iterations.

2.2.2 A Horn-ICE Learning Algorithm Based on Decision Trees

Despite the success of tools such as Microsoft's Static Driver Verifier and GPUVerify, the class of conjunctive formulas is often too restrictive to express the invariants and methods contracts required to prove a program correct. To enable the use of Horn-ICE learning also in such situations, we now present a Horn-ICE learning algorithm, named Horn-ICE-DT [2], that learns general formulas represented as decision trees. This algorithm is an extension of one of our previous learners for the ICE learning framework, named ICE-DT [3], which in turn is based on the classical decision tree learning algorithm of Quinlan [129]. Since Horn-ICE-DT is a generalization of ICE-DT, we here focus on the former algorithm and refer the reader to Garg et al. [3] for details about the latter.

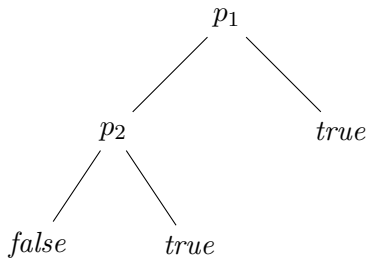


Figure 2.3: A decision tree $\mathcal{T}$ representing the formula $\varphi_{\mathcal{T}} \coloneqq (p_1 \wedge \neg p_2) \vee \neg p_1$.

Horn-ICE-DT aims to construct a decision tree representing a Boolean combination of predicates that are evaluated over program configurations. As for Sorcar, we assume that a set $\mathcal{P}$ of such predicates is given, either provided by the user or automatically generated from the code of the program (e.g., by instantiating templates with various combinations of the variables occurring in the program). Moreover, let us assume for the moment that the set $\mathcal{P}$ is finite. We relax this restriction later in this section.

In general, a *decision tree* is a finite binary tree $\mathcal{T}$ whose nodes have either two children (internal nodes) or no children (leaf nodes). In addition, each internal node is labeled with a predicate from $\mathcal{P}$, while each leaf node is labeled with either *true* or *false*. Figure 2.3 shows an example of such a decision tree.

In the context of Horn-ICE learning, we view a decision tree as a Boolean classifier that evaluates program configurations in the following way: starting at the root, we recursively descend the tree, branching left if the program configuration satisfies the predicate at the current node or right if the configuration does not satisfy it; the final evaluation of the program configuration is then the Boolean label of the leaf node that is eventually reached. Thus, every decision tree $\mathcal{T}$ can be seen as a representation of a Boolean formula $\psi_{\mathcal{T}}$ defined as

$$\psi_{\mathcal{T}} \coloneqq \bigvee_{\pi \in \Pi} \bigwedge_{\lambda \in \pi} \lambda,$$

where Π is the set of all paths from the root of $\mathcal{T}$ to a leaf labeled with *true* and $\lambda \in \pi$ denotes the occurrence of the label λ on a node along the path π (negated if the path branches right). We say that a decision tree $\mathcal{T}$ is consistent with a Horn-ICE sample $\mathcal{S}$ if $\psi_{\mathcal{T}}$ is consistent with $\mathcal{S}$.

While constructing a decision tree, our algorithm needs to deal with *partial trees* where some of the leaf nodes are yet unlabeled. Instead of a label, each such node ν stores a finite set of program configurations, which we denote by $C_\nu \subset C_P$. The sets of configurations at each node are pairwise disjoint and record the program configurations that still need processing in the tree.

Algorithm 2: The Horn-ICE-DT algorithm [2].

```
1  Procedure Horn-ICE-DT(S = (S_+, S_-, S_⇒)):
2      Initialize a partial decision tree T with unlabeled root node ν_r and assign all
         program configurations in S to C_{v_r};
3      Initialize a partial evaluation function η;
4      while there exists an unlabeled node in T do
5          ν ← Select-Node();
6          if ν is pure then Label(ν);
7          if ν is not pure or labeling ν did not succeed then
8              Split(ν);
9              if the split was unsuccessful then abort;
10         end
11     end
12     return T;
13 end

14 Procedure Select-Node():
15     return an unlabeled node to process next;
16 end

17 Procedure Label(ν):
18     Try to label node ν with true or false and return whether this was successful;
19 end

20 Procedure Split(ν):
21     Split node ν or abort if this is not possible;
22 end
```

Algorithm 2 presents our learning algorithm in pseudo code. Given a Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$, the algorithm starts by creating a partial decision tree $\mathcal{T}$ which has a single unlabeled node that stores all program configurations appearing in $\mathcal{S}$. Moreover, it initializes a partial function η that maps each configuration in S_+ to *true*, each configuration in S_- to *false*, and is undefined for each configuration in $S_\Rightarrow$ that is not already contained in S_+ or S_-. This auxiliary function is used to keep track of the label (i.e., evaluation) of individual program configurations and is updated during the run of the algorithm. To simplify the following description, we call a program configuration c *positive* if $\eta(c) = \mathit{true}$, *negative* if $\eta(c) = \mathit{false}$, and *unlabeled* otherwise.

The key idea of our algorithm is to repeatedly select an unlabeled leaf node and either label the node (with *true* or *false*) or grow the tree. More precisely, after the algorithm

has selected an unlabeled leaf node ν using the procedure `Select-Node` (Line 5), it checks if ν is *pure* in the sense that each configuration $c \in C_\nu$ is either (a) positive or unlabeled, or (b) negative or unlabeled. If ν is pure, Algorithm 2 calls the procedure `Label` (Line 6), which tries to label the node with *true* (if Case (a) is satisfied) or *false* (if Case (b) is satisfied). To label ν with $b \in \{true, false\}$, the procedure `Label` executes two steps. First, it tentatively maps all unlabeled configurations in C_ν to b, resulting in a new evaluation function η' that extends the current evaluation η. Second, it calls a modified version of the pebbling algorithm by Dowling and Gallier [57] to check whether η' can be made consistent with $\mathcal{S}$ (which is defined as expected). This involves a satisfiability check of Horn clauses and might force unlabeled configurations in other parts of the tree to be switched positive or negative in order to satisfy the Horn counterexamples. If η' can be made consistent with $\mathcal{S}$, the procedure `Label` updates η with all tentative changes, labels the leaf ν with b, and reports success. If η' cannot be made consistent with $\mathcal{S}$, the procedure `Label` reverts all tentative changes and reports failure.

If the node ν is not pure or it is not possible to label it with *true* or *false*, Algorithm 2 calls the procedure `Split` (Line 8) to grow the tree. This procedure involves four steps. First, we select a suitable predicate $p \in \mathcal{P}$ according to a statistical measure based on the concept of information gain [129]. Second, we replace the node ν by a partial tree with three nodes: a root node ν_r labeled with the predicate p as well as two unlabeled leaf nodes, ν_p (the left child) and $\nu_{\neg p}$ (the right child). Third, we split the set C_ν into two disjoint subsets $C_p, C_{\neg p}$ such that C_p contains all program configurations $c \in C_\nu$ that satisfy the predicate p and $C_{\neg p}$ contains all program configurations that do not. Fourth, we store the set C_p in the node ν_p and $C_{\neg p}$ in the node $\nu_{\neg p}$. For our algorithm to make meaningful progress, we require that a split partitions the set C_ν into two nonempty subsets. If none of the predicates in $\mathcal{P}$ allows for such a split, the procedure aborts.

To complete the discussion of Algorithm 2, let us briefly sketch further details of the procedures `Select-Node`, `Label`, and `Split`. Note that all three procedures are designed to (heuristically) produce small decision trees (in terms of the number of nodes) and, hence, small formulas.

`Select-Node` The task of the `Select-Node` procedure is to find a suitable unlabeled node to process next. In contrast to classical decision tree learning, where the order in which the tree is expanded does not matter, we have to make this choice carefully because changes in one part of the tree can trigger changes in other parts in order to satisfy the Horn counterexamples. To account for this, we have experimented with various strategies to select the next node, including breadth-first search, depth-first search, random selection, and selections based on

maximum or minimum entropy. However, a simple breadth-first search performed best in our empirical evaluation.

`Label` The procedure `Label` needs to check whether a pure node can be labeled with *true* (or *false*) and which unlabeled program configurations have to be turned positive or negative in order to satisfy the Horn counterexamples. To this end, we have developed a novel algorithm for checking the satisfiability of Horn clauses based on the pebbling algorithm of Dowling and Gallier [57]. This new algorithm has two fundamental features. First, it computes the minimal set of (unlabeled) program configurations that need to be turned positive or negative to satisfy the Horn constraints. Note that this property is essential to keep the size of trees small because fixing the labels of too many configurations early on often results in unnecessary splits later in the learning process. Second, our procedure works incremental and reuses information of previous invocations. As a result, we can guarantee an amortized runtime of $\mathcal{O}((|S_+| + |S_-| + |S_\Rightarrow|) \cdot |\mathcal{S}|)$, where the first factor amounts to the total number of counterexamples in the Horn sample.

`Split` Splitting a node entails selecting a suitable predicate $p \in \mathcal{P}$ and performing the actual split. While the latter task is straightforward, selecting a "good" attribute is crucial for the performance of Algorithm 2 as it immediately affects the size of the resulting tree. Since computing minimal decision trees is known to be computationally hard [86], the procedure `Split` uses a heuristic approach based on (weighted) information gain, a statistical measure that is commonly used in decision tree learning [129]. Moreover, we account for Horn counterexamples by adding a penalty if a predicate separates a program configuration in the antecedent of a Horn counterexample from the program configuration in the consequent.

A careful analysis of Algorithm 2 shows that it always constructs a decision tree that is consistent with a given Horn-ICE sample $\mathcal{S}$ if (a) the constraints in $S_\Rightarrow$ are satisfiable and (b) the sample $\mathcal{S}$ is separable in the sense that for each pair $c_1, c_2 \in \mathcal{S}$ of program configurations, there exists a predicate $p \in \mathcal{P}$ such that $c_1 \models p$ if and only if $c_2 \not\models p$. Moreover, total runtime of Algorithm 2 is in $\mathcal{O}((|S_+| + |S_-| + |S_\Rightarrow|) \cdot |\mathcal{S}|)$. This result is summarized in the following theorem.

Theorem 2.3 (cf. Ezudheen et al. [2, Theorem 3.1])**.** *Let $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$ be a Horn-ICE sample and $\mathcal{P}$ a finite set of predicates. If the constraints in $S_\Rightarrow$ are satisfiable and S is separable, then Algorithm 2 learns a decision tree that is consistent with S in time $\mathcal{O}((|S_+| + |S_-| + |S_\Rightarrow|) \cdot |\mathcal{S}|)$.*

Our algorithm can detect whether $\mathcal{S}$ is separable following a strategy introduced in our earlier work on ICE-DT [3]. For each pair of inseparable configurations $c_1, c_2 \in \mathcal{S}$

(which can be pre-computed), we add the Horn constraints $(\{c_1\}, c_2)$ and $(\{c_2\}, c_1)$ to $S_\Rightarrow$, resulting in a new set $S'_\Rightarrow$. Running Algorithm 2 with the augmented Horn-ICE sample $\mathcal{S}' = (S_+, S_-, S'_\Rightarrow)$ is then guaranteed to construct a decision tree consistent with $\mathcal{S}$ if and only if there exists such a tree. Hence, if Algorithm 2 aborts, it follows that no consistent decision tree over $\mathcal{P}$ exists.

The modification above allows us to apply Algorithm 2 even in situations where the set $\mathcal{P}$ of predicates is countably infinite (e.g., comparisons of program variables with rational values). Starting with a finite set Q containing the first, say, $\ell \in \mathbb{N} \setminus \{0\}$ elements of $\mathcal{P}$, we run Algorithm 2 and, if it aborts, add the next $\ell' \in \mathbb{N} \setminus \{0\}$ predicates to Q. Once the algorithm succeeds, we have fount a consistent tree and return it. It is not hard to verify that this approach is indeed guaranteed to converge to an invariant if one is expressible as a decision tree over $\mathcal{P}$. This result is summarized next.

Theorem 2.4 (cf. Garg et al. [3, Theorem 2])**.** *Let P be a program and $\mathcal{P}$ an enumerable set of predicates over the program configurations S_P. When embedded in the Horn-ICE framework, Horn-ICE-DT learns an inductive invariant that proves the program correct if one can be expressed as a decision tree over $\mathcal{P}$.*

In the presence of numeric variables in the program (and hence in the program configurations), we can use a modification similar to Quinlan's C4.5 and C5.0 suite of algorithms [128] to automatically generate predicates based on the numeric values that appear in the Horn-ICE sample. Note that such a data-driven approach is substantially more efficient than a naive enumeration of predicates that does not consider the data in the sample.

2.3 Invariant Synthesis for Incomplete Verification Engines

Deductive verification is a highly effective approach if the generated verification conditions fall into a decidable logic. For many real-world programs, however, this is not the case. For instance, when a program accesses the heap, involves non-linear arithmetic, or its annotations require quantification, the validity problem for the resulting verification conditions is typically undecidable, thus, preventing the use of off-the-shelf constraint solvers.

A standard technique to address the problem of undecidable verification conditions is to build sound-but-incomplete decision procedures, thus skirting the undecidability barrier. Several such techniques exist: for example, for reasoning with quantified formulas, tactics such as bounded or model-based *quantifier instantiation* [72, 113] are effective in practice, and they are known to be complete in certain settings [108]; in

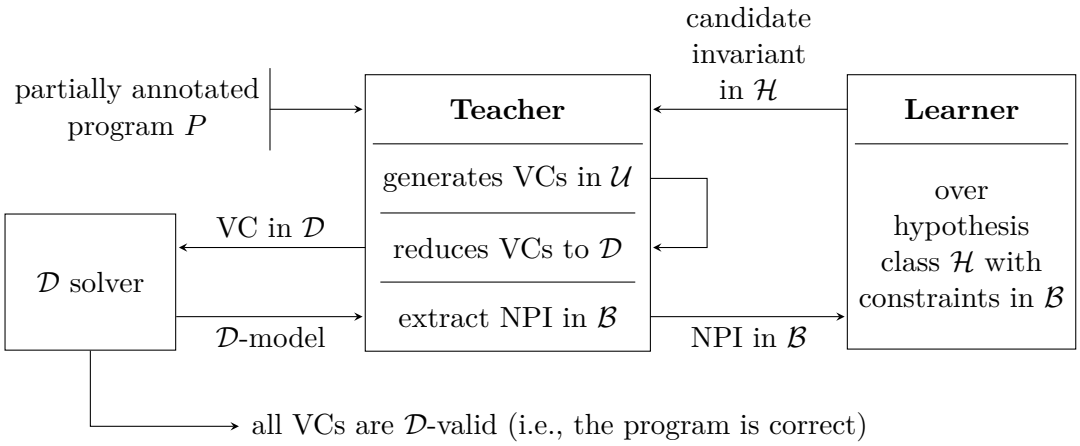


Figure 2.4: The NPI framework for synthesizing invariants when checking validity of verification conditions is undecidable [6, 8].

the realm of heap verification, on the other hand, a method called *natural proofs* aims to provide automated and sound-but-incomplete methods for checking the validity of verification conditions with specifications in separation logic [108, 124, 127].

Although classical invariant synthesis techniques, such as Houdini [66], are sometimes used with sound-but-incomplete verification engines, there is no principled argument as to why this should work in general. In fact, we are not aware of any systematic technique for synthesizing invariants or method contracts when the underlying verification problem cannot be expressed in a decidable logic. When verification is undecidable and the engine resorts to sound-but-incomplete heuristics to check the validity of verification conditions, it is unclear how to extend deductive techniques such as interpolation [110] or PDR/IC3 [41] to this setting. Data-driven learning of invariants is also hard to extend because the verification engine can—in general—not produce the concrete configurations (i.e., counterexamples) that the learner needs.

Motivated by the fact that sound-but-incomplete verification engines have become increasingly powerful in the last decade, we have developed an extension of the Horn-ICE framework that provides a principled method to synthesize invariants (and contracts) when the validity problem for verification conditions is undecidable [6, 8]. As shown in Figure 2.4, our extension resembles Horn-ICE learning in that it consists of a feedback loop with two entities: a teacher and a learner. In this more advanced setting, however, the teacher uses a sound-but-incomplete verification engine, and counterexamples are logical constraints, named *non-provability information (NPI)*, rather than concrete program configurations. We call this extension the *NPI framework*.

In the following three subsections, we present the NPI framework in more detail: Section 2.3.1 gives a general overview of its main features, while Sections 2.3.2 and 2.3.3 discuss the teacher and learner, respectively. To ease our presentation in the remainder, we use a Hoare-style notation [82] of the form $\{\alpha\}s\{\beta\}$ where α, β are formulas evaluated over program configurations and s is a program snippet. Such a triple, called *Hoare triple*, expresses that if the pre-condition α is met, executing the program snippet s establishes the post-condition β. In the context of deductive verification, a verification engine takes such Hoare triples as input and produces corresponding verification conditions, which we denote by $VC(\{\alpha\}s\{\beta\})$.

2.3.1 The NPI Framework

As shown in Figure 2.4, the NPI framework involves four distinct logics: $\mathcal{U}$, $\mathcal{D}$, $\mathcal{B}$, and $\mathcal{H}$. For simplicity of exposition, we assume a uniform signature for all of these logics in terms of constant symbols, relation symbols, functions, types, and so on.

We begin our description of the NPI framework by fixing an undecidable logic $\mathcal{U}$, which is ideally needed for validating the verification conditions that arise from a program P. Since checking the validity of formulas in $\mathcal{U}$ is undecidable, we assume that the verification engine approximates verification conditions in a decidable logic $\mathcal{D}$ (e.g., using bounded quantifier instantiation, bounded unfolding of recursive functions, or natural proofs). This approximation needs to be sound in the sense that if the resulting formulas are valid in $\mathcal{D}$, then the original verification conditions are valid in $\mathcal{U}$ as well. In addition, if a formula is not valid in $\mathcal{D}$, we require that the constraint solver for the logic $\mathcal{D}$ returns a model (i.e., a satisfying assignment) for the negation of the formula. Note that this model may not be a model for the negation of the verification condition in $\mathcal{U}$.

Next, we fix a hypothesis class $\mathcal{H}$ for invariants consisting of positive Boolean combinations over a fixed set $\mathcal{P}$ of predicates. Note that only considering positive formulas over $\mathcal{P}$ is not a restriction as one can always add negated predicates to $\mathcal{P}$, thus effectively synthesizing any Boolean combination of predicates. The restriction to positive Boolean formulas is, in fact, desirable because it allows restricting invariants to not negate certain predicates, which is useful when predicates have intuitionistic definitions (as several recursive definitions of heap properties do). We encourage the reader to think of predicates such as "the program variable x points to a sorted list", "the variable y point to balanced tree", and so on.

The NPI framework proceeds in rounds, where in each round the synthesizer proposes invariants in $\mathcal{H}$. The teacher (i.e., the sound-but-incomplete verification engine) generates verification conditions in accordance to these invariants in the underlying

logic $\mathcal{U}$. It then proceeds to translate them into the decidable logic $\mathcal{D}$ and invokes a constraint solver that decides validity in $\mathcal{D}$. If the verification conditions are found to be $\mathcal{D}$-valid, then the program is correct because the verification engine soundly reduced the verification conditions.

However, if the formula is found not to be $\mathcal{D}$-valid, the constraint solver returns a $\mathcal{D}$-model for the negation of the formula. The teacher then extracts from this model non-provability information, expressed as Boolean formulas in a Boolean theory $\mathcal{B}$, that capture more general reasons why the verification failed (we introduce non-provability information shortly). This non-provability information is communicated to the learner, which then proceeds to synthesize a new invariant that satisfies the non-provability information provided in all previous rounds. We assume that the underlying decidable logic $\mathcal{D}$ is stronger than propositional logic $\mathcal{B}$, meaning that every valid statement in $\mathcal{B}$ is valid in $\mathcal{D}$ as well.

In order to extract meaningful non-provability information, we make the natural assumption that a sound-but-incomplete verification engine can do at least minimal Boolean reasoning. More precisely, we assume that if a Hoare triple $\{\alpha\}s\{\beta\}$ is not provable, then Boolean weakenings of the pre-condition α and Boolean strengthening of the post-condition β must also be unprovable. We call a verification engine with this property *normal*.

2.3.2 A Teacher for Non-Provability Information

Let us first introduce the following notations to formally define non-provability information and its extraction from a failed verification attempt. For any $\mathcal{U}$-formula φ, let $\mathit{approx}(\varphi)$ denote the $\mathcal{D}$-formula that the verification engine generates such that the $\mathcal{D}$-validity of $\mathit{approx}(\varphi)$ implies the $\mathcal{U}$-validity of φ. Moreover, we say that a formula α is *weaker* (*stronger*) than a formula β in a logic $\mathcal{L}$ if $\vdash_{\mathcal{L}} \beta \to \alpha$ ($\vdash_{\mathcal{L}} \alpha \to \beta$), where $\vdash_{\mathcal{L}} \varphi$ means that φ is valid in $\mathcal{L}$. Finally, for a set $Q \subseteq \mathcal{P}$ of predicates, let $\bigvee Q = \bigvee_{p \in Q} p$ and $\bigwedge Q = \bigwedge_{p \in Q} p$ denote the disjunction and conjunction of all predicates in Q, respectively.

To ease the following exposition, let us assume that the given program P has a single location at which we need to synthesize an inductive invariant to prove P correct. Further, suppose the learner conjectures a formula γ as an inductive invariant, and the verification engine fails to prove the verification condition corresponding to a Hoare triple $\{\alpha\}s\{\beta\}$, where either α, β, or both might involve the synthesized formula. This means that the negation of $\mathit{approx}(\mathit{VC}(\{\alpha\}s\{\beta\}))$ is $\mathcal{D}$-satisfiable and the verification engine needs to extract non-provability information from a model of it. To this end, we assume that the program snippet s has been augmented with a set of Boolean

ghost variables $g_1, \ldots, g_n$ that track the predicates $p_1, \ldots, p_n$ appearing in the invariant (i.e., these ghost variables are assigned the values of the predicates). The valuation $\vec{v} = (v_1, \ldots, v_n)$ of the ghost variables in the model before the execution of s and the valuation $\vec{v}' = (v'_1, \ldots, v'_n)$ after its execution allow us now to derive the desired non-provability information.

The type of non-provability information the teacher extracts depends on where the synthesized formula γ appears in a Hoare triple $\{\alpha\}s\{\beta\}$. More specifically, γ might appear in α, in β, or in both. We now handle all three cases individually.

- Assume the verification of a Hoare triple of the form $\{\alpha\}s\{\gamma\}$ fails; in other words, the verification engine cannot prove a verification condition where the pre-condition α is a user-supplied annotation, and the post-condition is the synthesized formula γ. Then, $\mathit{approx}(\mathit{VC}(\{\alpha\}s\{\gamma\}))$ is not $\mathcal{D}$-valid, and the decision procedure for $\mathcal{D}$ generates a model for its negation. Since γ is a positive Boolean combination, the reason why $\vec{v}'$ does not satisfy γ is due to the variables mapped to *false* in $\vec{v}'$ since any valuation extending $\vec{v}'$ can also not satisfy γ. Intuitively, this means that the $\mathcal{D}$-solver is not able to prove the predicates in $P_f = \{p_i \in \{p_1, \ldots, p_n\} \mid v'_i = \mathit{false}\}$. In other words, $\{\alpha\}s\{\bigvee P_f\}$ is unprovable (a witness to this fact is the model of the negation of $\mathit{approx}(\mathit{VC}(\{\alpha\}s\{\gamma\}))$ from which the evaluation $\vec{v}'$ is derived). Moreover, note that any invariant γ' stronger than $\bigvee P_f$ results in an unprovable verification condition because the verification engine is assumed to be normal. Consequently, the disjunction $\chi = \bigvee P_f$ can be seen as a *weakening constraint*. The teacher now returns χ to the learner, asking it to never conjecture an invariant γ' in future rounds that is stronger in the logic $\mathcal{B}$ than χ (i.e., $\not\vdash_{\mathcal{B}} \gamma' \rightarrow \chi$ must always hold).
- Assume now that the verification of a Hoare triple of the form $\{\gamma\}s\{\beta\}$ fails; in other words, the verification engine cannot prove a verification condition where the post-condition β is a user-supplied annotation and the pre-condition is the synthesized formula γ. Using similar arguments as above, the conjunction $\eta = \bigwedge P_t$ with $P_t = \{p_i \in \{p_1, \ldots, p_n\} \mid v_i = \mathit{true}\}$ can be seen as a *strengthening constraint* (note that we here consider the values of $\vec{v}$). The teacher now returns η to the learner, asking it to never conjecture an invariant γ' in future rounds that is weaker in the logic $\mathcal{B}$ than η (i.e., $\not\vdash_{\mathcal{B}} \eta \rightarrow \gamma'$ must always hold).
- Finally, consider the case when the verification of a Hoare triple is of the form $\{\gamma\}s\{\gamma\}$ fails; in other words, the verification engine cannot prove a verification condition where both the pre-condition and the post-condition are the synthesized annotation γ. In this case, the verification engine can offer advice on strengthening or weakening γ. Analogous to the two cases above, the teacher returns a pair of formulas (η, χ), called *inductivity constraint*, based on the variables mapped to *true* in $\vec{v}$ and to *false* in $\vec{v}'$, respectively. The meaning of such a constraint is that

the invariant synthesizer must conjecture an invariant γ' in future rounds such that either $\not\models_{\mathcal{B}} \eta \to \gamma'$ or $\not\models_{\mathcal{B}} \gamma' \to \chi$ holds.

We subsume all these constraints under the term *non-provability information (NPI)*. Note that weakening, strengthening, and inductivity constraints are constraints in the logic $\mathcal{B}$. Thus, the learner, which we discuss next, only needs to reason in this logic but not in the more complex logics $\mathcal{U}$ or $\mathcal{D}$.

2.3.3 Learning from Non-Provability Information

We assume that the learner stores the non-provability information in a so-called *NPI sample* $\mathfrak{S} = (W, S, I)$ consisting of a finite set W of weakening constraints, a finite set S of strengthening constraints, and a finite set I of inductivity constraints. We say that a formula γ is *consistent* with an NPI sample $\mathfrak{S}$ if $\not\models_{\mathcal{B}} \gamma \to \chi$ for each $\chi \in W$, $\not\models_{\mathcal{B}} \eta \to \gamma$ for each $\eta \in S$, and $\not\models_{\mathcal{B}} \eta \to \gamma$ or $\not\models_{\mathcal{B}} \gamma \to \chi$ for each $(\eta, \chi) \in I$. The learner's task is then to synthesize a formula γ from the class $\mathcal{H}$ of positive Boolean formulas that is consistent with the current NPI sample $\mathfrak{S}$.

Our learner solves this logical synthesis problem by reducing it to a data-driven Horn-ICE learning problem. The main idea is to (a) treat each predicate $p \in \mathcal{P}$ as a Boolean variable for the purpose of Horn-ICE learning and (b) to translate an NPI sample $\mathfrak{S}$ into an *equi-consistent* Horn-ICE sample $\mathcal{S}_{\mathfrak{S}}$, meaning that a positive Boolean formula is consistent with $\mathfrak{S}$ if and only if it is consistent with $\mathcal{S}_{\mathfrak{S}}$. Then, learning consistent $\mathcal{H}$-formulas in the NPI framework amounts to learning consistent $\mathcal{H}$-formulas in the Horn-ICE framework. This approach allows us to use learning algorithms such as the modified elimination algorithm (Houdini) and Sorcar to synthesize invariants in the presence of sound-but-incomplete verification engines.

Our translation, which relies on two functions, c and d. The function c translates a conjunction $\bigwedge Q$, where $Q \subseteq \mathcal{P}$ is a subset of predicates, into the valuation $c(\bigwedge Q) = (v_1, \ldots, v_n)$ with $v_i = \mathit{true}$ if and only if $p_i \in Q$. The function d, on the other hand, translates a disjunction $\bigvee Q$ into the valuation $d(\bigvee Q) = (v_1, \ldots, v_n)$ with $v_i = \mathit{false}$ if and only if $p_i \in Q$.

Given an NPI sample $\mathfrak{S}$, we obtain the Horn-ICE sample $\mathcal{S}_{\mathfrak{S}}$ by substituting every conjunction $\bigwedge Q$ in $\mathfrak{S}$ with $c(\bigwedge Q)$ and every disjunction $\bigvee Q$ with $d(\bigvee Q)$. Exploiting the fact that the learner generates positive Boolean formulas only, this translation indeed results in an equi-consistent Horn-ICE sample (see Neider et al. [8, Theorem 1]). Moreover, a careful analysis of the properties of non-provability information shows that at least one incorrect or unprovable formula is excluded from $\mathcal{H}$ in every iteration of the NPI framework. If we assume $\mathcal{P}$ to be finite, then there exist only finitely many

semantically distinct formulas in $\mathcal{H}$, and the NPI framework converges in finite time. In total, we obtain the following result.

Theorem 2.5 (cf. Neider et al. [8, Theorem 2])**.** *Assume a normal verification engine for a program P to be given. Moreover, let $\mathcal{P}$ be a finite set of predicates over the set C_P of program configurations and $\mathcal{H}$ the class consisting of positive Boolean combinations over predicates in $\mathcal{P}$. If there exists an invariant in $\mathcal{H}$ that the verification engine can use to prove P correct, then the NPI framework is guaranteed to converge to such an invariant in finite time.*

If the learner can produce consistent formulas that are minimal with respect to a total order on $\mathcal{H}$, we can relax the requirement of $\mathcal{P}$ being finite. In this case, Theorem 2.5 remains true even if $\mathcal{P}$ contains infinitely many predicates.

2.4 Notes on Related Work

Invariant synthesis is the central problem in automated program verification, and various techniques have been proposed over the years. Examples include abstract interpretation [54], interpolation [91, 110], IC3 and PDR [41, 93], predicate abstraction [27], abductive inference [56], as well as synthesis algorithms that rely on constraint solving [53, 78, 79].

Subsequent to the work of Grebenshchikov et al. [75], Horn clauses have crystallized as a "universal language" to express verification conditions of programs [35, 38]. For instance, SeaHorn [80] is a verification framework that translates verification conditions into constraint Horn clauses that can be solved using several backend solvers, such as Z3 [114].

Complementary to the purely deductive methods mentioned above are data-driven invariant synthesis techniques. In fact, this type of invariant synthesis has seen increasing interest lately [44, 63, 70, 71, 119, 121, 123, 137–140, 154, 155]. When the program under consideration manipulates complex data structures, such as arrays or pointers, or when one needs to reason about complicated memory models and their semantics, the invariant for the correctness of the program might still be simple. In such a scenario, a black-box, data-driven guess-and-check approach, guided by a finite set of program configurations, has been shown to be advantageous. This observation has motivated the development of the ICE learning framework [69] and, subsequently, the Horn-ICE learning framework [2]. To the best of our knowledge, ICE learning was the first robust framework for learning inductive invariants.

The development of Horn-ICE learning algorithms has focused on two important classes of invariants: conjunctive invariants and invariants that can be expressed as decision trees. The latter type can easily be translated into formulas in disjunctive normal form and, hence, provides a standardized way to represent arbitrary formulas over a particular set of predicates.

Learning conjunctive formulas has a long history. An early example is the so-called elimination algorithm [95], which operates in the Probably Approximately Correct Learning (PAC) model. Daikon [62] was the first technique to apply the elimination algorithm in a software setting, learning likely invariants from dynamic traces. Later, the popular Houdini algorithm [66] adapted the elimination algorithm to compute inductive invariants in a fully automated manner. As Garg et al. [69] and subsequently we [2] have argued, Houdini can be seen as a learning algorithm for conjunctive formulas in both the ICE learning and the Horn-ICE learning framework. Our Sorcar algorithm [10] also builds on top of the elimination algorithm but extends it to be property-driven (i.e., to take the program's assertion into account).

Apart from ICE-DT and Horn-ICE-DT, we are aware of multiple other algorithms that learn decision trees in the context of software verification. Among the most prominent ones is the algorithm by Champion et al. [48], which learns decision trees from Horn samples in order to synthesize refinement types for higher-order functional programs. However, this algorithm is different from ICE-DT and Horn-ICE-DT in that it learns one annotation at a time, while our approaches learn all annotations simultaneously. Moreover, Champion et al.'s algorithm does neither guarantee always to construct a decision tree if one exists nor that the learner eventually converges to a solution when the hypothesis class is infinite. Both are true for ICE-DT and Horn-ICE-DT (under mild assumptions).

Another example is the algorithm by Zhu, Magill, and Jagannathan [153], which uses decision tree learning as a back-end for solving constrained Horn clauses. This algorithm has two main differences to our approach. First, Zhu, Magill, and Jagannathan's algorithm generates predicates automatically, whereas we operate within a fixed set of predicates (octagonal constraints over the program variables). Note, however, that this is not a restriction but a design choice that balances effectiveness with performance: in fact, our algorithm can easily handle any other user-specified set of decidable predicates. Second, Zhu, Magill, and Jagannathan's algorithm operates in the classical machine learning setup with only positively and negatively labeled data: if a conjecture is found to be non-inductive, additional positive and negative examples are generated by unwinding the constrained Horn clauses a finite number of times (which increases during the learning process). However, this approach of generating counterexamples does not guarantee convergence to a solution (if one exists)—a stark contrast to ICE-DT and Horn-ICE-DT, which both provide this guarantee.

One interesting question is whether (Horn-)ICE algorithms (particularly Houdini and Sorcar) are qualitatively related to purely deductive algorithms such as IC3. For programs with Boolean domains, Vizel et al. [149] have studied this question and found that (Horn-)ICE learning and IC3 are different in various regards:

- IC3 finds invariants by bounded symbolic exploration, forward from initial configurations and backward from bad configurations (hence, inherently unfolding loops), while (Horn-)ICE algorithms do not;
- (Horn-)ICE algorithms instead use implications and Horn counterexamples, respectively, which can relate configurations arbitrarily far away from initial or bad configurations, and there seems to be no analog to this in IC3;
- it is unclear how to restrict IC3 to synthesize invariants in a particular hypothesis class, such as conjunctions over a particular set of predicates;
- IC3 works in close integration with a SAT solver, whereas (Horn-)ICE algorithms are essentially independent of the verification engine, communicating with SAT or SMT solvers only indirectly; and
- we are not aware of any guarantees that IC3 can give in terms of the number of rounds or conjectures, whereas the (Horn-)ICE algorithms Houdini and Sorcar give guarantees that are linear in the number of predicates.

Despite these differences, however, Vizel et al. have proposed a new framework that generalizes both (Horn-)ICE and IC3.

In the context of verifying programs that work over dynamically allocated data structures, most verification engines are necessarily sound but incomplete (as the underlying decision problems are undecidable), and invariant synthesis is hard. Nonetheless, shape analysis has been used successfully to synthesize invariants in various settings [45, 104, 134]. However, most of these methods are tailored to memory safety and other shallow properties, but they do not handle rich properties expressing full functional correctness of data structures as we do in the NPI framework. Interpolation has also been suggested to synthesize invariants involving a combination of data and shape properties [17]. It is, however, not clear how this technique can be applied to a more complicated heap structure, such as an AVL tree, where shape and data properties are not cleanly separated but are intricately connected. Recent work also includes synthesizing heap invariants in the logic proposed by Itzhaky et al. [87] by extending IC3 [88, 93]. However, we are not aware of any approach that can fully automatically verify the extensive benchmark suit that our NPI framework can.

INTELLIGENT REACTIVE AND FUNCTIONAL SYNTHESIS

3

Automated synthesis techniques offer an effective and promising way to solve a fundamental practical problem: constructing correct and verified hardware and software from formal specifications. Rather than designing and implementing a complex system by hand, synthesis techniques enable the construction of its constituent components in an automated fashion, thus, freeing engineers from this complex and error-prone task. In addition to being fully automatic, synthesis techniques produce hardware and software that is correct-by-construction, meaning that the given specification is guaranteed to be satisfied. Many synthesis techniques also detect if the synthesis task is unrealizable (i.e., the specification cannot be implemented in hardware or software), in which case the engineer can correct the specification and repeat the synthesis process.

In this chapter, we show how inductive methods from the area of machine learning can be used to complement and improve classical, purely deductive synthesis approaches. In particular, we consider the following two synthesis settings:

- In Section 3.1, we consider reactive synthesis. The task in this setting is to automatically translate a user-provided (temporal) specification into a reactive system, which is a circuit or a piece of software that continuously interacts with its environment (e.g., by serving user requests). The key challenge in this context arises from the fact that a reactive system is not operating in isolation, but it has to satisfy its specification regardless of how the (potentially antagonistic) environment might act. Among the most prominent examples of reactive synthesis are the synthesis of (parts of) the high-performance bus controller for ARM's on-chip communication standard [39] and the abstraction-based controller design approach, which reduces control problems for continuous systems via finite-state abstractions to reactive synthesis tasks [130].

- In Section 3.2, we consider functional synthesis. The task here is to synthesize the implementation of a function (e.g., a mathematical expression or a piece of loop-free code) from user-given specifications of the desired input-output behavior. As in reactive synthesis, the synthesized object has to satisfy the given specification for every possible input, but the specification does not involve temporal constraints and, hence, does typically not require memory for its implementation. Examples of this type of synthesis include Sketch [143], a system to synthesize software from incomplete code, as well as Flash Fill [76], a feature of Microsoft Excel that synthesizes macros for string manipulation from examples.

In Section 3.3, we present a framework that unifies (almost) all synthesis techniques developed in this work (and many more found in the literature). Our ultimate goal is to provide a common vocabulary and set of definitions that can be used to understand and combine learning-based synthesis techniques across a wide range of domains. We conclude this chapter in Section 3.4 with a brief discussion of related work.

3.1 Reactive Synthesis

The concept of reactive synthesis can be traced back to work by Alonzo Church in 1957, which is today known as *Church's synthesis problem* [50, 51]. In modern parlance, Church's problem asks the following: given a specification on the sequence of inputs and outputs of a reactive system expressed in a suitable logic, construct one that satisfies this specification (regardless of the environment's behavior) or determine that none exists. This question has attracted a significant amount of attention over the last six decades, and we refer the reader to the summary articles by Thomas [144, 145] for a gentle introduction.

The goal of this section is to show how modern machine learning techniques can be used to solve Church's synthesis problem. In particular, we describe how the synthesis problem for safety specifications, a specific type of specifications that requires the system always to operate safely, can be reduced to the Horn-ICE framework of Section 2.1. This reduction allows us to use any Horn-ICE learning algorithm for reactive synthesis.

Throughout this section, we follow the game-theoretic approach to reactive synthesis as popularized by McNaughton [111]. More precisely, we view the problem as an infinite-duration, two-player game on a directed graph that is played by two antagonistic players: Player 0 (who embodies the system and seeks to satisfy the specification) and Player 1 (who embodies the environment and wants to violate it). The actual synthesis proceeds in three steps. First, the specification and a model of the environment are converted into an infinite game. Second, one computes a winning strategy for the system, which prescribes how the system needs to play in order to win against every move of the environment. Third, the winning strategy is translated into hard- or software, resulting in an implementation of a reactive system that satisfies the given specification.

As alluded to above, we focus on safety games, a class of infinite games that arises from safety specifications. Such specifications are in fact among the most important in practice (e.g., see Dwyer, Avrunin, and Corbett [58] for a survey of common specification patterns) and capture many other interesting properties, including bounded-horizon reachability. In contrast to McNaughton's original setting, however, we consider safety games not only over finite graphs but over graphs with potentially infinitely many (even uncountably many) vertices. Such games occur naturally, for instance, when

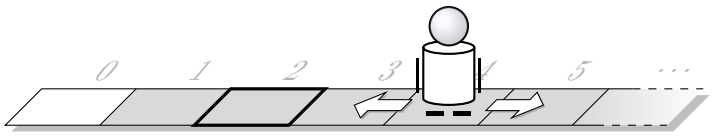
the interaction between the controlled system and its environment is too complex to be modeled by finite graphs (e.g., in motion planning over unbounded environments) or when the system has access to dynamic data structures, such as lists, stacks, or queues.

Formally, a *safety game* is a five-tuple $\mathcal{G} = (V_0, V_1, E, F, I)$ consisting of two disjoint sets V_0, V_1 of vertices controlled by Player 0 and Player 1, respectively (we denote their union by $V = V_0 \cup V_1$ and assume $V \neq \emptyset$), a directed edge relation $E \subseteq V \times V$, a nonempty set $F \subseteq V$ of *safe vertices*, and a nonempty set $I \subseteq F$ of *initial vertices*. The directed graph (V, E) is typically called *game graph*. In contrast to the classical setting, we do not restrict V to be finite but allow even uncountable sets. However, we do make the following two restrictions to the edge relation (where $E(X)$ denote the image of a set $X \subseteq V$ under the edge relation E): we assume that (i) every vertex has at least one outgoing edge (i.e., $E(\{v\}) \neq \emptyset$ for each $v \in V$), and (ii) $E(\{v\})$ is finite for every $v \in V$, though not necessarily bounded. Note that the first restriction is standard and merely avoids situations in which the game gets stuck. On the other hand, the second restriction is required to make the setting amenable to machine learning.

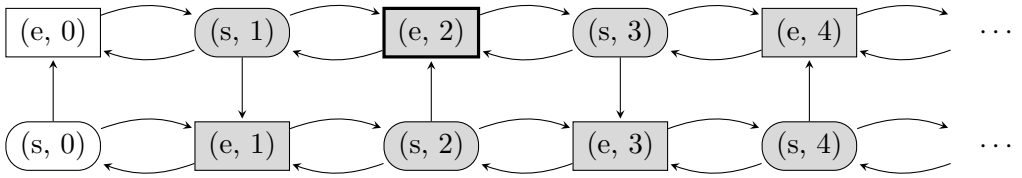
A safety game is played in rounds: initially, a token is placed on one of the initial vertices $v_0 \in I$; in each round, the player controlling the current vertex then moves the token along one of the outgoing edges to the next vertex. This process of moving the token is repeated ad infinitum and results in an infinite sequence $\pi = v_0 v_1 \ldots \in V^\omega$ with $v_0 \in I$ and $(v_i, v_{i+1}) \in E$ for every $i \in \mathbb{N}$, which is called a *play*. The winner of a play is determined by the winning condition F in the sense that a play $\pi = v_0 v_1 \ldots$ is *winning for Player 0* if $v_i \in F$ for every $i \in \mathbb{N}$—otherwise it is *winning for Player 1*.

As mentioned above, synthesizing a reactive system amounts to computing a so-called winning strategy for Player 0, which prescribes how Player 0 needs to move to win a play. For the sake of brevity, we skip a formal definition of winning strategies and introduce a proxy object instead, which we call a *winning set*. On an intuitive level, a winning set is a subset of the safe vertices that contains all initial vertices and is a trap for Player 1 (i.e., Player 0 can force any play to stay inside this set regardless of how Player 1 plays). Formally, we say that $W \subseteq V$ is a winning set if it satisfies the following four properties:

1. $I \subseteq W$;
2. $W \subseteq F$;
3. $E(\{v\}) \cap W \neq \emptyset$ for all $v \in W \cap V_0$ (*existential closedness*); and
4. $E(\{v\}) \subseteq W$ for all $v \in W \cap V_1$ (*universal closedness*).



(a) A robot moving on an infinite conveyor belt. The safe area is shaded gray, while the unsafe area is shown in white. The robot starts inside the area decorated by a bold border.



(b) A safety game with vertex set $\{e, s\} \times \mathbb{N}$ modeling a discretized version of the motion planning problem of Figure 3.1a (e represents the environment, while s represents the system). Vertices of Player 0 (the system) are drawn as ellipses, while vertices of Player 1 (the environment) are drawn as rectangles. Initial vertices are decorated with a bold border, and safe vertices are shaded gray.

Figure 3.1: An example of robotic motion planning (top) together with a possible encoding as a safety game (bottom).

It is not hard to verify that a winning set W immediately provides a strategy for Player 0 to win any play: starting in $I \subseteq W$, Player 0 simply moves to a (fixed) successor vertex inside W every time the play reaches one of his vertices (note that this is possible since W is existentially closed). As W is also universally closed, a straightforward induction over the length of plays proves that every play that starts inside I and is played according to this strategy stays inside W, no matter how Player 1 plays. Thus, Player 0 wins since $W \subseteq F$.

Let us illustrate safety games and the notion of winning sets with the example in Figure 3.1, which is motivated by robotic motion planning. In this example, a robot moves on an infinite conveyor belt that is "bounded on the left" (see Figure 3.1a). The conveyor belt is partitioned into a safe and an unsafe area, shaded gray and white, respectively. The robot starts inside the area marked with a bold border and is controlled by the system (i.e., Player 0), which can move the robot one unit to the left (if it is not already at the left edge), one unit to the right, or not move it at all. The conveyor belt, on the other hand, is controlled by the environment (i.e., Player 1), which can move the belt to the left or the right, effectively moving the robot by one unit in the corresponding direction (if it is not already at the left edge). Starting with Player 1, both players make their moves in alternation, and the system's goal is to keep the robot inside the safe area.

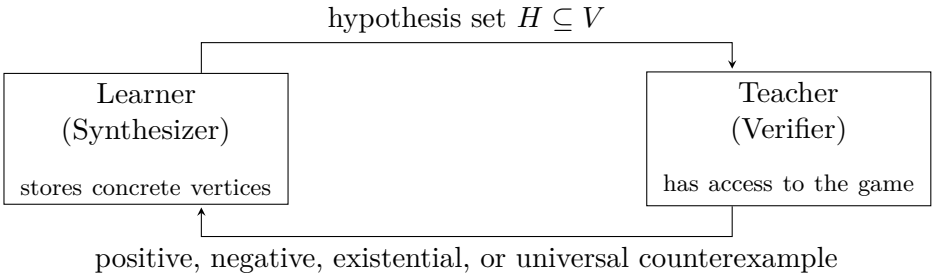


Figure 3.2: Our generic framework for learning winning sets [9, 13].

The setting in Figure 3.1a allows for various interpretations, depending on whether we want to discretize the positions the robot can assume (we discuss various symbolic representations of safety games later in this section). For instance, Figure 3.1b shows the game graph resulting from abstracting the conveyor belt into cells of width one, where we do not distinguish between positions in the interval $[i, i+1)$ for $i \in \mathbb{N}$. For this particular game, a winning set is $W = \{(s,i) \in V \mid i \geq 1\} \cup \{(e,i) \in V \mid i \geq 2\}$, and an obvious winning strategy for Player 0 is to always move the robot one unit to the right.

If the game graph underlying a safety game is finite, a winning set (indeed the largest one) can be computed in linear time using a simple fixed-point computation [74]. However, for games over infinite game graphs, this is no longer an option as a fixed-point computation might not converge in finite time. To overcome this severe practical limitation, we propose a novel framework that learns winning sets rather than computes them explicitly [9, 13].

Figure 3.2 presents a schematic view of our framework for learning winning sets. Similar to the methods presented in Chapter 2, it follows the principle of counterexample-guided inductive synthesis [142] and consists of a feedback loop with two entities: a *teacher*, who knows the safety game, and a *learner*, whose objective is to learn a winning set, but who is agnostic to the game.

In every loop iteration, the learner conjectures a hypothesis $H \subseteq V$ based on the information about the game it has accumulated so far. Then, the teacher checks whether this set H is, in fact, a winning set. Although the teacher does not know a winning set (the task is to learn one after all), it can verify whether the hypothesis is one by checking the four conditions of the definition of winning sets. If the hypothesis satisfies these conditions, then H is a winning set, and the learning stops. If this is not the case, the teacher replies with one out of four different types of counterexamples, which mirror the four conditions in the definition of winning sets:

1. If $I \not\subseteq H$, then the teacher returns a *positive counterexample* $v \in I \setminus H$.
2. If $H \not\subseteq F$, then the teacher returns a *negative counterexample* $v \in H \setminus F$.
3. If there exists a vertex $v \in H \cap V_0$ with $E(\{v\}) \cap H = \emptyset$, then the teacher returns an *existential counterexample* $v \to (v_1 \vee \cdots \vee v_n)$ with $\{v_1, \ldots, v_n\} = E(\{v\})$.
4. If there exists a vertex $v \in H \cap V_1$ with $E(\{v\}) \not\subseteq H$, then the teacher returns a *universal counterexample* $v \to (v_1 \wedge \cdots \wedge v_n)$ with $\{v_1, \ldots, v_n\} = E(\{v\})$.

The meaning of a positive counterexample is that any future hypothesis needs to include this vertex (as it is initial), whereas a negative counterexample must be excluded (as it is not a safe vertex). An existential counterexample $v \to (v_1 \vee \cdots \vee v_n)$ signals that the hypothesis is not existentially closed and requires that if a future hypothesis contains v, it also needs to contains an least one vertex of the vertices $v_1, \ldots, v_n$. Similarly, a universal counterexample $v \to (v_1 \wedge \cdots \wedge v_n)$ signals that the hypothesis is not universally closed and requires that if a future hypothesis contains v, it needs to contains all vertices $v_1, \ldots, v_n$. Note that existential and universal counterexamples are always finite objects since $E(\{v\})$ is finite for every $v \in V$.

We assume that the learner accumulates counterexamples in a so-called *game sample* $\mathcal{S}_\mathcal{G} = (S_+, S_-, S_\exists, S_\forall)$ consisting of a finite set S_+ of *positive counterexamples*, a finite set S_- of *negative counterexamples*, a finite set $S_\exists$ of *existential counterexamples*, and a finite set $S_\forall$ of *universal counterexamples*. After receiving a new counterexample, the task of the learner is then to generate a hypothesis $H \subseteq V$ that is *consistent* with the current game sample $\mathcal{S}_\mathcal{G}$ in the sense that

1. $v \in H$ for each $v \in S_+$;
2. $v \notin H$ for each $v \in S_-$;
3. $v \in H$ implies $\{v_1, \ldots, v_n\} \cap H \neq \emptyset$ for each $v \to (v_1 \vee \cdots \vee v_n) \in S_\exists$; and
4. $v \in H$ implies $\{v_1, \ldots, v_n\} \subseteq H$ for each $v \to (v_1 \wedge \cdots \wedge v_n) \in S_\forall$.

Once the learner has generated a new hypothesis, the feedback loop continues with the next iteration. This process repeats until a winning set is found.

Although infinite games involve two antagonistic players, we can reduce the learning setup outline above to the Horn-ICE framework of Section 2.1. The idea of this reduction is to replace each counterexample of a game sample $\mathcal{S}_\mathcal{G}$ in the following manner:

1. we interpret each positive counterexample $v \in S_+$ as a negative one,
2. we interpret each negative counterexample $v \in S_-$ as a positive one,
3. we replace each existential counterexample $v \to (v_1 \vee \cdots \vee v_n)$ with a Horn counterexample $(v_1 \wedge \cdots \wedge v_n) \to v$; and

4. we replace each universal counterexample $v \to (v_1 \wedge \cdots \wedge v_n)$ with n Horn counterexamples $v_1 \to v, \ldots, v_n \to v$.

It is not hard to verify that the resulting sample, which we denote by $\mathcal{S}_{Horn}$, is indeed a Horn-ICE sample. Moreover, $\mathcal{S}_{\mathcal{G}}$ and $\mathcal{S}_{Horn}$ are "equi-consistent" in the sense that every set $H \subseteq V$ of vertices is consistent with $\mathcal{S}_{\mathcal{G}}$ if and only if $V \setminus H$ is consistent with $\mathcal{S}_{Horn}$ (cf. Neider and Markgraf [9, Lemma 1]).[5] This reduction allows us to use any Horn-ICE learning algorithm to learn winning sets.

Our learning framework is straightforward to implement if the underlying game graph is finite, in which case the teacher can be built on top of an explicit representation of the game. However, if the underlying game graph becomes too large or is infinite, one has to choose a suitable representation for sets of vertices and the edge relation that allows performing operations on the graph symbolically. More precisely, the chosen symbolic representation must feature Boolean operations (i.e., union, intersection, and complementation), and the image $E(A)$ and preimage $E^{-1}(A)$ of symbolically represented sets $A \subseteq V$ need to be computable. Moreover, the emptiness problem (i.e., "given a set A, decide whether $A = \emptyset$") needs to be decidable, and it must be possible to extract an element from A if it is nonempty.

In the remainder of this section, we present implementations of our framework for two symbolic representations that satisfy these requirements: Section 3.1.1 covers safety games that are represented in terms of finite-state machines, called rational safety games [13], whereas Section 3.1.2 covers safety games that are represented in terms of linear real arithmetic, called LRA safety games [9]. Moreover, Section 3.1.3 introduces so-called regular safety games [5], which are a subclass of rational safety games that allow for a richer and, hence, more efficient learning setup.

To simplify the following description, let us assume that a winning set exists and can be represented in the chosen symbolic representation (e.g., as a formula in linear real arithmetic). In this case, the game sample is guaranteed to be *non-contradictory* in the sense that a consistent hypothesis in the chosen symbolic representation always exists. However, if no winning set exists or cannot be expressed symbolically, the feedback loop either repeats forever, or contradictory counterexamples get added to the game sample eventually. The latter can be detected (e.g., using a modified version of the pebbling algorithm of Section 2.2.2), in which case the feedback loop stops and reports that no winning set exists.

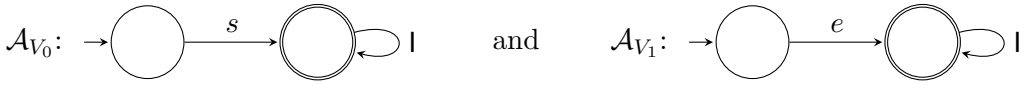
[5] By abuse of notation, we here use a Boolean formula $\gamma \colon V \to \mathbb{B}$, or classifier, generated by a Horn-ICE learner and the corresponding set $\{v \in V \mid \gamma(v) = \mathit{true}\}$ interchangeably.

3.1.1 Rational Safety Games

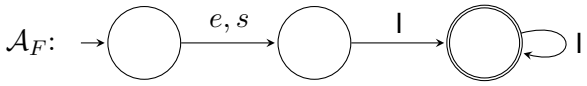
Our first symbolic representation of safety games, which we call *rational safety games*, uses finite-state machines to encode the (infinite) game graph [13]. Inspired by regular model checking [40], the key idea is to label each vertex of the game graph uniquely with a finite word over an a priori fixed alphabet Σ and represent sets of vertices by means of regular sets, accepted by nondeterministic finite automata (NFAs). Moreover, the edge relation is represented by a finite-state transducer. For the sake of simplicity, we do not distinguish between a vertex and its finite word representation.

To avoid cluttering this section with a series of formal definitions, let us introduce rational safety games through the example of Figure 3.1 (on Page 34). More precisely, let us consider the discretized version of the game in Figure 3.1b where the set of vertices is $V = \{e, s\} \times \mathbb{N}$.

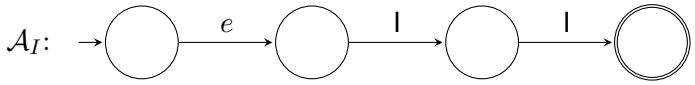
A straightforward encoding is to associate a vertex $(x, i) \in \{e, s\} \times \mathbb{N}$ with the finite word $x\mathsf{I}^i$ over the alphabet $\Sigma = \{e, s, \mathsf{I}\}$ where $x \in \{e, s\}$ and I^i is the (variable-length) unary encoding of i. Based on this encoding, we can use the two NFAs



to represent the vertices of Player 0 and Player 1, respectively.[6] Similarly, we can use the NFA



to represent the set F of safe vertices and the NFA



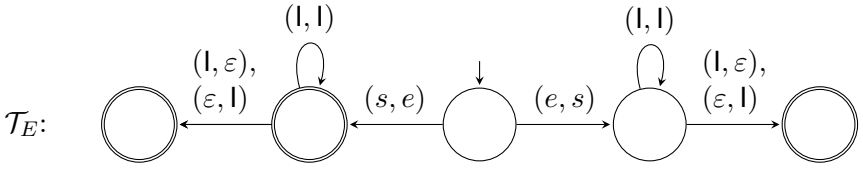
to represent the set I of initial vertices. Note that the sets V_0, V_1, and F are countably infinite in this example.

As mentioned above, we use transducers as a symbolic representation of the edge relation of a rational safety game. Intuitively, a *transducer* is an NFA over the extended

[6] We here use the usual graphical notation for NFAs: states are drawn as circles, accepting states are drawn as double-circles, the initial state has an incoming arrow, and transitions are drawn as arrows connecting states.

alphabet $\widehat{\Sigma} \coloneqq (\Sigma \cup \{\varepsilon\}) \times (\Sigma \cup \{\varepsilon\})$ that processes pairs $(u, v) \in \Sigma^* \times \Sigma^*$ of words. The empty word ε denotes that no letter is consumed in a computational step and allows relating words of different lengths. Note that the ability to process inputs asynchronously (as opposed to synchronous, letter-by-letter processing) makes this class of transducers expressive enough to encode a wide range of relations (called rational relations), including the configuration graphs of Turing machines.

The edge relation E of the game in Figure 3.1b can now be represented by the transducer



where the left branch corresponds to moves of Player 0, while the right branch corresponds to moves of Player 1. Note that the transitions labeled with (s, e) and (e, s) ensure that both players move in alternation and transitions labeled with $(\mathsf{I}, \varepsilon)$ and $(\varepsilon, \mathsf{I})$ capture the movement of the robot to the left and right, respectively. It is not hard to verify that this transducer accepts a pair $(x\mathsf{I}^i, y\mathsf{I}^j)$ of words if and only if $\big((x, i), (y, j)\big) \in E$. Thus, $\mathcal{T}_E$ captures exactly the edge relation of Figure 3.1b.

Since we represent sets of vertices using NFAs, each of the learner's hypotheses $H \subseteq V$ needs to be given in the form of an NFA, which we denote by $\mathcal{A}_H$. On the other hand, counterexamples are either words over the alphabet Σ (in the case of positive and negative counterexamples) or finite sets of words over Σ with logical constraints (in the case of existential and universal counterexamples). In this section, we assume the original learning setting of Figure 3.2 (on Page 35) and not the reduction to Horn-ICE learning; we use this reduction later for LRA games.

It is left to show how to implement a teacher and learner for rational safety games. We begin with a sketch of a generic teacher and then present two possible implementations for the learner.

A Teacher for Rational Safety Games

The teacher's task is to check whether a given NFA $\mathcal{A}_H$ represents a winning set, which amounts to checking the inclusion of initial vertices, the exclusion of non-safe vertices, and existential and universal closedness. Each of these checks can be performed using a series of standard constructions from automata theory (including intersection, complement, projection, and cylindrification). More precisely, for each of the four

properties of winning sets, we compute an NFA $\mathcal{A}$ such that the language of $\mathcal{A}$ is empty if and only if the hypothesis $\mathcal{A}_H$ satisfies the property. Moreover, if $\mathcal{A}_H$ does not satisfy the property, then each word u in the language of $\mathcal{A}$ is either immediately a counterexample (in the case of initial and safe vertices) or the left-hand side of an existential or universal counterexample. In the latter cases, existential or universal counterexample can simply be obtained by computing the image of the word u under the edge relation represented by the transducer $\mathcal{T}_E$. Note that the order in which the four properties are checked is not important for the correctness of our framework, but it might influence its performance. We refer the reader to Neider and Topcu [13, Section 4] for a detailed description.

Two Learners for Rational Safety Games

The task of a learner is to construct an NFA $\mathcal{A}$ that is consistent with the current game sample $\mathcal{S}_\mathcal{G} = (S_+, S_-, S_\exists, S_\forall)$. To this end, we have developed two learners, named SAT-Synth and RPNI-Synth, which we sketch below. For technical reasons, both learners generate deterministic finite automata (DFAs) rather than NFAs.

SAT-Synth The key idea underlying SAT-Synth is to reduce the learning problem to a series of constraint satisfaction problems in propositional logic and to use a highly optimized SAT solver to search for a solution. Given a game sample $\mathcal{S}_\mathcal{G} = (S_+, S_-, S_\exists, S_\forall)$, SAT-Synth creates and solves a sequence of propositional formulas $\left(\Phi_n^{\mathcal{S}_\mathcal{G}}\right)_{1,2,\dots}$ that have the following two properties:

- the formula $\Phi_n^{\mathcal{S}_\mathcal{G}}$ is satisfiable if and only if there exists a DFA with $n \in \mathbb{N}\backslash\{0\}$ states that is consistent with $\mathcal{S}_\mathcal{G}$; and
- a satisfying assignment v of $\Phi_n^{\mathcal{S}_\mathcal{G}}$ contains sufficient information to construct a DFA $\mathcal{A}_v$ that has n states and is consistent with $\mathcal{S}_\mathcal{G}$.

By starting with $n = 1$ and increasing n by one until $\Phi_n^{\mathcal{S}_\mathcal{G}}$ becomes satisfiable, we obtain an effective algorithm that learns consistent DFAs from game samples (see Neider and Topcu [13, Theorem 1]).

Given the properties of $\Phi_n^{\mathcal{S}_\mathcal{G}}$, it is not hard to verify that SAT-Synth possesses two crucial properties, which play an important role in proving the overall convergence of our framework: (a) SAT-Synth is guaranteed to terminate if $\mathcal{S}_\mathcal{G}$ is contradiction-free (since n will eventually be large enough), and (b) SAT-Synth learns minimal consistent DFAs (in terms of the number of states). Note, however, that it is already computationally hard to synthesize a minimal DFA that is consistent with positive and negative examples only [73], which justifies the use of a SAT solver for this task.

RPNI-Synth RPNI-Synth is based on the popular RPNI algorithm [120], which is a polynomial-time heuristic for learning DFAs from positive and negative words. Given a game sample $\mathcal{S}_\mathcal{G} = (S_+, S_-, S_\exists, S_\forall)$, it performs the following three steps:

1. RPNI-Synth determines a (minimal) set of words that the existential and universal counterexamples force to be included in a hypothesis H and adds this set to S_+. This computation can be done, for instance, using a modified version of the pebbling algorithm of Section 2.2.2.

2. RPNI-Synth constructs the so-called prefix-tree acceptor of S_+ (i.e., the tree-like DFA that accepts precisely the words in S_+). Note that this automaton is consistent with $\mathcal{S}_\mathcal{G}$.

3. RPNI-Synth successively tries to merges states of the prefix-tree acceptor, where a merge is considered successful if the resulting DFA remains consistent with $\mathcal{S}_\mathcal{G}$ (since merging states of a DFA increases the accepted language in terms of language inclusion, the merged DFA is guaranteed to be consistent with S_+, but might no longer consistent be with S_-, $S_\exists$, or $S_\forall$). If a merge was successful, RPNI-Synth proceeds to merge further states of the resulting DFA. If it was not successful, RPNI-Synth discards the current merge and proceeds with the DFA of the last successful merge (or the initial one if no merge was successful yet). The algorithm stops once there are no more merges left to try.

It is not hard to verify that RPNI-Synth indeed constructs a DFA that is consistent with $\mathcal{S}_\mathcal{G}$ (if $\mathcal{S}_\mathcal{G}$ is contradiction-free): it starts with a consistent one and discards any intermediate DFA that is not consistent with $\mathcal{S}_\mathcal{G}$. However, the resulting DFA might not be minimal. Hence, we encourage the reader to think of RPNI-Synth as a polynomial-time heuristic that runs in polynomial time but learns consistent DFAs that are not necessarily minimal.

The fact that SAT-Synth learns minimal consistent DFAs allows us to show that our framework converges to a winning set in finite time if one exists. This result is summarized below.

Theorem 3.1 (cf. Neider and Topcu [13, Theorem 2])**.** *Given a rational safety game $\mathcal{G}$, SAT-Synth is guaranteed to learn a winning set after a finite number of iterations if there exists one that is expressible as a DFA.*

The proof of the above theorem exploits that adding more and more counterexamples to a game sample ensures that hypotheses grow monotonically in size. Thus, all hypotheses of size less or equal to the size of a minimal DFA accepting a winning set will have been

exhausted eventually. Once this happens, SAT-Synth proposes a winning set. RPNI-Synth, on the other hand, does not have the same convergence guarantee. However, our experimental evaluation shows that RPNI-Synth effectively learns winning sets for most of our benchmarks.

3.1.2 LRA Safety Games

Our second symbolic representation of safety games, which we call *LRA safety games*, uses Linear Real Arithmetic (LRA) to encode the (infinite) game graph [9]. The key idea of this representation is to model vertices as d-dimensional real vectors and use quantifier-free first-order formulas in LRA to represent both sets of vertices and the edge relation. Note that LRA games are well suited for modeling cyber-physical systems and permit game graphs with uncountably many vertices.

Again, let us introduce LRA safety games through the example of Figure 3.1 (on Page 34). This time, however, we consider an undiscretized version of the game, where the robot can assume any position $x \in \mathbb{R}_{\geq 0}$ on the conveyor belt. A straightforward way to represent the game's vertices is to use two variables, p and x: the variable p can assume two values, 0 and 1, and models which player moves next; the variable x, on the other hand, corresponds to the position of the robot and is restricted to non-negative real values.

Based on the aforementioned encoding, we can use the two formulas

$$\varphi_{V_0}(p, x) \coloneqq p = 0 \wedge x \geq 0 \quad \text{and} \quad \varphi_{V_1}(p, x) \coloneqq p = 1 \wedge x \geq 0$$

to represent the vertices of Player 0 and Player 1, respectively. Similarly, we can use the formula

$$\varphi_I(p, x) \coloneqq (p = 0 \vee p = 1) \wedge x \geq 1$$

to represent the set F of safe vertices and the formula

$$\varphi_I(p, x) \coloneqq p = 1 \wedge (x \geq 1 \wedge x \leq 2)$$

to represent the set I of initial vertices.

Finally, we represent the transition relation E using the formula

$$\begin{aligned}\varphi_E(p, x, p', x') \coloneqq &\Big[p = 0 \wedge p' = 1 \wedge [(x \geq 1 \wedge x' = x - 1) \vee x' = x + 1 \vee x' = x]\Big] \vee \\ &\Big[p = 1 \wedge p' = 0 \wedge [(x \geq 1 \wedge x' = x - 1) \vee x' = x + 1]\Big],\end{aligned}$$

where the variables p, x encode the source vertex and the variables p', x' encode the destination vertex. Note that the first disjunct of formula φ_E corresponds to the moves of Player 0, while the second corresponds to the moves of Player 1.

Since we use quantifier-free first-order formulas in LRA to represent sets of vertices, each of the learner's hypotheses $H \subseteq V$ needs to be given in the form of such a formula, which we denote by φ_H. In contrast to rational safety games, we here consider the reduction to Horn-ICE learning as introduced on Page 36. This means that counterexamples are either d-dimensional real vectors (in the case of positive and negative counterexamples) or Horn counterexamples.

It is left to show how to implement a teacher and learner for rational safety games, which we do in the following.

A Teacher for LRA Safety Games

The teacher's task is to check whether a given LRA formula φ_H represents a winning set, which amounts to checking the inclusion of initial vertices, the exclusion of non-safe vertices, and existential and universal closedness. Each of these checks can be performed in a straightforward way using calls to an SMT solver (such as CVC4/CVC5 [28, 31] or Z3 [114]). More precisely, for each of the four properties of winning sets, we generate a formula ψ such that ψ is valid if and only if the hypothesis φ_H satisfies the property. Moreover, if φ_H does not satisfy the property, then a model for $\neg\psi$ can be used to derive a counterexample, which we then translate into the Horn-ICE setting as described on Page 36. Note again that the order in which the four properties are checked is not important for the correctness of our framework, but it might influence its performance. We refer the reader to Neider and Markgraf [9, Section V] for more details.

A Learner for LRA Safety Games

The task of a learner is to construct a quantifier-free LRA formula φ_H that is consistent with the current Horn-ICE sample $\mathcal{S} = (S_+, S_-, S_\Rightarrow)$. To this end, we can immediately apply any Horn-ICE learning algorithm described in Section 2.2. For instance, utilizing Horn-ICE-DT from Section 2.2.2 results in an effective learning algorithm for winning sets represented as decision trees. This new synthesis algorithm, which we call DT-Synth, inherits all advantageous properties from Horn-ICE-DT, including the guarantee to converge to a solution if one can be expressed as a decision tree. The theorem below summarizes this result.

Theorem 3.2 (cf. Neider and Markgraf [9, Theorem 4])**.** *Given an LRA safety game $\mathcal{G}$ and a finite set $\mathcal{P}$ of predicates over the symbolic representation of vertices, DT-Synth is guaranteed to learn a winning set after a finite number of iterations if there exists one that is expressible as a decision tree over $\mathcal{P}$.*

Analogous to Section 2.2.2, we can relax the requirement that the set of predicates is finite. More precisely, if $\mathcal{P}$ is enumerable, DT-Synth still converges to a winning set if one can be expressed as a decision tree over $\mathcal{P}$ (cf. Neider and Markgraf [9, Theorem 4]).

3.1.3 Regular Safety Games

Regular safety games [5] are a special case of rational safety games in which the edge relation E is given by so-called *length-preserving transducers*. This class of transducers disallows transitions of the form (ε, a) and (a, ε) for any symbol $a \in \Sigma$ and, hence, cannot relate words of different lengths. Consequently, the infinite graph encoded by a length-preserving transducer consists of an infinite collection of finite graphs where only the vertices represented by words of the same length can be connected.

Length-preserving transducers are a popular tool to model and verify parameterized systems (i.e., reactive systems with a parameterized number of interconnected components). A prototypical example is the Dining Philosopher Protocol, in which the parameter is the number n of philosophers, and one would like to prove liveness for all values of $n \geq 3$. In this section, however, we are not interested in verification but in synthesizing parameterized systems with safety objectives. Analogous to parameterized verification, our goal is to synthesize systems for all (potentially infinite) parameter values at once.

Figure 3.3 illustrates a parameterized version of the safety game of Figure 3.1 (on Page 34), where the parameter $n \geq 2$ corresponds to the (finite) width of the conveyor belt. In this example, a vertex $(x, i) \in \{e, s\} \times \mathbb{N}$ is represented by a finite word of the form $x\mathsf{O}^i\mathsf{I}\mathsf{O}^{n-i-1}$ over the alphabet $\Sigma = \{e, s, \mathsf{O}, \mathsf{I}\}$. Note that the graph is partitioned into an infinite number of finite graphs, where the representation of all vertices inside the same component have the same length.

The fact that (i) length-preserving transducers can only connect vertices represented by words of the same length and (ii) a unique largest winning set $W^\star$ (with respect to set inclusion) exists in every safety game[7] allows us to answer whether a vertex

[7] It is not hard to verify that winning sets are closed under union (i.e., if W and W' are two winning sets in a safety game $\mathcal{G}$, then $W \cup W'$ is also a winning set in $\mathcal{G}$). Thus, the largest winning set—with respect to set inclusion—is unique (see Markgraf et al. [5, Theorem 1]).

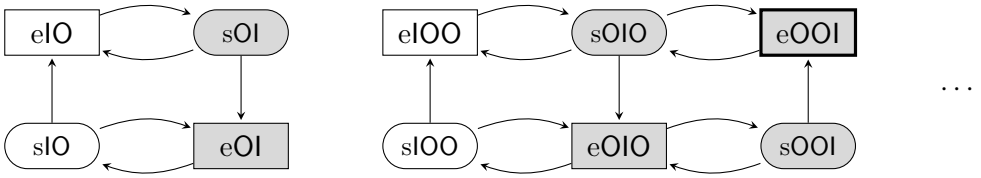


Figure 3.3: A parameterized version of the safety game of Figure 3.1 (on Page 34) for $n \geq 2$.

$u \in \Sigma^*$ is a member of the largest winning set. Indeed, answering this question is straightforward. Given a regular safety game $\mathcal{G} = (V_0, V_1, E, F, I)$ represented by NFAs $\mathcal{A}_{V_0}, \mathcal{A}_{V_1}, \mathcal{A}_F, \mathcal{A}_I$ and a length-preserving transducer $\mathcal{T}_E$, we first construct a finite safety game $\mathcal{G}_u = (V_0^u, V_1^u, E^u, F^u, I^u)$ where

- V_0^u and V_1^u are the finite restriction of the languages of $\mathcal{A}_{V_0}$ and $\mathcal{A}_{V_1}$ to vertices in $\Sigma^{|u|}$, respectively ($\Sigma^{|u|}$ here denotes the set of all words over Σ that have the same length as u);
- E^u is the finite restriction of the edge relation defined by $\mathcal{T}_E$ to vertices in $\Sigma^{|u|}$;
- $I^u = \{u\}$; and
- $F^u = \{v \in \Sigma^{|u|} \mid \mathcal{A}_F \text{ accepts } u\}$.

In a second step, we then solve the finite safety game $\mathcal{G}_u$ (e.g., using a fixed-point computation) to determine whether Player 1 can enforce the visit of a vertex outside F^u when starting in vertex u. Since computing winning strategies in finite safety games can be done in linear time (see Grädel, Thomas, and Wilke [74]), we obtain an effective procedure to answer whether a vertex belongs to the largest winning. Note that this membership query exploits the length-preserving nature of the transducer and is undecidable for rational safety games.

We now exploit the fact that deciding membership in the largest winning set is decidable to design a learning framework that is more efficient than the one for rational and LRA safety games. The key idea of this new framework, which is shown in Figure 3.4, is to (a) bias the learning process towards learning the largest winning set (rather than not caring about which winning set to learn) and (b) use simpler types of counterexamples (as compared to existential and universal counterexamples, which require non-trivial logical reasoning).

Our framework follows the *minimally adequate teacher (MAT)* principle proposed by Angluin [22] and supports two types of queries: membership queries and equivalence queries. In our setting, these two queries are as follows:

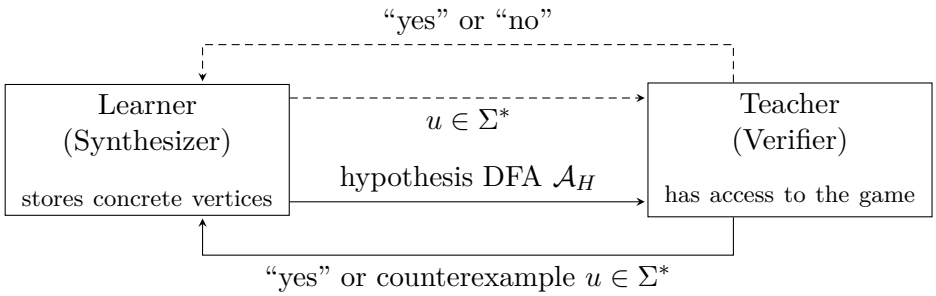


Figure 3.4: Our framework for learning winning sets in regular safety games [5]. Membership queries are shown as dashed arrows, while equivalence queries are shown as solid arrows.

- On a membership query, the learner provides a vertex $u \in \Sigma^*$, and the teacher has to check whether u is a member of the largest winning set $W^\star$.
- On an equivalence query, the learner conjectures a hypothesis DFA $\mathcal{A}_H$, and the teacher has to check whether $\mathcal{A}_H$ represents the largest winning set $W^\star$. If $\mathcal{A}_H$ represents $W^\star$, the teacher replies "yes". If not, the teacher replies with a counterexample $u \in \Sigma^*$ in the symmetric difference of the language of $\mathcal{A}_H$ and $W^\star$ (i.e., $\mathcal{A}_H$ does not accept $W^\star$, which is witnessed by u).

The feedback loop of our framework repeats until the teacher replies with "yes". By definition of equivalence queries, the learner has then conjectured a winning set, which we return. Note that the teacher is not allowed to reply with universal or existential counterexamples but must communicate why a conjecture is not existentially or universally closed based on a single word $u \in \Sigma^*$ (we show how this can be done shortly).

It is left to show how to implement a teacher and a learner for regular safety games, which we do next.

A Teacher for Regular Safety Games

The teacher's task is to answer membership and equivalence queries. We do this as follows:

Membership queries: On a membership query, the teacher needs to decide whether a given word $u \in \Sigma^*$ is a member of the largest winning set $W^\star$ and returns "yes" or "no" accordingly. This can be done as described above (see Page 45).

Equivalence queries: On an equivalence query, the teacher needs to check whether a given DFA $\mathcal{A}_H$ represents the largest winning set $W^\star$. This test can be performed similarly to rational safety games by checking all four properties of winning sets (cf. Section 3.1.1). However, we make the following modifications to the existential and universal closedness checks to account for the fact that a minimally adequate teacher cannot reply with existential or universal counterexamples (the checks for positive and negative counterexamples remain unchanged):

- If $\mathcal{A}_H$ is not existentially closed (which can be checked as described in Section 3.1.1), then there exists a vertex u that $\mathcal{A}_H$ accepts, but all of its successors $u_1, \ldots, u_n$ are rejected by $\mathcal{A}_H$. To determine how to resolve this issue, the teacher performs a membership query with u. If this membership query returns "no", the vertex u is not a member of the largest winning set and should be excluded; thus, the teacher returns u as a counterexample. If this membership query returns "yes", on the other hand, the vertex u is a member of the largest winning set and, hence, one (or more) of its successors need to be included in the next hypothesis as well. The teacher can test which ones by asking membership queries for each successor $u_1, \ldots, u_n$. Once one of these queries returns "yes" (which is guaranteed if a winning set exists), the teacher returns the corresponding vertex as a counterexample. (We comment shortly on the fact that returning a single successor vertex might not result in the largest winning set.)
- If $\mathcal{A}_H$ is not universally closed (which can be checked as described in Section 3.1.1), then there exists a vertex u that $\mathcal{A}_H$ accepts, but at least one of its successors $u_1, \ldots, u_n$ is rejected by $\mathcal{A}_H$. Again, the teacher performs a membership query with u to determine how to resolve this issue. If this membership query returns "no", the vertex u is not a member of the largest winning set and should be excluded; thus, the teacher returns u as a counterexample. If this membership query returns "yes", on the other hand, any successor that is not accepted by $\mathcal{A}_H$ is a counterexample, and the teacher returns one of them.

If the conjecture $\mathcal{A}_H$ passes all four checks, the teacher replies with "yes".

It is not hard to verify that such a learner satisfies the required protocol and can be implemented using standard automata constructions. Note that a teacher as defined above even replies "yes" if the conjectured DFA represents a winning set that is not necessarily the largest one: a check for existential closedness can only reply a single counterexample, which can cause other successor vertices that are also members of the largest winning set to be missing. However, since any winning set is sufficient for synthesizing a reactive system, we exploit this property of the teacher as a means to terminate the feedback loop early.

A Learner for Regular Safety Games

The task of the learner is to construct a DFA $\mathcal{A}_H$ that is consistent with the information obtained from membership and equivalence queries thus far. To this end, we can use any off-the-shelf learning algorithm that operates in the minimally adequate teacher setting, such as the algorithm by Angluin [22], the one by Rivest and Schapire [133], and the one by Kearns and Vazirani [95]. We call the resulting algorithm P-Synth (where the prefix "P" stands for "Parameterized").

Theorem 3.3 (cf. Markgraf et al. [5, Theorem 2])**.** *Given a regular safety game $\mathcal{G}$, P-Synth is guaranteed to learn a winning set if there exists one that is expressible as a DFA.*

The number of membership and equivalence queries that P-Synth asks depends on the actual learning algorithm used. In the case of Kearns and Vazirani's algorithm, for instance, the number of equivalence queries is at most n and the number of membership queries is in $\mathcal{O}(n(n+n|\Sigma|)+n\log m)$ where m is the length of the longest counterexample returned by the teacher (cf. Markgraf et al. [5, Proposition 1]).

3.2 Functional Synthesis

We now turn to the area of functional synthesis and present Alchemist-CS-DT [11, 12], a learning-based framework for synthesizing functions or—equivalently—loop-free program expressions from logical specifications. Similar to the synthesis techniques we have discussed in the previous section, Alchemist-CS-DT follows the principle of counterexample-guided inductive synthesis (CEGIS) [142]. This time, however, the feedback loop is more complex and cannot as easily be partitioned into a teacher and a learner as before.

The precise synthesis problem we tackle in this section is that of finding (the implementation of) a function f that satisfies a logical specification of the form $\forall\vec{x}\colon \psi(f,\vec{x})$, where ψ is a quantifier-free first-order formula over a logic with fixed interpretations of constants, functions, and relations (except for f). We here assume that the quantifier-free fragment of this logic admits a decidable satisfiability problem, and effective procedures for producing models of satisfiable formulas are available.

For the rest of this section, let f be a function symbol with arity n representing the target function that is to be synthesized. We assume that the specification is written in a first-order logic $\mathcal{L}$ over an arbitrary set of function symbols $\mathcal{F}$ (including a special symbol f), constant symbols $\mathcal{C}$, and relation symbols $\mathcal{R}$, all of which have

fixed interpretations, except for f. Moreover, we assume that $\mathcal{L}$ is interpreted over a countable universe D and, further, a constant symbol exists for every element of D. Without loss of generality, we allow negations only at atomic predicates and require that equality is a relation in the logic (with the standard model-theoretic interpretation).

The logical specification itself is of the form $\forall \vec{x}\colon \psi(f, \vec{x})$ where ψ is a first-order formula constructed according to the following grammar (with x being a variable, $g \in \mathcal{F}$, $c \in \mathcal{C}$, $R \in \mathcal{R}$, and $\vec{\tau}$ being a vector of terms with a suitable arity):

$$\begin{aligned} \text{Terms:} \quad & \tau ::= x \mid c \mid f(\tau_1, \ldots, \tau_n) \mid g(\vec{\tau}) \\ \text{Formulas:} \quad & \varphi ::= R(\vec{\tau}) \mid \neg R(\vec{\tau}) \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \end{aligned}$$

A prototypical example in real arithmetic is the specification

$$\psi(f, x_1, x_2) \coloneqq f(x_1, x_2) \leq x_1 \wedge f(x_1, x_2) \leq x_2 \wedge \big(f(x_1, x_2) = x_1 \vee f(x_1, x_2) = x_2\big),$$

which uniquely describes the maximum of the two arguments x_1 and x_2.

Given a specification $\forall \vec{x}\colon \psi(f, \vec{x})$, the synthesis problem is now to find a definition of f in the form of an expression e such that $\forall \vec{x}\colon \psi(e/f, \vec{x})$ is valid; here, $\psi(e/f, \vec{x})$ denotes that every occurrence of f in ψ is replaced with e (after renaming the free variables in e accordingly). In order to make this problem computationally tractable, we impose two restrictions: one on the type of functions (i.e., expressions) that we want to synthesize and one on the logical specification.

Let us begin with the restriction on the type of functions. Intuitively, we focus on a specific type of functions, named piece-wise, that partition the input domain into a finite set of regions and apply an $\mathcal{L}$-expression in each region. More formally, a *piece-wise function* is a nested if-then-else expression with free variables $y_1, \ldots, y_n$ that is constructed according to the grammar

$$\text{Expressions:} \quad e ::= c \mid y_i \mid g(\vec{e}) \mid \mathit{ite}\big(R(\vec{e}), e, e\big)$$

where $i \in \{1, \ldots, n\}$, $c \in \mathcal{C}$, $R \in \mathcal{R}$, and $g \in \mathcal{F} \setminus \{f\}$.[8] An example for a piece-wise function is the expression

$$e \coloneqq \mathit{ite}(y_1 \leq y_2, y_2, y_1),$$

which computes the maximum of y_1 and y_2 by partitioning the input into two regions ($y_1 \leq y_2$ and $y_1 > y_2$) and applying the two (sub-)expressions y_2 and y_1 to each region accordingly. Note that e is indeed a solution to the specification $\psi(f, x_1, x_2)$ above.

[8] If desired, our framework can be restricted to synthesize expressions that only use a subset $\hat{\mathcal{C}} \subseteq \mathcal{C}$ of constants, $\hat{\mathcal{R}} \subseteq \mathcal{R}$ of relations, and $\hat{\mathcal{F}} \subseteq \mathcal{F}$ of functions. This is useful in situations where one wants to exclude the use of certain constants, relations, or functions in the synthesized expression.

The restriction on the type of specifications, on the other hand, is more involved and requires a detailed discussion. We want to capture the intuitive property that a specification maps each input to an output independent of how other inputs are mapped to their outputs. Moreover, for any given expression e that does not satisfy the specification, it must be possible to algorithmically find a (not necessarily unique) input $\vec{\iota} \in D^n$ for e such that (a) there exists an evaluation $\vec{x}$ (from which $\vec{\iota}$ is derived) causing the formula $\psi(f, \vec{x})$ to evaluate to false, and (b) the falsehood is caused solely by the evaluation of e on $\vec{\iota}$. We call such specifications single-point refutable and use the input $\vec{\iota}$ as a means to correct an incorrect hypothesis in the CEGIS loop.

To make the notion of single-point refutable specifications mathematically precise, we introduce a first-order formula

$$iso_{\vec{u},v,b}(\psi),$$

where $\vec{u}$ is a vector of n first-order variables (n is the arity of the function to be synthesized), v is a first-order variable (different from ones in $\vec{u}$), and $b \in \{true, false\}$. Intuitively, this function, named *isolate transformer*, captures whether ψ evaluates to b if f maps the inputs given by the variables $\vec{u}$ to the output given by v and independent of how f is interpreted on other inputs. To implement $iso_{\vec{u},v,b}$, every function application of f is modified so that it evaluates to the value of v if the input matches $\vec{u}$ and to a special value $\perp \notin D$ if the input does not match $\vec{u}$ (the latter signals that the value of f should be "ignored" in these inputs). Functions on terms that involve $\perp$ are evaluated to $\perp$ as well. Additionally, relations are evaluated to b only if none of its arguments evaluates to $\perp$; otherwise, they are mapped to $\neg b$ so as to signal that none of them contributes to making ψ evaluate to b. Note that the formula $iso_{\vec{u},v,b}(\psi)$ is over the free variables $\vec{x}$, $\vec{u}$, and v, but the function symbol f does no longer occur (it is substituted by either v or $\perp$). Also, note that we cannot draw any conclusion in the case that $iso_{\vec{u},v,b}(\psi)$ evaluates to $\neg b$. We refer the reader to Neider, Saha, and Madhusudan [11] for a detailed description of the isolate transformer and an example.

We can now formally define single-point refutable specifications. To this end, let us fix a specification of the form $\forall \vec{x} \colon \psi(f, \vec{x})$. Then, we say that $\forall \vec{x} \colon \psi(f, \vec{x})$ is *single-point refutable* if for every expression $e \colon D^n \to D$ that does not satisfy the specification (i.e., the specification does not hold under this interpretation for f) there exists an interpretation for the variables $\vec{x}$ and an input $\vec{\iota}$ that is an interpretation for the variables $\vec{u}$ such that when the variable v is interpreted to be $e(\vec{\iota})$, the isolate transformer $iso_{\vec{u},v,false}(\psi)$ evaluates to *false* (cf. Neider, Saha, and Madhusudan [11, Definition 2.8]).

Let us illustrate the definition of single-point refutable specifications through the following examples. We fix the underlying logic to be the first-order theory of arithmetic.

- The specification

$$\forall x_1, x_2 \colon f(17, 21) = 16 \wedge f(76, 30) = 52 \wedge \cdots \wedge f(22, 45) = 15$$

  is single-point refutable. More generally, any set of input-output examples can be written as a conjunction of constraints that forms a single-point refutable specification.

- The specification

$$\forall x_1, x_2 \colon f(x_1, x_2) \leq x_1 \wedge f(x_1, x_2) \leq x_2 \wedge \big(f(x_1, x_2) = x_1 \vee f(x_1, x_2) = x_2\big)$$

  of the aforementioned maximum function is single-point refutable.

- The specification

$$\forall x \colon f(0) = 0 \wedge f(x + 1) = f(x) + 1$$

  is not single-point refutable. To see why, observe that the sub-formula $f(x+1) = f(x)+1$ relates two distinct inputs on which f is evaluated. Thus, if an expression e does not satisfy the specification (e.g., $e(i) = 0$ for all $i \in \mathbb{N}$), we cannot isolate a single input on which e is incorrect. This situation is captured by the isolate transformer: when the formula $iso_{\vec{u},v,b}(\psi)$ is parameterized with $b = \mathit{false}$, at least one of $f(x + 1)$ and $f(x)$ evaluates to $\perp$ and, hence, the whole formula cannot evaluate to *false*.

- Specifications arising from the Horn-ICE framework are not single-point refutable.

Let us now present our general framework for synthesizing piece-wise functions from single-point refutable specifications, named Alchemist-CS-DT [11, 12]. The high-level idea is to intertwine logical synthesis methods with highly scalable classification algorithms from machine learning. The former are used to generate appropriate predicates (to partition the input space into regions) and expressions (to apply in each region), whereas the task of the latter is to learn how to assemble the predicates and expressions into a piece-wise function that satisfies the specification.

Alchemist-CS-DT stores three finite sets, which can be accessed and modified globally: a set $\mathcal{P}$ of predicates over the background logic $\mathcal{L}$, a set E of expressions over $\mathcal{L}$, and a set $\mathcal{S}$ of multi-labeled (counter-)examples, where each example $(\vec{\iota}, Z) \in \mathcal{S}$ consists of an input $\vec{\iota} \in D^n$ and a set $Z \subseteq E$ of expressions. Moreover, it maintains the following two invariants:

1. for all examples $(\vec{\iota}, Z) \in S$, each expression $e \in Z$ is *suitable* for the input $\vec{\iota}$ in the sense that there exists a function g satisfying the specification such that $e(\vec{\iota}) = g(\vec{\iota})$; and

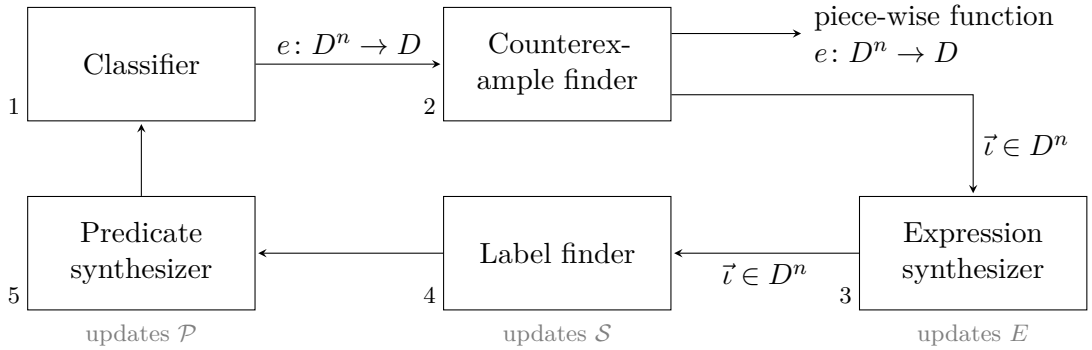


Figure 3.5: High-level view of Alchemist-CS-DT [11, 12].

2. for each pair $(\vec{\iota}_1, Z_1), (\vec{\iota}_2, Z_2) \in \mathcal{S}$ with $Z_1 \cap Z_2 = \emptyset$, there exists a predicate $p \in \mathcal{P}$ that can separate the inputs $\vec{\iota}_1, \vec{\iota}_2$ in the sense that $\vec{\iota}_1 \models p$ if and only if $\vec{\iota}_2 \not\models p$.

We provide further context for both invariants shortly.

Figure 3.5 presents a high-level view of Alchemist-CS-DT. Similar to Section 3.1 (as well as the ICE learning and Horn-ICE learning), our framework consists of a feedback loop that follows the principle of counterexample-guided inductive synthesis. Each iteration of this loop proceeds in five subsequent phases, which we describe below. Note that our framework does not define a single component for the teacher and the learner. However, we encourage the reader to think of Phase 2 as the teacher and the remaining phases as the learner.

Phase 1: Each iteration of the feedback loop starts by invoking a learning algorithm to learn a classifier $\kappa\colon D^n \to E$ over the set $\mathcal{P}$ of predicates that is consistent with the sample $\mathcal{S}$ in that $\kappa(\vec{\iota}) \in Z$ for each $(\vec{\iota}, Z) \in \mathcal{S}$. Intuitively, this means that κ divides the set of inputs into regions and maps each region to a single expression such that the mapping is consistent with the sample. We call this a *multi-label learning problem.*

A suitable type of classifier is the class of decision trees, which divides the input space into a finite number of disjoint subsets, each corresponding to a conjunction of (potentially negated) predicates from $\mathcal{P}$. To learn such decision trees, we have devised a novel learning algorithm tailored explicitly to the multi-label learning problem above. However, to avoid cluttering our presentation too much, we skip an in-depth description here and refer the reader to Neider, Saha, and Madhusudan [11, 12] for further details.

Once a consistent decision tree has been learned, we convert it into a piece-wise function in the logic $\mathcal{L}$. Our conversion recursively replaces each inner node of the tree with an if-then-else expression (to define the regions) and each leaf with the expression associated with that leaf (to define the output of the synthesized function). The resulting expression $e\colon D^n \to D$ is then handed over to the next phase.

Phase 2: Given an expression e, Alchemist-CS-DT now invokes the so-called *counterexample finder* to check whether e satisfies the specification. To this end, it calls an SMT solver to check whether the formula

$$\exists \vec{x} \exists \vec{u} \exists v\colon v = e(\vec{u}) \wedge \neg \mathit{iso}_{\vec{u},v,\mathit{false}}(\psi)$$

is satisfiable. By definition of single-point refutable specifications, we are guaranteed that this formula is satisfiable if and only if e does not satisfy the specification. Moreover, the valuation of the variables $\vec{u}$ in a model is a concrete input $\vec{\iota} \in D^n$ on which e is definitely wrong in the sense that there exists no function satisfying the specification that maps $\vec{\iota}$ to $e(\vec{\iota})$ (see Neider, Saha, and Madhusudan [11, Lemma 2.7]). Consequently, if the above formula is unsatisfiable, Alchemist-CS-DT stops and returns the current function e. If it is satisfiable, on the other hand, Alchemist-CS-DT hands $\vec{\iota}$ over to the next phase.

Phase 3: The input $\vec{\iota}$ serves as a counterexample to the current hypothesis and is passed to a so-called *expression synthesizer*. The goal of this synthesizer is to generate an expression $\hat{e}$ that is suitable for the input $\vec{\iota}$; remember that this means that there exists a function g satisfying the specification such that $\hat{e}(\vec{\iota}) = g(\vec{\iota})$. We facilitate this by generating a simpler synthesis problem with a specification of the form $\forall \vec{x}\colon \psi{\downarrow_{\vec{\iota}}}(\widehat{f}, \vec{x})$ where $\widehat{f}$ is a fresh uninterpreted function symbol and

$$\psi{\downarrow_{\vec{\iota}}}(\widehat{f}, \vec{x}) \coloneqq \mathit{iso}_{\vec{u},v,\mathit{true}}(\psi)[\vec{\iota}/\vec{u}, \widehat{f}(\vec{\iota})/v].$$

Intuitively, the formula $\psi{\downarrow_{\vec{\iota}}}(\widehat{f}, \vec{x})$ isolates the original specification to the input-output pair $(\vec{u}, v)$ and demands that it evaluates to *true* under this restriction. Then, we substitute $\vec{\iota}$ for $\vec{u}$ and a fresh function symbol $\widehat{f}$ (evaluated on $\vec{\iota}$) for v. As a result, we obtain that any expression $\hat{e}$ synthesized for $\widehat{f}$ maps $\vec{\iota}$ to a value that is consistent with the original specification (see Neider, Saha, and Madhusudan [11, Lemma 3.1]).

It is important to note that this new synthesis problem is more manageable than the original problem (since it only requires synthesizing an expression for a single input) and that we can use any existing expression synthesizer to solve it (e.g., enumeration-based or SMT-based methods as used in the SyGuS competition [18]). One of the most critical challenges in this context is to synthesize an expression

that is general in the sense that it works for all (or at least many) inputs in the region of $\vec{\iota}$. For the sake of brevity, however, we skip a detailed description here and refer the reader to Neider, Saha, and Madhusudan [11, 12], where we have developed an effective synthesizer for expression in linear integer arithmetic.

After the expression synthesizer has synthesized a suitable expression $\hat{e}$ for the counterexample $\vec{\iota}$, it adds $\hat{e}$ to the global set E (if this expression is not already present). Then, Alchemist-CS-DT passes $\vec{\iota}$ onto the next phase. Note that adding $\hat{e}$ to E maintains Invariant 1 above.

Phase 4: Give a counterexample $\vec{\iota}$, Alchemist-CS-DT invokes a so-called *label finder* to determine the set $Z \subseteq E$ of expressions that are suitable for $\vec{\iota}$. To this end, the label finder checks for each expression $e \in E$ whether the formula

$$\psi \downarrow_{\vec{\iota}} (e(\vec{\iota}) / \widehat{f}(\vec{\iota}), \vec{x})$$

is valid by means of an SMT solver. If the formula is valid, the label finder adds e to Z (which is initially empty); otherwise, it continues with the next expression in E. Once all suitable expressions for $\vec{\iota}$ have been identified, the label finder adds the pair $(\vec{\iota}, Z)$ as a new example to the sample $\mathcal{S}$.

If the expression $\hat{e}$ synthesized in the preceding phase was not already contained in E, Alchemist-CS-DT also needs to check whether $\hat{e}$ is suitable for any of the existing inputs in the sample $\mathcal{S}$. To this end, the label finder repeats the procedure laid out above for each such input and updates the sample $\mathcal{S}$ accordingly. Then, Alchemist-CS-DT proceeds to the next phase.

Phase 5: The goal of the last phase is ensure that the global set $\mathcal{P}$ of predicates satisfies Invariant 2 above (i.e., any two examples $(\vec{\iota}_1, Z_1), (\vec{\iota}_2, Z_2) \in \mathcal{S}$ with $Z_1 \cap Z_2 = \emptyset$ can be separated by a predicate $p \in \mathcal{P}$). Intuitively, if two examples $(\vec{\iota}_1, Z_1), (\vec{\iota}_2, Z_2) \in \mathcal{S}$ have no suitable expression in common, it means that the inputs $\vec{\iota}_1$ and $\vec{\iota}_2$ must not be mapped to the same region of the input space—otherwise, the synthesized function violates the specification on one of the inputs. Hence, if $Z_1 \cap Z_2 = \emptyset$, the set $\mathcal{P}$ needs to contain at least one predicate p such that $\vec{\iota}_1 \models p$ if and only if $\vec{\iota}_2 \not\models p$.

Alchemist-CS-DT delegates the task of constructing suitable predicates to a component that we call *predicate synthesizer*. This specialized synthesis task is again more manageable than the original one (since it only requires separating a finite number of data points), and we can either use existing SyGuS solvers [18] or classical machine learning algorithms to solve it. Once the predicate synthesizer has generated sufficiently many new predicts and added them to the global set $\mathcal{P}$, Alchemist-CS-DT continues with the next iteration of the feedback loop, which begins with Phase 1.

Alchemist-CS-DT's feedback loop continues until the counterexample finder (Phase 2) verifies that the current hypothesis expression e satisfies the specification. Once this happens, the feedback loop stops, and Alchemist-CS-DT returns e.

However, we cannot guarantee that the feedback loop always terminates in finite time (e.g., if no function exists that satisfies the specification). Nonetheless, our experimental evaluation shows that Alchemist-CS-DT effectively synthesizes functions for a wide range of real-world synthesis tasks.

3.3 Abstract Learning Framework for Synthesis

Virtually all learning-based synthesis techniques in current literature, including the ones presented in this and the previous chapter, rely on the principle of counterexample-guided inductive synthesis (CEGIS) [142], either implicitly or explicitly. In fact, CEGIS has emerged as a powerful paradigm in a host of different domains, including synthesizing program invariants for verification [48, 2, 66, 69, 3, 10], synthesizing program expressions [20, 11, 12, 135, 143], synthesizing string transformers for spreadsheets (e.g., Flash Fill [76]), reactive synthesis [5, 9, 13], and superoptimization [136], to name but a few.

To better understand the nuances of different synthesis approaches that use CEGIS, this section develops a general theory of CEGIS through a formalism we call an *abstract learning framework for synthesis* [4]. This framework aims to be general and abstract, encompassing a majority (if not all) known CEGIS frameworks and several other synthesis algorithms not generally viewed as falling under the umbrella of CEGIS. The ultimate goal is to provide a common set of concepts, definitions, and vocabulary that can be used to understand and combine learning-based synthesis techniques across different domains.

Figure 3.6 gives an overview of our abstract learning framework for synthesis. We first give a formal definition and then describe its components and their relations in detail. In particular, we defer the definition of the teacher and the learner to a later point in this section, when we have set up all necessary preliminaries.

Formally, an *abstract learning framework for synthesis (ALF)* [4] is a tuple

$$\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}}), \gamma, \kappa)$$

consisting of

- a class $\mathcal{C}$, called the *concept space*;
- a class $\mathcal{H}$, called the *hypothesis space*;

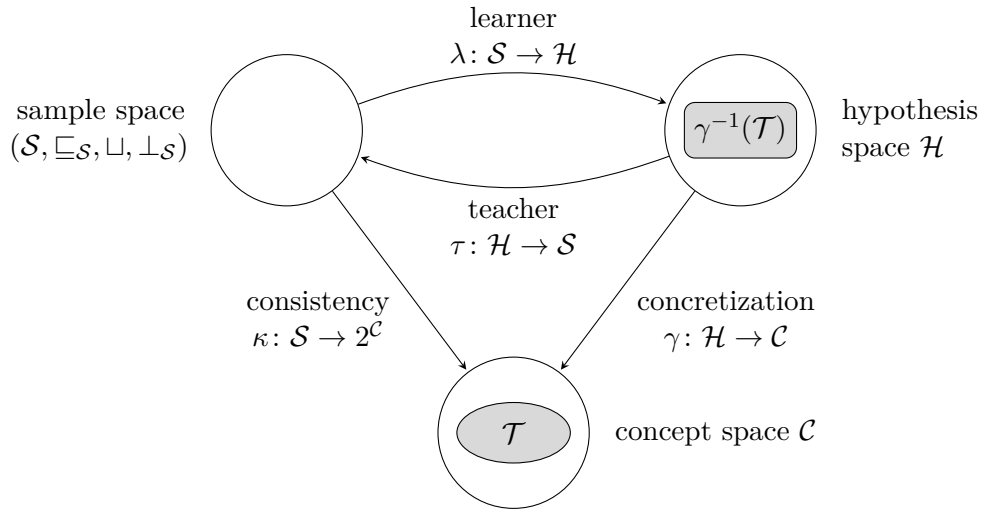


Figure 3.6: Components of our abstract learning framework for synthesis (ALF) [4].

- a class $\mathcal{S}$ that is equipped with a join semi-lattice $(\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}})$, called the *sample space*;
- a *concretization function* $\gamma\colon \mathcal{H} \to \mathcal{C}$; and
- a *consistency function* $\kappa\colon \mathcal{S} \to 2^{\mathcal{C}}$ that satisfies $\kappa(\bot_{\mathrm{s}}) = \mathcal{C}$ and $\kappa(S_1 \sqcup S_2) = \kappa(S_1) \cap \kappa(S_2)$ for all $S_1, S_2 \in \mathcal{S}$.

If the second condition for the consistency function is relaxed to $\kappa(S_1 \sqcup S_2) \subseteq \kappa(S_1) \cap \kappa(S_2)$, we speak of a *general ALF*. Moreover, we say that an ALF has a *complete* sample space if $(\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}})$ is a complete join semi-lattice (i.e., the join is defined for arbitrary subsets of $\mathcal{S}$). In this case, the consistency relation has to satisfy $\kappa(\bigsqcup \mathcal{S}') = \bigcap_{S \in \mathcal{S}'} \kappa(S)$ for each $\mathcal{S}' \subseteq \mathcal{S}$ (and $\kappa(\bigsqcup \mathcal{S}') \subseteq \bigcap_{S \in \mathcal{S}'} \kappa(S)$ for general ALFs).

As in computational learning theory (cf. Kearns and Vazirani [95]), we consider a concept space $\mathcal{C}$ that contains the objects that we are interested in. For example, in the Horn-ICE framework of Section 2.1, an element $C \in \mathcal{C}$ corresponds to a set of program configurations. In the functional synthesis setting of Section 3.2, the concept space could contain functions from $\mathbb{R}^d$ to $\mathbb{R}$.

The hypothesis space $\mathcal{H}$ contains the objects that a learning algorithm produces. These are representations of (some) elements from the concept space $\mathcal{C}$. For example, if $\mathcal{C}$ consists of sets of program configurations, then $\mathcal{H}$ could be the set of all conjunctive formulas over some set $\mathcal{P}$ of predicates (as in the case of the Sorcar algorithm). On the other hand, if $\mathcal{C}$ consists of functions from $\mathbb{R}^d$ to $\mathbb{R}$, then $\mathcal{H}$ could be the set of all functions expressible in Linear Real Arithmetic (as in Section 3.2).

The relation between hypotheses and concepts is given by a concretization function $\gamma\colon \mathcal{H} \to \mathcal{C}$ that maps hypotheses to concepts. This function defines the semantics of hypotheses and is usually given naturally.

In classical computational learning theory, one often assumes that the task is to learn a unique target concept, in which case samples typically consist of positive and negative examples. However, if the task is to infer a target concept that is not uniquely defined but should instead satisfy specific properties, just relying on positive and negative examples is often no longer sufficient. For instance, the Horn-ICE framework does not require the learner to find a specific invariant—as any invariant suffices—and introduces Horn counterexamples in addition to positive and negative examples as a means to refute an incorrect hypothesis.

To reflect this observation, we define the sample space in more general terms. More precisely, we equip the sample space with a bounded join semi-lattice $(\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}})$ where $\sqsubseteq_{\mathrm{s}}$ is a partial order over $\mathcal{S}$ with $\bot_{\mathrm{s}}$ as the least element and $\sqcup$ is the binary least upper-bound operator on $\mathcal{S}$ with respect to $\sqsubseteq_{\mathrm{s}}$. Intuitively, the teacher returns an element $S \in \mathcal{S}$ in order to provide some (new) information about a target concept; the learner, on the other hand, uses the join operator to combine the samples returned during the learning process. The least element $\bot_{\mathrm{s}}$ corresponds to the empty sample, and we encourage the reader to think of the join as the union of samples.

The consistency relation κ captures the semantics of samples with respect to the concept space by assigning to each sample $S \in \mathcal{S}$ the set $\kappa(S)$ of concepts that are consistent with the sample. The first condition on κ states that all concepts are consistent with the empty sample $\bot_{\mathrm{s}}$. The second condition states that the set of samples consistent with the join of two samples is precisely the set of concepts consistent with both of the samples. Intuitively, this means that joining samples does not introduce new inconsistencies, and existing inconsistencies transfer to larger samples. The condition $\kappa(S_1 \sqcup S_2) \subseteq \kappa(S_1) \cap \kappa(S_2)$ is natural in that it expresses that if a concept is consistent with the join of two samples, then the concept must be consistent with both of them individually. The condition $\kappa(S_1 \sqcup S_2) \supseteq \kappa(S_1) \cap \kappa(S_2)$, on the other hand, is debatable: it claims that samples taken together cannot eliminate a concept that they could not eliminate individually. To reflect this, we have introduced the notion of a general ALF. However, we have not found any natural example that requires such a generalization and, hence, restrict our attention to ALFs (instead of general ALFs) in the rest of this section.

For the remainder, it is helpful to define the set $\kappa_{\mathcal{H}}(S) \coloneqq \{H \in \mathcal{H} \mid \gamma(H) \in \kappa(S)\}$ of hypothesis that are consistent with $S \in \mathcal{S}$. Moreover, we say that a sample $S \in \mathcal{S}$ is *realizable* if there exists a hypothesis that is consistent with S (i.e., $\kappa_{\mathcal{H}}(S) \neq \emptyset$).

As we have argued above, most learning-based synthesis settings do not have a unique target concept that the learner needs to produce, but they only require the learner to synthesize some concept from an a priori fixed subset $\mathcal{T} \subseteq \mathcal{C}$ of the concept space. We call such a subset a *target specification* and define an *ALF instance* to be a pair $(\mathcal{A}, \mathcal{T})$ where $\mathcal{A}$ is an ALF over the concept space $\mathcal{C}$ and $\mathcal{T} \subseteq \mathcal{C}$ is a target specification. Moreover, we say that the target specification is *realizable by a hypothesis*, or simply realizable, if there is an $H \in \mathcal{H}$ such that $\gamma(H) \in \mathcal{T}$. For a hypothesis $H \in \mathcal{H}$, we usually write $H \in \mathcal{T}$ instead of $\gamma(H) \in \mathcal{T}$.

In our abstract learning framework for synthesis, a teacher and a learner interact with each other to synthesize an element $H \in \mathcal{H}$ such that $\gamma(H) \in \mathcal{T}$ (we formalize both the teacher and the learner shortly). However, there is a subtle point worth emphasizing. In most synthesis settings, the teacher does not explicitly know the target specification $\mathcal{T}$. Instead, it knows properties that define the target specification and can check whether a hypothesis $H \in \mathcal{H}$ satisfies these properties. For instance, in the Horn-ICE learning framework, the teacher does not know an inductive invariant but can check whether a hypothesis is one. The analogous observation from the area of functional synthesis is that the teacher knows the logical specification that a function should fulfill (and can check them), but it does not know a function that satisfies the specification.

Given an ALF $\mathcal{A} = (\mathcal{C}, \mathcal{H}, (\mathcal{S}, \sqsubseteq_\mathrm{s}, \sqcup, \bot_\mathrm{s}), \gamma, \kappa)$, we define a *learner* to be a function $\lambda \colon \mathcal{S} \to \mathcal{H}$ that assigns a hypothesis to every sample. Moreover, we call a learner λ *consistent* if $\gamma\big(\lambda(S)\big) \in \kappa(S)$ for all realizable samples $S \in \mathcal{S}$. Note that a learner is agnostic of the target specification of an ALF instance.

A teacher, on the other hand, is defined with respect to an ALF instance $(\mathcal{A}, \mathcal{T})$ with $\mathcal{A}$ as above and $\mathcal{T} \subseteq \mathcal{C}$. Formally, a *teacher* is a function $\tau \colon \mathcal{H} \to \mathcal{S}$ that satisfies the following two properties, which we call progress and honesty:

- $\tau(H) = \bot_\mathrm{s}$ for all hypotheses $H \in \mathcal{T}$, and $\gamma(H) \notin \kappa\big(\tau(H)\big)$ for all $H \notin \mathcal{T}$ (*progress*); and
- $\mathcal{T} \subseteq \kappa\big(\tau(H)\big)$ for each $H \in \mathcal{H}$ (*honesty*).

Intuitively, the progress property states that if a hypothesis is in $\mathcal{T}$, then the teacher must return the "empty" sample $\bot_\mathrm{s}$, signaling that the learner has learned a target; if this is not the case, the teacher must return a sample that rules out the current hypothesis. Note that this ensures that a consistent learner can never propose the same hypothesis twice and, hence, makes progress. On the other hand, honesty demands that the teacher only returns samples that are consistent with all target concepts. The latter property ensures that the teacher does never rule out a hypothesis in the target specification.

Similar to the principle of counterexample-guided inductive synthesis, the learner and teacher interact iteratively (see Figure 3.6 on Page 56): in each round, the learner proposes a hypothesis $\lambda(S)$ for the current sample $S \in \mathcal{S}$ and adds the feedback $\tau(\lambda(S))$ of the teacher to obtain the new sample. The combined behavior of the teacher and learner in one round is then the function $f_{\tau,\lambda} \colon \mathcal{S} \to \mathcal{S}$ with $f_{\tau,\lambda}(S) := S \sqcup \tau(\lambda(S))$. Moreover, we can describe the overall learning process by a transfinite sequence of samples $\langle S^{\alpha}_{\tau,\lambda} \mid \alpha \in \mathbb{O} \rangle$, where $\mathbb{O}$ denotes the class of all ordinals, that is defined by

- $S^{0}_{\tau,\lambda} := \bot_{\mathrm{s}}$;
- $S^{\alpha+1}_{\tau,\lambda} := f_{\tau,\lambda}(S^{\alpha}_{\tau,\lambda})$ for successor ordinals; and
- $S^{\alpha}_{\tau,\lambda} := \bigsqcup_{\beta<\alpha} S^{\beta}_{\tau,\lambda}$ for limit ordinals.

If the sample lattice is not complete, the above definition is restricted to the first two items and yields a sequence indexed by natural numbers. The learning stops once the learner proposes a hypothesis H with $H \in \mathcal{T}$.

The following lemma states that the teacher's properties of progress and honesty transfer to the iterative setting for consistent learners if the target specification is realizable. The proof is a transfinite induction, which crucially relies on the properties of the teacher and the consistency relation.

Lemma 3.1 (cf. Löding, Madhusudan, and Neider [4, Lemma 1])**.** *Let $\mathcal{T}$ be realizable, λ be a consistent learner, and τ be a teacher. If $(\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}})$ is a complete sample lattice, then*

1. *the learner makes progress in that for all $\alpha \in \mathbb{O}$ either $\kappa(S^{\alpha}_{\tau,\lambda}) \supsetneq \kappa(S^{\alpha+1}_{\tau,\lambda})$ and $\lambda(S^{\alpha}_{\tau,\lambda}) \notin \kappa(S^{\alpha+1}_{\tau,\lambda})$ holds or $\lambda(S^{\alpha}_{\tau,\lambda}) \in \mathcal{T}$ holds; and*
2. *the sample sequence is consistent with the target specification in that $\mathcal{T} \subseteq \kappa(S^{\alpha}_{\tau,\lambda})$ holds for all $\alpha \in \mathbb{O}$.*

If $\mathcal{S}$ is a non-complete sample lattice, then Items 1 and 2 hold for all $\alpha \in \mathbb{N}$.

From the lemma above, we can conclude that the transfinite sequence of hypotheses constructed by the learner converges to a hypothesis in the target specification. This fact is formalized next.

Theorem 3.4 (cf. Löding, Madhusudan, and Neider [4, Theorem 1])**.** *Let $(\mathcal{S}, \sqsubseteq_{\mathrm{s}}, \sqcup, \bot_{\mathrm{s}})$ be a complete sample lattice, $\mathcal{T}$ be realizable, λ be a consistent learner, and τ be a teacher. Then, there exists an ordinal $\alpha \in \mathbb{O}$ such that $\lambda(S^{\alpha}_{\tau,\lambda}) \in \mathcal{T}$.*

Theorem 3.4 ratifies the choice of our definitions, and its proof crucially relies on all aspects of our definitions (i.e., the honesty and progress properties of the teacher, the condition imposed on κ in an ALF, the notion of consistent learners, and so on). In fact, one can view our definitions as axiomatic because omitting any of the properties no longer allows us to prove convergence to a target concept.

Although Theorem 3.4 demonstrates that our framework is set up correctly, it only shows transfinite convergence, which is often referred to as convergence in the limit. However, convergence in finite time is clearly the more desirable notion, and we now sketch two strategies for designing learners that converge in finite time: finite hypothesis spaces and Occam learners.[9] To simplify the following presentation, we say that a learner λ converges for a teacher τ if there exists a natural number $n \in \mathbb{N}$ such that $\lambda(S^n_{\lambda,\tau}) \in \mathcal{T}$, which means that the learner produces a target hypothesis after n steps. Furthermore, we say that λ converges if it converges for every teacher.

Finite Hypothesis Spaces It is not hard to verify that if the hypothesis space (or the concept space) is finite, then any consistent learner converges: by Lemma 3.1, the learner always makes progress and, hence, never proposes two hypotheses that correspond to the same concept. Consequently, the learner only produces a finite number of hypotheses before finding one in the target (or it declares that no such hypothesis exists).

In fact, many synthesis engines rely on a finite hypothesis space. Examples include Houdini [66] (when viewed as a Horn-ICE learner) as well as Sorcar and Horn-ICE-DT when used over a finite set of predicates (see Section 2.2). Moreover, most SyGuS solvers [18, 19] provide a simple mechanism to restrict the hypothesis space to a finite set through syntactic constraints (i.e., grammars that only permit a finite number of derivations).

Occam Learners Our second tactic for convergence relies on the Occam razor principle, which intuitively states that the simplest concept—or theory—that explains a set of observations is preferable as a virtue in itself. To formalize this idea, we assume that the hypothesis space is equipped with a total quasi-order $\prec$. A quasi-order (also called pre-order) is a transitive and reflexive relation, and the relation being total means that $H \prec H'$ or $H' \prec H$ holds for all $H, H' \in \mathcal{H}$. The difference to an ordinary order relation is that $H \prec H'$ and $H' \prec H$ can hold in a quasi-order even if $H \neq H'$. We call a total quasi-order over the hypothesis space a *complexity ordering*, as it intuitively provides a means to measure the

[9] We have also developed a third, novel mechanism to guarantee convergence in finite time based on so-called tractable well-founded quasi-orders. For the sake of brevity, however, we omit a discussion and refer the reader to Löding, Madhusudan, and Neider [4] for details.

complexity of a hypothesis. A prototypical example is to order DFAs according to their number of states.

An *Occam learner* is now a learner that always constructs hypotheses that are minimal with respect to a fixed complexity order $\prec$. In fact, we can show that if $\mathcal{T}$ is realizable and λ is a $\prec$-Occam learner, then λ converges to a $\prec$-minimal target element (cf. Löding, Madhusudan, and Neider [4, Theorem 2]). To see why this is the case, pick any target element $T \in \gamma^{-1}(\mathcal{T})$, which exists because $\mathcal{T}$ is realizable. Since τ is honest, $T \in \kappa(S^n_{\tau,\lambda})$ for all $n \in \mathbb{N}$ by Lemma 3.1. Thus, on the iterated sample sequence, a $\prec$-Occam learner never constructs an element strictly above T with respect to $\prec$. Since there are only finitely many hypotheses that are not strictly above T, and since the learner always makes progress according to Lemma 3.1, it converges to a target element in finitely many steps. This target element does not have any other target elements below it and, thus, is $\prec$-minimal.

Several existing algorithms in the literature are Occam learners, including several enumeration-based solvers for SyGuS [18, 19]. Examples from this work are the extensions of Sorcar and Horn-ICE-DT over infinite sets of predicates (see Section 2.2) as well as the SAT-Synth algorithm described in Section 3.1.1.

Finally, let us note that not all synthesis techniques can be phrased in terms of an ALF. For instance, the P-Synth algorithm of Section 3.1.3 actively queries the teacher for membership information, which cannot (yet) be modeled in our framework. An extension of our abstract learning framework for synthesis to active learning would be interesting future work.

3.4 Notes on Related Work

In the area of reactive synthesis, games over various types of infinite graphs have been studied, predominantly in the context of pushdown graphs [101]. For more general classes of game graphs, a constraint-based approach [34], relying on constraint solvers such as Z3 [114], and various learning-based approaches have been proposed, including the methods for safety games presented in this chapter [5, 9, 13] as well as an earlier approach for reachability games [115]. In the context of safety games over finite graphs, recent work [116] has demonstrated the ability of learning-based techniques to extract small implementations of reactive systems from pre-computed systems with a potentially large number of states.

The framework of regular model checking [40], on which P-Synth, RPNI-Synth, and SAT-Synth are based, is used in a number of different domains to verify properties such as safety [40, 49, 81, 117] and liveness [107, 126]. In particular, for verification

of safety and liveness properties in parameterized systems, regular model checking has successfully been used [49, 107]. Similar to our setting, the approaches by Chen et al. [49] and Lin and Rümmer [107] also employ Angluin-style active learning of deterministic finite automata to verify properties of parameterized systems.

An orthogonal approach to the algorithms presented in Section 3.1 is to apply decision tree learning as a post-processing step after a strategy has been synthesized in a classical manner [43]. The underlying idea is to generate a concise representation of the given strategy, which is helpful to reduce the memory requirement or when one is interested in inspecting and understanding the strategy.

In the area of functional synthesis, our task is closely related to the syntax-guided synthesis framework (SyGuS) [18], which provides a standardized input language similar to SMTLib [30], to describe synthesis problems. Several solvers following the counterexample-guided inductive synthesis (CEGIS) approach [142] for SyGuS have been developed, including enumerative solvers, solvers based on constraint solving, one based on stochastic search, and one based on the program synthesizer Sketch [141]. A solver based on CVC4 [31] has also been presented. We refer the reader to the results of the SyGuS competition [19] for an in-depth discussion of available solvers.

There has also been work on synthesizing piece-wise affine models of hybrid dynamical systems from input-output examples [21, 32, 64] (we refer the reader to Paoletti et al. [122] for a comprehensive survey). The setting there is to learn an affine model passively (i.e., without feedback on whether the synthesized model satisfies some specification), and, consequently, the learned model only approximates the actual system. A tool for learning guarded affine functions, which uses a CEGIS approach, is Alchemist [135]. In contrast to the setting of Section 3.2, however, Alchemist requires that the specification uniquely determines the function to synthesize.

The abstract learning frameworks for synthesis presented in this chapter generalize the principle of counterexample-guided inductive synthesis [142] and can be used to model almost all learning-based synthesis approaches found in the literature. Examples include synthesizing loop-free programs [77], synthesizing synchronizing code for concurrent programs [47], work on using synthesis to mine specifications [92], synthesizing bit-manipulating programs and deobfuscating programs [89], superoptimization [136], deductive program repair [98], synthesis of recursive functional programs over unbounded domains [99], as well as synthesis of protocols using enumerative CEGIS techniques [146]. An example for employing a human as a teacher is Flashfill by Gulwani [76], which synthesizes string manipulation macros from user-given input-output examples in Microsoft Excel.

Finally, it is worth mentioning that Jha and Seshia [90] have independently proposed a framework similar to ours. Their work focuses on improving the understanding of the

relative power of CEGIS variants when the types of counterexamples vary that the teacher is allowed to return. Moreover, the authors study the impact of bounding the memory available to the learner.

4 INTELLIGENT SPECIFICATION OF SYSTEM PROPERTIES

Virtually all formal methods assume that the (formal) specifications are available in a suitable format, are functionally correct, and express precisely the properties the engineer has in mind. However, all of these assumptions are often unrealistic in practice. In fact, formalizing system requirements is known to be notoriously difficult and error-prone. Even worse, the training effort required to reach proficiency with formal methods (including their underlying specification languages) can be disproportionate to the expected benefits [147], and the use of formalisms such as temporal logics requires a level of sophistication that many users might never achieve [83].

To alleviate this severe practical problem, we have developed a series of methods to learn formal specifications, thereby removing the need to write specifications by hand. The key idea is that an engineer provides examples of a system's desired or undesired behavior (e.g., traces of executions that the specification should allow or disallow), and an automated tool then learns a specification that is consistent with the given examples. Note that learning specifications "completely from scratch" is only a first step towards a computer-aided approach to writing formal specifications. Ideally, an engineer would write a high-level sketch of a specification, and an automated tool would fill out missing parts—either in interaction with the engineer or based on given examples. We further comment on this vision in Chapter 5, where we discuss future work.

Throughout this chapter, we consider non-terminating systems whose behavior can be modeled by infinite words. To make this notion mathematically precise, let $\mathcal{P}$ be a set of atomic propositions (or predicates), which capture the relevant properties of the system in question. Then, a non-terminating execution of a system is an infinite word $\alpha = a_0 a_1 \cdots \in (2^{\mathcal{P}})^{\omega}$, where the symbol $a_t \in 2^{\mathcal{P}}$ corresponds to sets of propositions that are true at time point $t \in \mathbb{N}$. To ease our notation, we use $\alpha[t, \infty) = a_t a_{t+1} \cdots$ to denote the infinite suffix of α starting at position $t \in \mathbb{N}$, and we denote the infix of α starting at position t and leading up to (but excluding) position $t' \geq t$ by $\alpha[t, t') = a_t \cdots a_{t'-1}$. Moreover, we address the symbol at position $t \in \mathbb{N}$ of an infinite word by $\alpha[t]$ (i.e., $\alpha[t] = \alpha[t, t+1)$). Note that $\alpha[t, t)$ is the empty word ε for every $t \in \mathbb{N}$.

While there exists a wide range of specification formalisms in the literature, we focus on two specific logics that allow reasoning about temporal properties of systems: Linear Temporal Logic (LTL) [125] and the core fragment of the Property Specification Language (PSL) [61]. This choice is motivated by the fact that both LTL and PSL have been widely adopted in practice and constitute the de facto standard for expressing properties of reactive systems (e.g., PSL has been canonized as IEEE standard 1850 [24]).

Moreover, both logics share another important property: two LTL (or PSL) formulas φ and ψ are equivalent if and only if they are satisfied by the same ultimately-periodic words (i.e., infinite words of the form $uv^\omega = uvvv\cdots \in (2^{\mathcal{P}})^\omega$ where $u \in (2^{\mathcal{P}})^*$ is a finite word and $v \in (2^{\mathcal{P}})^+$ is a non-empty finite word).[10] This property allows us to restrict our setting without loss of generality to learning from ultimately-periodic words. However, it is essential to mention that our algorithms translate to learning LTL and PSL specifications over finite words in a straightforward manner.

The remainder of this chapter considers two distinct settings. In Section 4.1, we assume that an engineer can provide both positive and negative examples of a specification, which correspond to system behaviors that the specification should allow or prohibit, respectively. This setting arises naturally when the engineer is tasked with writing a formal specification for a new system yet to be developed. However, a learning-based approach is also helpful in situations where one wants to infer a specification of an existing system, which is an essential task in many applications, including explainable artificial intelligence, modernization of legacy software, and so on. The problem here is that one only has access to positive examples: one can only observe what a system can do but never observe what it cannot do. Section 4.2 addresses this issue and shows how to learn specifications from positive examples only. We conclude this chapter in Section 4.3 with a brief discussion of related work.

4.1 Learning Specifications from Positive and Negative Examples

In this section, we present two algorithms for learning temporal logic specifications from positive and negative examples. We begin in Section 4.1.1 with a learning algorithm for LTL [125]. In Section 4.1.2, we then show how to extend this algorithm to learn formulas in the core fragment of PSL [61].

Throughout this section, we assume that the data to learn from is given as two (potentially empty) finite, disjoint sets $S_+, S_- \subset (2^{\mathcal{P}})^\omega$ of ultimately-periodic words $uv^\omega \in (2^{\mathcal{P}})^\omega$. The words in S_+ are interpreted as positive examples, while the words in S_- are interpreted as negative examples. Similar to Chapters 2 and 3, we call the pair $\mathcal{S} = (S_+, S_-)$ a *sample*.

[10] This fact is a consequence of the following two results. First, if φ is an LTL or PSL formula, then the set of all infinite words satisfying φ is ω-regular (i.e., can be recognized by a Büchi automaton [23, 125]). Second, two ω-regular languages L_1, L_2 coincide (i.e., $L_1 = L_2$) if and only if both contain the same ultimately-periodic words. Note that the latter is a folklore result, which one obtains from the characterization of ω-regular languages in terms of Büchi automata.

4.1.1 Learning LTL Specifications from Positive and Negative Examples

Linear Temporal Logic (LTL) [125] is an extension of propositional Boolean logic with modalities that allow expressing temporal properties. Starting with a finite, non-empty set $\mathcal{P}$ of atomic propositions, formulas in LTL are inductively defined according to the following grammar:

$$\varphi ::= p \in \mathcal{P} \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\,\varphi \mid \varphi\,\mathsf{U}\,\varphi$$

The reader should interpret the temporal operator X as "next" and U as "until". We also allow syntactic sugar in the form of the formulas $\mathit{true} \coloneqq p \vee \neg p$ for some $p \in \mathcal{P}$, $\mathit{false} \coloneqq \neg\mathit{true}$, as well as $\psi \wedge \varphi$ and $\psi \rightarrow \varphi$, which are defined as usual. Moreover, we allow the additional temporal formulas $\mathsf{F}\,\psi \coloneqq \mathit{true}\,\mathsf{U}\,\psi$ ("finally") and $\mathsf{G}\,\psi \coloneqq \neg\,\mathsf{F}\,\neg\psi$ ("globally"). The *size* of an LTL formula φ, which we denote by $|\varphi|$, is the number of its unique subformulas. Finally, let $\Lambda_{ltl} = \{\neg, \vee, \mathsf{X}, \mathsf{U}\}$ be the set of LTL operators (to which we add the syntactic sugar if desired).

LTL formulas are interpreted over infinite words $\alpha \in (2^{\mathcal{P}})^{\omega}$ (i.e., infinite sequences of sets of atomic propositions, modeling which facts about a system are true at different points in time). Similar to propositional logic, the semantics of LTL is defined in terms of a satisfaction relation $\models$, which formalizes when an infinite word $\alpha \in (2^{\mathcal{P}})^{\omega}$ *satisfies* an LTL formula:

- $\alpha \models p$ if and only if $p \in \alpha[0]$;
- $\alpha \models \neg\varphi$ if and only if $\alpha \not\models \varphi$;
- $\alpha \models \varphi_1 \vee \varphi_2$ if and only if $\alpha \models \varphi_1$ or $\alpha \models \varphi_2$;
- $\alpha \models \mathsf{X}\,\varphi$ if and only if $\alpha[1, \infty) \models \varphi$ or $\alpha \models \varphi_2$; and
- $\alpha \models \varphi_1\,\mathsf{U}\,\varphi_2$ if and only if there exists a $t \in \mathbb{N}$ such that $\alpha[t, \infty) \models \varphi_2$ and $\alpha[t', \infty) \models \varphi_1$ for each $t' \in \{0, \ldots, t-1\}$.

Note that the satisfaction of a formula, due to the temporal operators, depends on the satisfaction of its subformulas on (potentially different) infinite suffixes of α.

A prototypical example of an LTL formula is the request-response property $\mathsf{G}(p \rightarrow \mathsf{F}\,q)$. This formula states that every request p needs to be answered eventually by a response q (sometimes $\mathsf{X}\,\mathsf{F}\,q$ is used instead of $\mathsf{F}\,q$, excluding that the response arrives at the same point in time as the request is sent). Thus, the ultimately-periodic word $(\{p\}\{q\})^{\omega}$ satisfies the formula, while $\{q\}(\{p\})^{\omega}$ does not.

Having defined the syntax and semantics of LTL, we can now formalize our learning setup. First, we call an LTL formula φ *consistent* with a sample $\mathcal{S} = (S_+, S_-)$ if $\alpha \models \varphi$ for each $uv^{\omega} \in S_+$ (i.e., all positive examples satisfy φ) and $\alpha \not\models \varphi$ for each $uv^{\omega} \in S_-$

(i.e., all negative examples violate φ). Then, the fundamental learning task we solve in this section is the following:

> *"given a sample $\mathcal{S}$, compute an LTL formula of minimal size that is consistent with $\mathcal{S}$".*

Note that the above task asks to construct an LTL formula that is minimal among all consistent formulas. The motivation for this requirement is threefold. Firstly, we observe that the problem becomes simple without a restriction on the size: for $\alpha \in S_+$ and $\beta \in S_-$, one can easily construct a formula $\varphi_{\alpha,\beta}$ with $\alpha \models \varphi_{\alpha,\beta}$ and $\beta \not\models \varphi_{\alpha,\beta}$, which describes the first symbol where α and β differ using a sequence of X-operators and an appropriate propositional formula; then, $\varphi^\star_{\mathcal{S}} \coloneqq \bigvee_{\alpha \in S_+} \bigwedge_{\beta \in S_-} \varphi_{\alpha,\beta}$ is trivially consistent with $\mathcal{S}$. However, simply enumerating all differences of a sample is clearly of little help towards learning a specification that describes the behavior of a system on more than just the given examples. Secondly, small formulas tend to provide a good generalization of the behavior represented by the sample and avoid the possibility of simply overfitting it. Thirdly, small formulas are simpler for humans to interpret, which justifies spending effort on learning formulas that are as small as possible.

Algorithm 3 now presents our learning algorithm for LTL formulas from positive and negative examples in pseudo code. The key idea underlying this algorithm is to reduce the construction of a minimally consistent LTL formula to a series of constraint satisfaction problems in propositional logic and to use a highly optimized SAT solver to search for a solution. More precisely, given a sample $\mathcal{S}$ and a natural number $n \in \mathbb{N} \setminus \{0\}$, we construct a propositional formula $\Phi^{\mathcal{S}}_n$ that has the following two properties:

1. $\Phi^{\mathcal{S}}_n$ is satisfiable if and only if there exists an LTL formula of size n that is consistent with $\mathcal{S}$; and

2. if v is a model of $\Phi^{\mathcal{S}}_n$, then v contains sufficient information to construct an LTL formula φ_v of size n that is consistent with $\mathcal{S}$.

By starting with $n = 1$ and incrementing n until $\Phi^{\mathcal{S}}_n$ becomes satisfiable, we obtain an effective learning algorithm for specifications expressed in LTL.

On a technical level, the formula $\Phi^n_{\mathcal{S}}$ is the conjunction

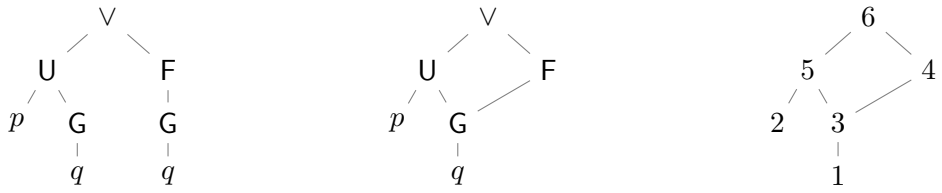
$$\Phi^n_{\mathcal{S}} \coloneqq \Phi^{syn}_n \wedge \Phi^{sem}_n,$$

where Φ^{syn}_n encodes the syntactic structure of the prospective LTL formula and Φ^{sem}_n imposes semantic constraints that enforce that the prospective LTL formula is consistent with the sample. In the remainder of this section, we describe both Φ^{syn}_n and Φ^{sem}_n in detail. Moreover, we show the termination and correctness of our learning algorithm.

Algorithm 3: SAT-based learning algorithm for LTL specifications [7].

Input: a sample $\mathcal{S}$

```
1 n ← 0;
2 repeat
3 |   n ← n + 1;
4 |   Construct and solve Φ_n^S;
5 until Φ_n^S is satisfiable, say with model v;
6 Construct and return φ_v;
```



(a) Syntax tree. (b) Syntax DAG. (c) Identifiers.

Figure 4.1: The syntax tree (left), the syntax DAG (middle), and the identifiers of the syntax DAG (right) of the LTL formula $(p \,\mathsf{U}\, \mathsf{G}\, q) \vee (\mathsf{F}\, \mathsf{G}\, q)$.

Syntactic Constraints

The formula Φ_n^{syn} relies on a canonical syntactic representation of LTL formulas, which we call syntax DAG. A *syntax DAGs* is essentially a syntax tree (i.e., the unique tree that is derived from the inductive definition of an LTL formula) in which common sub-formulas are merged. This merging results in a directed, acyclic graph (DAG), whose number of nodes coincides with the number of sub-formulas of the prospective LTL formula. Figures 4.1a and 4.1b illustrate syntax trees and syntax DAGs, respectively.

To simplify our encoding, we assign a unique identifier $k \in \{1, \ldots, n\}$ to each node of a syntax DAG such that (a) the identifier of the root is n and (b) the identifier of an inner node is larger than the identifiers of its children (see Figure 4.1c). This encoding entails that Node 1 is always a leaf, necessarily labeled with an atomic proposition.

We encode a syntax DAG using three types of propositional variables:

- $x_{i,\lambda}$ where $i \in \{1, \ldots, n\}$ and $\lambda \in \Lambda_{ltl} \cup \mathcal{P}$;
- $l_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$; and
- $r_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$.

Intuitively, the variables $x_{i,\lambda}$ encode a labeling of the syntax DAG in the sense that if a variable $x_{i,\lambda}$ is set to *true*, then Node i is labeled with λ (recall that each node is labeled with either an operator from Λ or an atomic proposition from $\mathcal{P}$). The variables $l_{i,j}$ and $r_{i,j}$, on the other hand, encode the structure of the syntax DAG (i.e., the left and right child of inner nodes): if variable $l_{i,j}$ ($r_{i,j}$) is set to *true*, then j is the identifier of the left (right) child of Node i. By convention, we ignore the variables $r_{i,j}$ if Node i is labeled with a unary operator; similarly, we ignore both $l_{i,j}$ and $r_{i,j}$ if Node i is labeled with an atomic proposition. Note that in the case of $l_{i,j}$ and $r_{i,j}$, the identifier i ranges from 2 to n because Node 1 is always labeled with an atomic proposition and, hence, cannot have children. Moreover, j ranges from 1 to $i-1$, reflecting that identifiers of children have to be smaller than the identifier of the current node.

To enforce that the variables $x_{i,\lambda}$, $l_{i,j}$, and $r_{i,j}$ in fact encode a syntax DAG, we impose the following four constraints:

$$\left[\bigwedge_{1\leq i\leq n}\bigvee_{\lambda\in\Lambda_{ltl}\cup\mathcal{P}} x_{i,\lambda}\right] \wedge \left[\bigwedge_{1\leq i\leq n}\bigwedge_{\lambda\neq\lambda'\in\Lambda_{ltl}\cup\mathcal{P}} \neg x_{i,\lambda}\vee\neg x_{i,\lambda'}\right] \tag{4.1}$$

$$\left[\bigwedge_{2\leq i\leq n}\bigvee_{1\leq j<i} l_{i,j}\right] \wedge \left[\bigwedge_{2\leq i\leq n}\bigwedge_{1\leq j<j'<i} \neg l_{i,j}\vee\neg l_{i,j'}\right] \tag{4.2}$$

$$\left[\bigwedge_{2\leq i\leq n}\bigvee_{1\leq j<i} r_{i,j}\right] \wedge \left[\bigwedge_{2\leq i\leq n}\bigwedge_{1\leq j<j'<i} \neg r_{i,j}\vee\neg r_{i,j'}\right] \tag{4.3}$$

$$\bigvee_{p\in\mathcal{P}} x_{1,p} \tag{4.4}$$

Constraint (4.1) ensures that each node is labeled with exactly one label. Similarly, Constraints (4.2) and (4.3) enforce that each node (except for Node 1) has exactly one left and exactly one right child. Furthermore, Constraint (4.4) makes sure that Node 1 is labeled with an atomic proposition.

Let Φ_n^{syn} now be the conjunction of Constraints (4.1) to (4.4). Then, one can construct a syntax DAG from a model v of Φ_n^{syn} in a straightforward manner:[11] label Node i with the unique label $\lambda \in \Lambda_{ltl} \cup \mathcal{P}$ such that $v(x_{i,\lambda}) = \mathit{true}$, designate Node n as the root, and arrange the nodes of the DAG as uniquely described by $v(l_{i,j})$ and $v(r_{i,j})$. Moreover, we can easily derive an LTL formula from this syntax DAG, which we denote by φ_v. Note, however, that φ_v is not yet related to the sample $\mathcal{S}$ and, thus, might or might not be consistent with it. To enforce that φ_v is indeed consistent with $\mathcal{S}$, we impose additional constraints (i.e., the formula Φ_n^{sem}), which we describe next.

[11] Remember that a model is a function that maps the free variables in a formula to the Boolean values *true* and *false*.

Semantic Constraints

Towards the definition of the formula Φ_n^{sem}, we construct for each ultimately periodic word $uv^\omega \in S_+ \cup S_-$ a propositional formula $\Phi_n^{u,v}$ that tracks the satisfaction of the LTL formula encoded by Φ_n^{syn} (and all its subformulas) on uv^ω. To this end, we use the following observation, which reduces the problem of deciding satisfaction of an LTL formula on an ultimately-periodic word to a problem on a finite prefix.

Observation 4.1 (cf. Neider and Gavran [7, Observation 1]). Let $uv^\omega \in (2^{\mathcal{P}})^\omega$, φ be an LTL formula over $\mathcal{P}$, and $k \in \mathbb{N}$. Then, $uv^\omega[|u|+k, \infty) = uv^\omega[|u|+m, \infty)$ holds for all $m \in \mathbb{N}$ with $m \equiv k \mod |v|$. In addition, $uv^\omega[|u|+k, \infty) \models \varphi$ if and only if $uv^\omega[|u|+m, \infty) \models \varphi$ holds for every LTL formula φ.

Intuitively, Observation 4.1 states that the suffixes of a word uv^ω eventually repeat periodically. Consequently, the valuation of an LTL formula on a word uv^ω can be determined based only on the finite prefix uv (recall that the semantics of temporal operators only depend on the suffixes of a word). To illustrate this claim, consider the LTL formula $\mathsf{X}\,\varphi$ and assume that we want to determine the satisfaction of $\mathsf{X}\,\varphi$ on $uv^\omega[|uv|-1, \infty)$ (i.e., $\mathsf{X}\,\varphi$ is evaluated at the end of the prefix uv). Then, Observation 4.1 permits us to compute the satisfaction based on the suffix $uv^\omega[|u|, \infty)$, as opposed to the original semantics of the X-operator, which recurs to $uv^\omega[|uv|, \infty)$ (i.e., the suffix starting at the next position). Similar, though more involved arguments can be used for all other temporal operators as well.

Each formula $\Phi_n^{u,v}$ is now built over an auxiliary set of propositional variables

- $y_{i,t}^{u,v}$ where $i \in \{1, \ldots, n\}$ is a node in the syntax DAG and $t \in \{0, \ldots, |uv|-1\}$ is a position in the finite prefix uv.

The meaning of these variables is that $y_{i,t}^{u,v}$ should be set to *true* if and only if the LTL subformula rooted at Node i satisfies the suffix $uv^\omega[t, \infty)$. Note that the set of variables for two distinct words from the sample must be disjoint.

To obtain this desired meaning of the variables $y_{i,t}^{u,v}$, we impose the following constraints:

$$\bigwedge_{1 \leq i \leq n} \bigwedge_{p \in \mathcal{P}} x_{i,p} \rightarrow \left[\bigwedge_{0 \leq t < |uv|} \begin{cases} y_{i,t}^{u,v} & \text{if } p \in uv[t] \\ \neg y_{i,t}^{u,v} & \text{if } p \notin uv[t] \end{cases} \right] \tag{4.5}$$

$$\bigwedge_{\substack{1 < i \leq n \\ 1 \leq j < i}} (x_{i,\neg} \wedge l_{i,j}) \rightarrow \left[\bigwedge_{0 \leq t < |uv|} \left[y_{i,t}^{u,v} \leftrightarrow \neg y_{j,t}^{u,v} \right] \right] \tag{4.6}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\vee} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \left[\bigwedge_{0\leq t<|uv|} \left[y_{i,t}^{u,v} \leftrightarrow (y_{j,t}^{u,v} \vee y_{j',t}^{u,v})\right]\right] \tag{4.7}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} (x_{i,\mathsf{X}} \wedge l_{i,j}) \rightarrow \left[\left[\bigwedge_{0\leq t<|uv|-1} y_{i,t}^{u,v} \leftrightarrow y_{j,t+1}^{u,v}\right] \wedge \left[y_{i,|uv|-1}^{u,v} \leftrightarrow y_{j,|u|}^{u,v}\right]\right] \tag{4.8}$$

$$\begin{aligned}\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\mathsf{U}} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow & \left[\left[\bigwedge_{0\leq t<|u|} y_{i,t}^{u,v} \leftrightarrow \bigvee_{t\leq t'<|uv|} \left[y_{j',t'}^{u,v} \wedge \bigwedge_{t\leq t''<t'} y_{j,t''}^{u,v}\right]\right] \wedge \right.\\ & \left.\left[\bigwedge_{|u|\leq t<|uv|} y_{i,t}^{u,v} \leftrightarrow \bigvee_{|u|\leq t'<|uv|} \left[y_{j',t'}^{u,v} \wedge \bigwedge_{t''\in t\looparrowright_{u,v} t'} y_{j,t''}^{u,v}\right]\right]\right]\end{aligned} \tag{4.9}$$

Constraint (4.5) implements the LTL semantics of atomic propositions and ensures that if Node i is labeled with $p \in \mathcal{P}$, then $y_{i,t}^{u,v}$ is set to *true* if and only if $p \in uv[t]$. Next, Constraints (4.6) and (4.7) implement the semantics of negation and disjunction, respectively: if Node i is labeled with $\neg$ and Node j is its left child, then $y_{i,t}^{u,v}$ is the negation of $y_{j,t}^{u,v}$; on the other hand, if Node i is labeled with $\vee$, Node j is its left child, and Node j' is its right child, then $y_{i,t}^{u,v}$ is the disjunction of $y_{j,t}^{u,v}$ and $y_{j',t}^{u,v}$. Moreover, Constraint (4.8) implements the semantics of the X-operator, following the idea of "returning to the beginning of the periodic part v" as sketched above. Finally, Constraint (4.9) implements the semantics of the U-operator: the first conjunction in the consequent covers the positions $t \in \{0, \ldots, |u| - 1\}$ in the initial part u, while the second conjunct covers the positions $t \in \{|u|, \ldots, |uv| - 1\}$ in the periodic part v. The second conjunct relies on an auxiliary set $t \looparrowright_{u,v} t'$ defined by

$$t \looparrowright_{u,v} t' := \begin{cases} \{t, \ldots, t' - 1\} & \text{if } t < t'; \\ \{|u|, \ldots, t' - 1, t, \ldots, |uv| - 1\} & \text{if } t \geq t', \end{cases}$$

which contains all positions in v "between t and t'".

Note that we have omitted the description of the missing operators $\wedge$, $\rightarrow$, F, G, and the constants *true* and *false*, which can be implemented analogously. Moreover, note that our SAT encoding is extensible, and additional LTL operators, such as weak until, weak release, and strong release, can easily be added.

For each $uv^\omega \in S_+ \cup S_-$, let $\Phi_n^{u,v}$ now be the conjunction of Constraints (4.5) to (4.9). Then, we define

$$\Phi_n^{sem} := \left[\bigwedge_{uv^\omega \in S_+} \Phi_n^{u,v} \wedge y_{n,0}^{u,v}\right] \wedge \left[\bigwedge_{uv^\omega \in S_-} \Phi_n^{u,v} \wedge \neg y_{n,0}^{u,v}\right].$$

Note that the subformula $\Phi_n^{u,v} \wedge y_{n,0}^{u,v}$ makes sure that $uv^\omega \in S_+$ satisfies the prospective LTL formula (more concretely, uv^ω starting from position 0 satisfies the LTL formula at the root of the syntax DAG), while $\Phi_n^{u,v} \wedge \neg y_{n,0}^{u,v}$ ensures that $uv^\omega \in S_-$ does not satisfy it.

This concludes the description of our encoding, and we are left with arguing the correctness and termination of Algorithm 3. We do this next.

Termination and Correctness

It is not hard to verify that Algorithm 3 indeed produces a minimal consistent LTL formula by virtue of the two defining properties of $\Phi_n^{\mathcal{S}}$ (cf. Neider and Gavran [7, Lemma 1]) and the way we increase n. Moreover, termination of Algorithm 3 follows from the existence of trivial solution $\varphi_{\mathcal{S}}^\star$ (see Page 68), and the size of $\varphi_{\mathcal{S}}^\star$ provides an upper bound on the value of n. In total, we obtain the following result.

Theorem 4.1 (cf. Neider and Gavran [7, Theorem 1])**.** *Given a sample $\mathcal{S}$, Algorithm 3 terminates and outputs a minimal LTL formula that is consistent with $\mathcal{S}$.*

Although SAT solving is a computationally expensive task, our approach performs well on medium-sized benchmarks [7]. However, to improve the performance, we have also developed an extension that incorporates decision tree learning. The key idea is to separate the learning process into two phases. In the first phase, we run Algorithm 3 on various subsets of the examples. The result is a (small) number of LTL formulas, which we named *LTL primitives*, that classify at least these subsets correctly. In the second phase, we then use a standard learning algorithm for decision trees to learn a Boolean combination of these LTL primitives that correctly classifies the whole set of examples, though it might not be minimal. We refer the reader to Neider and Gavran [7] for further details.

4.1.2 Learning PSL Specifications from Positive and Negative Examples

Although LTL is a very popular specification language, one of its major downsides is the limited expressive power compared to other temporal logics. Consequently, many properties that arise naturally (e.g., an event happening at every n-th point in time) cannot be expressed in LTL. In fact, the class of properties that can be expressed in LTL corresponds precisely to that of star-free ω-languages [151], which excludes—among others—all properties involving modulo counting.

To overcome this limitation, the *Property Specification Language (PSL)* has been proposed [61]. Although PSL is an extension of LTL and, hence, shares many of its advantageous properties, PSL differs from LTL in three important aspects:

1. the expressive power of PSL exceeds that of LTL (it is as expressive as the full class of ω-regular languages [23]);
2. PSL integrates easy-to-understand regular expressions in its syntax; and
3. specifications expressed in PSL can be arbitrarily more succinct than those expressed in LTL (see Roy, Fisman, and Neider [14, Proposition 1]).

Let us begin our definition of PSL by introducing one of its main constituents, namely regular expressions. To better align our notation with that of LTL, we use propositional formulas rather than symbols of an alphabet as atomic regular expressions. Moreover, we append the subscript "r" to Boolean operators in atomic expressions to not confuse them with LTL operators. This convention allows us to define the syntax of regular expressions using the following two simple grammar rules, where the first rule describes the construction of atomic expressions and the second rules describes the construction of general regular expressions:

$$\xi ::= p \in \mathcal{P} \mid \neg_r \xi \mid \xi \vee_r \xi$$
$$\rho ::= \varepsilon \mid \xi \mid \rho + \rho \mid \rho \circ \rho \mid \rho^*$$

As usual, the regular operator $+$ stands for choice, $\circ$ stands for concatenation, and * stands for finite repetition (Kleene star). We also allow the standard Boolean operators (e.g., conjunction, implication, and so on) as syntactic sugar in atomic expressions and use $\Lambda_{atm} = \{\neg_r, \vee_r\}$ to denote the set of operators that can be used to construct atomic expressions. Moreover, we denote the set of regular expression operators by $\Lambda_{re} \coloneqq \{+, \circ, ^*\}$.

To give meaning to regular expressions, we first associate with every atomic expression ξ a set $[\![\xi]\!] \subseteq 2^{\mathcal{P}}$ of symbols in the following way: $[\![p]\!] = \{A \in 2^{\mathcal{P}} \mid p \in A\}$, $[\![\neg_r \xi]\!] = 2^{\mathcal{P}} \setminus [\![\xi]\!]$, and $[\![\xi_1 \vee_r \xi_2]\!] = [\![\xi_1]\!] \cup [\![\xi_2]\!]$. Based on this notion, we define the semantics of regular expressions by means of a matching relation $\vdash$, which formalizes when a finite infix $\alpha[t, t')$ of an infinite word $\alpha \in (2^{\mathcal{P}})^\omega$ *matches* a regular expression:

- $u[t, t') \vdash \varepsilon$ if and only if $t = t'$;
- $u[t, t') \vdash \xi$ if and only if $t' = t + 1$ and $u[t] \in [\![\xi]\!]$;
- $u[t, t') \vdash \rho_1 + \rho_2$ if and only if $u[t, t') \vdash \rho_1$ or $u[t, t') \vdash \rho_2$;
- $u[t, t') \vdash \rho_1 \circ \rho_2$ if and only if there exists a $t'' \in \{t, \ldots, t'\}$ such that $u[t, t'') \vdash \rho_1$ and $u[t'', t') \vdash \rho_2$; and

- $u[t, t') \vdash \rho^*$ if and only if $t = t'$ or there exists a $t'' \in \{t+1, \ldots, t'\}$ such that $u[t, t'') \vdash \rho$ and $u[t'', t') \vdash \rho^*$.

In this section, we consider the so-called *core fragment* of the Property Specification Language [61], which we here abbreviate as *PSL* for the sake of simplicity. This fragment extends LTL with a single new operator

$$\rho \mapsto \psi,$$

where ρ is a regular expression and ψ is a PSL formula. The intuitive meaning of this operator, which is called *trigger operator*, is that a word $\alpha \in (2^{\mathcal{P}})^\omega$ satisfies the PSL formula $\rho \mapsto \psi$ if and only if ψ holds every time the regular expression ρ matches on a finite prefix of α. To define the semantics of the trigger operator formally, we extend the satisfaction relation $\models$ of LTL by

- $\alpha \models \rho \mapsto \psi$ if and only if $\alpha[0, t) \vdash \rho$ implies $\alpha[t-1, \infty) \models \psi$ for all $t \in \mathbb{N} \setminus \{0\}$.

Moreover, we denote the set of all PSL operators by $\Lambda_{psl} \coloneqq \Lambda_{ltl} \cup \{\mapsto\}$ and define the *size* $|\psi|$ of a PSL formula ψ to be the number of its unique subformulas and subexpressions. Note that PSL subsumes LTL, meaning that every LTL formula is also a PSL formula.

A prototypical example of a PSL formula is $(\mathit{true} \circ \mathit{true})^+ \mapsto p$, requiring that event p happens at every even point in time (the regular expression $(\mathit{true} \circ \mathit{true})^+$ matches every non-empty prefix of even length). Note that this property cannot be expressed in LTL as it requires modulo counting.

Analogous to LTL, we say that a PSL formula ψ is *consistent* with a sample $\mathcal{S} = (S_+, S_-)$ if $uv^\omega \models \psi$ for each $uv^\omega \in S_+$ and $uv^\omega \not\models \psi$ for each $uv^\omega \in S_-$. Then, the learning task for PSL is as follows:

> *"given a sample $\mathcal{S}$, compute a PSL formula of minimal size that is consistent with $\mathcal{S}$".*

Note that this task asks again to find a minimal PSL formula that is consistent with the sample.

Our learning algorithm for PSL formulas is shown as pseudo code in Algorithm 4. It follows the same idea as Algorithm 3 (on Page 69) and reduces the construction of a minimally consistent PSL formula to a series of constraint satisfaction problems in propositional logic. More precisely, we construct for a sample $\mathcal{S}$ and a natural number $n \in \mathbb{N} \setminus \{0\}$ a propositional formula $\Psi_n^{\mathcal{S}}$ that has the following two properties:

1. $\Psi_n^{\mathcal{S}}$ is satisfiable if and only if there exists a PSL formula of size n that is consistent with $\mathcal{S}$; and

Algorithm 4: SAT-based learning algorithm for PSL specifications [14].

Input: a sample $\mathcal{S}$

```
n ← 0;
repeat
    n ← n + 1;
    Construct and solve Ψ_n^S;
until Ψ_n^S is satisfiable, say with model v;
Construct and return ψ_v;
```

2. if v is a model of $\Psi_n^{\mathcal{S}}$, then v contains sufficient information to construct a PSL formula ψ_v of size n that is consistent with $\mathcal{S}$.

By starting with $n = 1$ and incrementing n until $\Psi_n^{\mathcal{S}}$ becomes satisfiable, we obtain an effective learning algorithm for specifications expressed in PSL.

As before, the formula $\Psi_{\mathcal{S}}^n$ is a conjunction

$$\Psi_{\mathcal{S}}^n := \Psi_n^{syn} \wedge \Psi_n^{sem},$$

where Ψ_n^{syn} encodes syntactic constraints of the prospective PSL formula and Ψ_n^{sem} enforces semantic constraints (i.e., that the prospective PSL formula is consistent with the sample). In the remainder of this section, we describe both formulas in detail and argue the termination and correctness of our learning algorithm.

Syntactic Constraints

The syntactic constraints closely resemble those of LTL, except that they also need to account for regular expressions and the trigger operator. As before, we use three types of variables to encode a syntax DAG:

- $x_{i,\lambda}$ where $i \in \{1, \ldots, n\}$ and $\lambda \in \Lambda_{atm} \cup \Lambda_{re} \cup \Lambda_{psl} \cup \mathcal{P} \cup \{\varepsilon\}$;
- $l_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$; and
- $r_{i,j}$ where $i \in \{2, \ldots, n\}$ and $j \in \{1, \ldots, i-1\}$.

Again, the variables $x_{i,\lambda}$ encode the labels of nodes in the syntax DAG, while the variables $l_{i,j}$ and $r_{i,j}$ encode their left and right children, respectively.

To ensure that these variables indeed encode a well-formed syntax DAG, we can reuse Constraints (4.1) to (4.4) (on Page 70), except that we let λ in Constraint (4.1) now range over $\Lambda_{atm} \cup \Lambda_{re} \cup \Lambda_{psl} \cup \mathcal{P} \cup \{\varepsilon\}$. However, we also need to enforce that the

labeling of the syntax DAG respects the type of the operators (e.g., children of a regular expression are always regular expressions themselves). In the case of the trigger operator, for instance, we can enforce this using the constraint

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} [x_{i,\mapsto} \wedge l_{i,j} \wedge r_{i,j'}] \rightarrow \left[\left[\bigvee_{\lambda\in\Lambda_{atm}\cup\Lambda_{re}\cup\mathcal{P}\cup\{\varepsilon\}} x_{j,\lambda}\right] \wedge \left[\bigvee_{\lambda\in\Lambda_{psl}\cup\mathcal{P}} x_{j',\lambda}\right]\right],$$

which states that the left operand of the operators is a regular expression and the right is a PSL formula. Since the constraints for all other operators follow a similar pattern, we omit them here for the sake of conciseness.

The formula Ψ_n^{syn} is now simply the conjunction of Constraints (4.1) to (4.4), with the modifications sketched above, as well as the constraints enforcing a correct nesting of operators. Note that every PSL formula needs to contain at least one atomic proposition from $\mathcal{P}$, which justifies the inclusion of Constraint (4.4).

Given a model v of Ψ_n^{syn}, we extract a PSL formula ψ_v as before: we first construct a syntax DAG by labeling Node i with the unique label λ such that $v(x_{i,\lambda}) = \mathit{true}$, designating Node n as the root, and arranging the nodes of the DAG as uniquely described by $v(l_{i,j})$ and $v(r_{i,j})$; then, we construct the PSL formula ψ_v that is uniquely determined by this syntax DAG. To enforce that ψ_v is indeed consistent with $\mathcal{S}$, we impose additional constraints (i.e., the formula Ψ_n^{sem}), which we describe next.

Semantic Constraints

Analogously to LTL, we design the formula Ψ_n^{sem} to capture the semantics of PSL in propositional logic. However, we also have to account for regular expressions this time, which complicate matters.

Let us begin by recalling Observation 4.1 (on Page 71): since the infinite suffixes of an ultimately-periodic word $\alpha \in (2^{\mathcal{P}})^\omega$ repeat, we know that for every LTL formula φ and every $k \in \mathbb{N}$, the suffix $\alpha[|u|+k, \infty)$ satisfies φ if and only if the suffix $\alpha[|u|+m, \infty)$ for all $m \in \mathbb{N}$ with $m \equiv k \mod |v|$ satisfies φ. In other words, the prefix uv of an ultimately-periodic word uv^ω already carries sufficient information to determine the satisfaction of an LTL formula on every suffix of the word. This insight allowed us to restrict our SAT-encoding to the prefix uv of each ultimately-periodic word uv^ω in the sample.

It is not hard to verify that Observation 4.1 remains true even in the case of PSL formulas. In contrast to LTL, however, actually computing which suffix satisfies a given PSL formula requires us to work with prefixes longer than uv. To illustrate this

claim, consider the ultimately-periodic word $uv^\omega = \emptyset\{p\}(\emptyset)^\omega$ and the PSL formula $\psi \coloneqq (\mathit{true} \circ \mathit{true})^+ \mapsto p$ (stating that p is true at every second position). By just considering the prefix $uv = \emptyset\{p\}\emptyset$, it seems that $uv^\omega \models \psi$. However, "unrolling" the repeating part $v = \emptyset$ once more, resulting in the prefix $uvv = \emptyset\{p\}\emptyset\emptyset$, immediately shows that $uv^\omega \not\models \psi$.

As the example above shows, gathering enough information about matchings of a regular expression inside a trigger operator requires unrolling the repeating part v of an ultimately-periodic word uv^ω multiple times. Similar to Observation 4.1, the lemma below provides an upper bound $b \in \mathbb{N}$ on the number of unrolling that is sufficient to determine the satisfaction of a trigger operator. This bound depends on the number n of nodes of the syntax DAG and a function $M_{u,v} \colon \mathbb{N} \to \mathbb{N}$ that maps a position t in the word uv^ω to an "appropriate" position within the prefix uv. Formally, we define $M_{u,v}$ by

$$M_{u,v}(t) \coloneqq \begin{cases} t & \text{if } t < |uv|; \text{ and} \\ |u| + ((t - |u|) \,\%\, |v|) & \text{if } t \geq |uv|, \end{cases}$$

where $a \,\%\, b$ is the remainder of the division $\frac{a}{b}$. The proof of our lemma uses a translation of the regular expression into a DFA, which can incur an exponential blow-up and causes the bound b to be exponential in n (the latter being an upper bound for the size of any regular expression that can appear in the prospective PSL formula).

Lemma 4.1 (cf. Roy, Fisman, and Neider [14, Lemma 1])**.** *Let $uv^\omega \in (2^{\mathcal{P}})^\omega$ be an ultimately-periodic word and $\psi = \rho \mapsto \psi'$ a PSL formula with $|\psi| = n$. Additionally, let $b = 2^n + 1$ and $t \in \{0, \ldots, |uv| - 1\}$. Then, $uv^\omega[t, \infty) \models \psi$ if and only if $uv^\omega[t, t') \vdash \rho$ implies $uv^\omega[M_{u,v}(t' - 1), \infty) \models \psi'$ for each $t' \in \{0, \ldots, |uv^b| - 1\}$.*

Note an important corollary of Observation 4.1 and Lemma 4.1: determining matchings of regular expressions requires us to consider the prefix uv^b, but the prefix uv is sufficient for determining the satisfaction of PSL operators. We exploit this fact in our construction below.

Equipped with appropriate bounds, we can now construct for each ultimately periodic word uv^ω in $S_+ \cup S_-$ a propositional formula $\Psi_n^{u,v}$ that tracks the satisfaction of the PSL formula encoded by Ψ_n^{syn} (and all its sub-formulas and sub-expressions) on uv^ω. Each of these formulas is built over two types of auxiliary variables, where we set $b = 2^n + 1$ as in Lemma 4.1:

- $y_{i,t}^{u,v}$ where $i \in \{1, \ldots, n\}$ is a node in the syntax DAG and $t \in \{0, \ldots, |uv| - 1\}$ is a position in the finite prefix uv; and

- $z^{u,v}_{i,t,t'}$ where $i \in \{1,\ldots,n\}$ is a node in the syntax DAG and $t,t' \in \{0,\ldots,|uv^b|-1\}$ are positions in the finite prefix uv^b with $t \leq t'$.

The meaning of these variables is that $y^{u,v}_{i,t}$ is set to *true* if and only if $uv^\omega[t,\infty)$ satisfies the PSL formula rooted at Node i (if that node is labeled with a PSL operator); similarly, $z^{u,v}_{i,t,t'}$ is set to *true* if and only if $uv^\omega[t,t')$ matches the regular expression rooted at Node i (if that node is labeled with a regular expression operator). Note that we have to create both the variables $y^{u,v}_{i,t}$ and $z^{u,v}_{i,t,t'}$ for each node since the "type" of a node is determined dynamically during SAT solving (by the label λ assigned to that node).

To ensure our auxiliary variables have the desired meaning, we proceed in two steps. First, we create constraints to ensure that the variables $z^{u,v}_{i,t,t'}$ indeed capture the matching of regular expressions. The exact constraints are shown below, where we use a slight modification of the Iverson bracket $[i=j]$ to obtain the truth value of the comparison $i=j$:

$$\bigwedge_{1\leq i\leq n} x_{i,\varepsilon} \rightarrow \left[\bigwedge_{0\leq t\leq t'<|uv^b|} z^{u,v}_{i,t,t'} \leftrightarrow [t=t']\right] \tag{4.10}$$

$$\bigwedge_{1\leq i\leq n}\bigwedge_{p\in\mathcal{P}} x_{i,p} \rightarrow \left[\bigwedge_{0\leq t\leq t'<|uv^b|} \begin{cases} z^{u,v}_{i,t,t'} & \text{if } t'=t+1 \text{ and } p\in uv^b[t] \\ \neg z^{u,v}_{i,t,t'} & \text{otherwise} \end{cases}\right] \tag{4.11}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} (x_{i,\neg_r} \wedge l_{i,j}) \rightarrow \left[\bigwedge_{0\leq t\leq t'<|uv^b|} \left[z^{u,v}_{i,t,t'} \leftrightarrow \neg z^{u,v}_{j,t,t'}\right]\right] \tag{4.12}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\vee_r} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \left[\bigwedge_{0\leq t\leq t'<|uv^b|} \left[z^{u,v}_{i,t,t'} \leftrightarrow z^{u,v}_{j,t,t'} \vee z^{u,v}_{j',t,t'}\right]\right] \tag{4.13}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,+} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \left[\bigwedge_{0\leq t\leq t'<|uv^b|} \left[z^{u,v}_{i,t,t'} \leftrightarrow z^{u,v}_{j,t,t'} \vee z^{u,v}_{j',t,t'}\right]\right] \tag{4.14}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j,j'<i}} (x_{i,\circ} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \left[\bigwedge_{0\leq t\leq t'<|uv^b|} \left[z^{u,v}_{i,t,t'} \leftrightarrow \bigvee_{t\leq t''\leq t'} z^{u,v}_{j,t,t''} \wedge z^{u,v}_{j',t'',t'}\right]\right] \tag{4.15}$$

$$\bigwedge_{\substack{1<i\leq n\\ 1\leq j<i}} (x_{i,*} \wedge l_{i,j}) \rightarrow \left[\bigwedge_{0\leq t\leq t'<|uv^b|} \left[z^{u,v}_{i,t,t'} \leftrightarrow \left[[t=t'] \vee \bigvee_{t<t''\leq t'} z^{u,v}_{j,t,t''} \wedge z^{u,v}_{i,t'',t'}\right]\right]\right] \tag{4.16}$$

Note that Constraints (4.10) to (4.16) closely follow the definition of the matching relation for regular expressions: Constraints (4.10) to (4.13) capture the semantics of atomic regular expressions, while Constraints (4.14) to (4.16) capture the operators $+$, $\circ$, and *. Furthermore, note that Constraints (4.13) and (4.14) coincide.

Next, we need to enforce the desired meaning for the variables $y_{i,t}^{u,v}$. To this end, we can reuse Constraints (4.5) to (4.9) for the operators $\neg, \vee, \mathsf{X}$, and U (see Page 72). In addition, we introduce the following constraint for the trigger operator:

$$\bigwedge_{\substack{1<i\leq n \\ 1\leq j,j'<i}} (x_{i,\mapsto} \wedge l_{i,j} \wedge r_{i,j'}) \rightarrow \left[\bigwedge_{0\leq t<|uv|} \left[y_{i,t}^{u,v} \leftrightarrow \bigwedge_{t<t'<|uv^b|} \left[z_{j,t,t'}^{u,v} \rightarrow y_{j',M_{u,v}(t'-1)}^{u,v} \right] \right] \right] \tag{4.17}$$

Note that this constraint connects variables of the form $y_{i,t}^{u,v}$ and $z_{i,t,t'}^{u,v}$.

For each $uv^\omega \in S_+ \cup S_-$, let $\Psi_n^{u,v}$ now be the conjunction of Constraints (4.5) to (4.17). Then, we define

$$\Psi_n^{sem} := \left[\bigwedge_{uv^\omega \in S_+} \Psi_n^{u,v} \wedge y_{n,0}^{u,v} \right] \wedge \left[\bigwedge_{uv^\omega \in S_-} \Psi_n^{u,v} \wedge \neg y_{n,0}^{u,v} \right].$$

which enforces that all positive words in $\mathcal{S}$ satisfy the prospective PSL formula ($y_{n,0}^{u,v}$ has to be *true*), while all negative words violate it ($y_{n,0}^{u,v}$ has to be *false*).

This concludes the description of our encoding, and we are left with arguing the correctness and termination of Algorithm 4. We do this next.

Termination and Correctness

As in the case of our learning algorithm for LTL, it is not hard to verify that Algorithm 4 indeed produces a minimal consistent PSL formula by virtue of the two defining properties of $\Psi_n^{\mathcal{S}}$ and the way we increase n. Moreover, the size of the LTL formula $\varphi_{\mathcal{S}}^\star$ (see Page 68) provides an upper bound on the value of n since every LTL formula is also a PSL formula. Thus, Algorithm 4 is guaranteed to terminate, and we obtain the following result.

Theorem 4.2 (cf. Roy, Fisman, and Neider [14, Theorem 1])**.** *Given a sample $\mathcal{S}$, Algorithm 4 terminates and outputs a minimal PSL formula that is consistent with $\mathcal{S}$.*

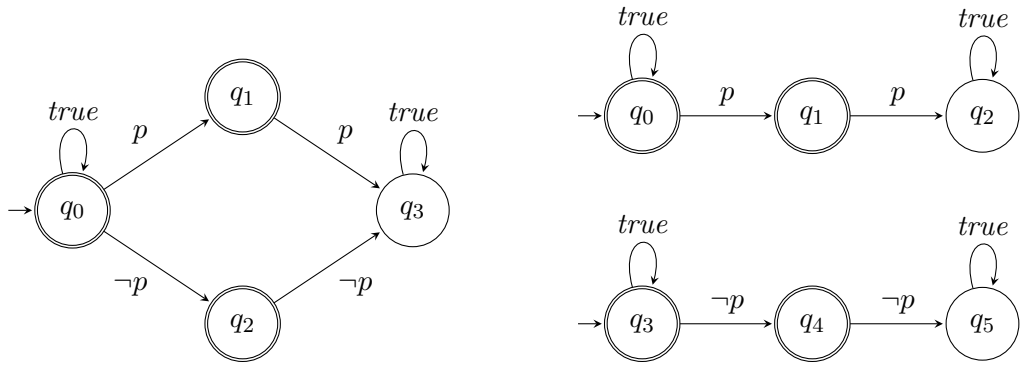
Finally, let us mention that one can easily restrict Algorithm 4 to learn regular expressions over finite words. Moreover, it is possible to modify Algorithm 4 to learn specifications in the form of ω-regular expressions. The latter requires a result similar to Lemma 4.1 in order to bound the length of prefixes that the algorithm needs to consider.

4.2 Learning Specifications from Positive Examples Only

The methods presented in Section 4.1 rely on the assumption that both positive and negative examples are available to learn a specification. However, there exist many situations where this is not the case. For instance, when inferring the specification of a system whose implementation is given but too big to be fully analyzed, one can extract input-output traces that represent possible executions of the system. Proving that a particular input-output trace is not a possible execution of the system, on the other hand, is a model checking problem, which can be infeasible to solve for complex designs. Similarly, one may want to deduce an environment specification to be used in reactive synthesis from observing the environment. For a black-box environment, however, we can never know that some behavior cannot occur. These observations give rise to the question of whether there is some way to learn from positive (or negative) examples only.

Learning from only one type of example is, in general, an ill-posed problem. In the case of positive examples, for instance, there is a spectrum of possible specification solutions ranging from *true* (i.e., the specification that allows any behavior) all the way to the specification that only allows precisely the set of positive examples. Both extremes make little sense, and a learning procedure should provide a mechanism to rule out these trivial specifications. In fact, a learning procedure should ideally be parameterized by a *tightness value*, which allows controlling where on the spectrum between the two extremes the learned specification lies.

In this section, we propose universal very-weak automata over infinite words as a representation for specifications, which has a natural definition of tightness, lends itself to an effective learning procedure, and leads to easily readable specifications. Formally, a *universal very-weak automaton (UVW)* is a universal co-Büchi automaton with the particular structural property that all loops are self-loops. On the one hand, being a co-Büchi automaton means that a UVW $\mathcal{A}$ accepts an infinite word $\alpha \in (2^{\mathcal{P}})^{\omega}$ if and only if every run of $\mathcal{A}$ on α visits non-final states only finitely often. On the other hand, the structural constraint implies that every run moves away from the initial state and eventually loops in a single state (note that the former can only happen finitely often).



(a) A UVW in non-decomposed form.

(b) A decomposition of the UVW in Figure 4.2a into two simple chains of length $n = 3$.

Figure 4.2: Two universal very-weak automata accepting the language defined by the LTL formula $\mathsf{G}(p \leftrightarrow \mathsf{X}\,\neg p)$. For the sake of readability, we use propositional formulas as labels on transitions. Final states are drawn as double-circles.

To ease our notation, we denote the language of an UVW $\mathcal{A}$ (i.e., the set of all words that $\mathcal{A}$ accepts) by $L(\mathcal{A})$.

Figure 4.2 depicts two examples of UVWs over the set $\mathcal{P} = \{p, q\}$ of atomic propositions. For now, let us consider Figure 4.2a and postpone the discussion of Figure 4.2b to a later point. Note that we here use propositional formulas as labels on transitions. Similar to the regular expressions discussed in Section 4.1.2, each such formula ξ uniquely represents a subset of symbols from the alphabet $2^{\mathcal{P}}$ (i.e., a subset of atomic propositions), which we again denote by $[\![\xi]\!] \subseteq 2^{\mathcal{P}}$. For instance, the set of symbols induced by the formula $p \vee \neg q$ is $[\![p \vee \neg q]\!] = \{\emptyset, \{p\}, \{p, q\}\}$.

A close inspection of the UVW in Figure 4.2a shows that this UVW accepts exactly the set of infinite words in which the atomic proposition p alternates—or, equivalently, that satisfy the LTL formula $\mathsf{G}(p \leftrightarrow \mathsf{X}\,\neg p)$ (e.g., the words $(\{p\}\{q\})^\omega$ and $(\{q\}\{p, q\})^\omega$ are accepted, while $\{p\}^\omega$ and $\{q\}^\omega$ are not). To verify this claim, recall that UVWs are universal co-Büchi automata, meaning that a word is accepted if and only if all runs visit non-final states only finitely often. In our example, this means that a run must never reach state q_3. However, the UVW is constructed such that every word in which the atomic proposition p does not alternate induces at least one run that eventually reaches state q_3 and, hence, is not accepting. Thus, the UVW accepts only those words in which p alternates.

It is worth emphasizing that every UVW can be interpreted as a collection of properties, each of which describing a situation that leads to a rejection of the input (e.g., the

upper path in the UVW of Figure 4.2a is taken when p occurs in two consecutive symbols, while the lower part is taken in case p does not occur in two consecutive symbols). We encourage the reader to assume this perspective for the rest of this section.

An essential feature of UVWs is that they characterize precisely the class of properties that can be expressed in both LTL and the universal fragment of Computation Tree Logic (ACTL) [109]. While this implies that there are some LTL properties that UVWs cannot express, the intersection of LTL and ACTL includes the vast majority of specifications found in case studies on specification types [16]. Moreover, by trading away the full expressivity of LTL, we obtain a mechanism that makes learning from only positive examples feasible: UVWs can be decomposed into so-called simple chains, which represent individual scenarios that describe how a system is required to react in specific circumstances. This decomposition has two major advantages: first, simple chains are easy to examine by a specification engineer and can be straightforwardly translated into LTL (we refer the reader to Ehlers [59] for more details); second, the maximum length of such a chain is a natural notion for the tightness value of a specification since longer chains allow for more complex behavior.

Formally, a *simple chain* [59] is a sequence $q_1, \ldots, q_n$ of distinct states of an UVW such that there exists a transition from q_i to q_{i+1} for each $i \in \{1, \ldots, n-1\}$ (note that this definition permits self-loops). A simple chain is called *longest* (or *maximal*) in an automaton if it cannot be extended by an additional state at the beginning or the end without losing the property that it is contained in the automaton. We say that a UVW is in decomposed form if there are no transitions between the maximal simple chains of the UVW and for every such simple chain $q_1, \ldots, q_n$, there are no "jumping transitions" (i.e., there does not exist a transition from state q_i to a state q_j for $j > i+1$). Without loss of generality, we can assume that every chain in a decomposed UVW has an initial state, and the last state is non-final—otherwise, the whole chain or the last state can be removed, respectively. For instance, Figure 4.2b shows the UVW of Figure 4.2a in decomposed form. For technical reasons, we permit multiple initial states, one per simple chain.

Given a set $S_+ \subset (2^{\mathcal{P}})^\omega$ of positive examples over atomic propositions $\mathcal{P}$, we now want to learn an "informative" UVW $\mathcal{A}$ that accepts all words in S_+ (i.e., $S_+ \subseteq L(\mathcal{A})$). Since we have already argued that this is an ill-posed problem in general, we propose a more specialized learning setting: we seek to learn the strictest automaton (i.e., accepting the smallest language in terms of language inclusion) that (a) includes all positive examples and (b) satisfies a syntactic cut-off criterion.

For UVWs, the length of the longest chain is a natural cut-off criterion, which we use as our tightness value. Formally, we say that a UVW $\mathcal{A}$ is *n-tight* for a finite set $S_+ \subset (2^{\mathcal{P}})^\omega$ and a natural number $n \in \mathbb{N}$ if three conditions hold:

- $\mathcal{A}$ accepts all positive examples (i.e., $S_+ \subseteq L(\mathcal{A})$);
- there does not exist a simple chain in $\mathcal{A}$ that is longer than n; and
- if $\mathcal{A}'$ is a UVW with $S_+ \subseteq L(\mathcal{A}') \subset L(\mathcal{A})$, then at least one simple chain in $\mathcal{A}'$ is longer than n.

To simplify our presentation, we typically drop the reference to the set S_+ if it is clear from the context.

A straightforward yet technical proof by contradiction shows that an n-tight UVW exists for every set S_+ of positive examples and every tightness value $n \in \mathbb{N} \setminus \{0\}$. Moreover, all n-tight UVWs for a set S_+ recognize the same (unique) language, which subsumes the set S_+ and is minimal with respect to language inclusion (cf. Ehlers, Gavran, and Neider [1, Lemma 1]). Thus, n-tightness is a suitable metric for learning from positive examples only, which gives rise to the following learning task:

> *"given a set $S_+ \subset (2^{\mathcal{P}})^\omega$ of positive examples (represented as ultimately-periodic words) and a natural number $n \in \mathbb{N} \setminus \{0\}$, compute an n-tight UVW".*

Note that decreasing and increasing the tightness value n allows us to control whether the specification permits more or less behavior, respectively. This is a consequence of the fact that longer chains provide more fine-grained control over which words to accept or reject.

Having defined the learning setup, let us now present our learning algorithm for n-tight UVWs. Its key idea is to enumerate all simple chains of length n that a decomposed automaton accepting all elements from S_+ can have; as we show later, taking all of these chains together then results in a UVW that is n-tight and, hence, solves our learning task. However, enumerating all simple chains that are consistent with the given positive examples is computationally inefficient as their number grows exponentially with n and the number of atomic propositions. To mitigate this problem, we define a partial order over simple chains that is consistent with language inclusion of UVWs. This approach allows us to consider only those simple chains that are strongest for this partial order.

To avoid cluttering our presentation with too many technical details, we introduce the notion of *strongest simple chains* using the example in Figure 4.3. Assuming that both chains in this figure are compatible with some set of positive examples over the atomic propositions $\mathcal{P} = \{p, q\}$, the one on the right-hand side is stronger than the one on the left-hand side in the sense that it rejects strictly more words. This claim can be shown as follows: first; both chains have the same length; second, at each self-loop and each edge between states, the set of symbols induced by formulas in the simple chain on the left is a subset of the symbols induced by formulas in the simple chain on the right.

Figure 4.3: Two simple chains, the one on the right-hand side is syntactically stronger than the one on the left-hand side.

Algorithm 5: Learning algorithm for n-tight UVWs [1].

Input: a finite set $S_+ \subset (2^{\mathcal{P}})^\omega$ of positive examples, consisting of ultimately-periodic words over a set $\mathcal{P}$ of atomic propositions, and a natural number $n \in \mathbb{N} \setminus \{0\}$

1 Compute the set Υ of all strongest simple chains of length n that accept all words in S_+ using the Pareto front enumeration algorithm by Ehlers [60];

2 Compute a UVW $\mathcal{A}_{S_+}$ recognizing the language $\bigcap_{C \in \Upsilon} \mathcal{L}(A_C)$, where $\mathcal{A}_C$ is the UVW that consists of only the simple chain C;

3 **return** $\mathcal{A}_{S_+}$ (or transform it into an equivalent LTL formula φ_{S_+} and **return** φ_{S_+});

Hence, every rejecting run of the chain on the left is also a rejecting run of the chain on the right.

The main steps of our learning algorithm for n-tight UVWs are now shown in Algorithm 5. We first compute all strongest simple chains of length n that accept all words in S_+ using a Pareto front enumeration algorithm by Ehlers [60].[12] This algorithm relies on a monotone function f_n, which—in our case—maps simple chains to either 0 or 1, depending on whether a simple chain accepts all words in S_+ (see Ehlers, Gavran, and Neider [1, Definition 2 and Lemma 2] for full details). Note that computing the output of the function f_n involves a model checking step to determine whether a simple chain accepts all words in S_+. However, this step can be performed in an efficient, straightforward manner because all words in S_+ are ultimately-periodic.

Once we have computed the set Υ of all strongest simple chains of length n that accept all words in S_+, we construct for every simple chain $C \in \Upsilon$ the corresponding UVW $\mathcal{A}_C$ that consists of only the simple chain C. Then, we compute an UVW $\mathcal{A}_{S_+}$ accepting

[12] Note that enumerating chains of length n is in fact sufficient: a shorter chain can always be extended to a chain of length n by duplicating the last (rejecting) state and rerouting the outgoing transitions of the previously last state to the new last state. This results in a language-equivalent chain of length n that is not missed during enumeration.

the language

$$L(\mathcal{A}_{S_+}) \coloneqq \bigcap_{C \in \Upsilon} \mathcal{L}(A_C),$$

which is possible since universal very-weak automata are closed under language intersection by just merging the state sets, transition relations, and initial states [59]. The resulting UVW $\mathcal{A}_{S_+}$ is indeed n-tight (as it is guaranteed to accept all words in S_+ and minimal in terms of language inclusion), which solves our learning task. If desired, the UVW $\mathcal{A}_{S_+}$ can finally be translated into an equivalent LTL formula φ_{S_+}, as described by Ehlers [59]. In total, we obtain the following result.

Theorem 4.3 (cf. Ehlers, Gavran, and Neider [1, Corollary 1])**.** *Given a finite set $S_+ \subset (2^{\mathcal{P}})^\omega$ of positive examples (represented as ultimately-periodic words) and a natural number $n \in \mathbb{N} \setminus \{0\}$, Algorithm 5 learns an n-tight UVW.*

To conclude, let us mention a simple yet effective way to improve the performance of our algorithm. After the Pareto front of strongest chains has been enumerated, we must merge all strongest simple chains into one UVW. Instead of naively adding all simple chains to the solution UVW, we can improve this process by alternating between adding a new simple chain and minimizing the resulting automaton as described by Adabala and Ehlers [16]. This approach has two advantages. On the one hand, it keeps the size of intermediate automata manageable. On the other hand, it allows us to use Algorithm 5 in an *anytime* fashion: if the process is stopped prematurely (e.g., when a given resource budget is exceeded), the result is still useful—a UVW that accepts a subset of the language that the final automaton (given sufficient computation resources) would accept.

4.3 Notes on Related Work

Learning of formal specifications from examples has recently attracted increasing attention, not only in the area of formal methods but also in (explainable) artificial intelligence. As done in this chapter, one typically distinguishes between learning from both positive and negative examples on the one hand and learning from only one type of example (most often from positive examples) on the other.

In the context of learning from positive and negative examples, the literature can be broadly structured along three dimensions. The first dimension is the type of logic or type of specification language. Examples include learning of specifications expressed in Signal Temporal Logic [100, 112], Linear Temporal Logic [46, 7, 132], the Property Specification Language [14] and even branching time logics, such as Computational

Temporal Logic [150]. To the best of our knowledge, we are the first to devise an algorithm for learning specifications in PSL or an equally expressive logic.

The second dimension is whether or not the learning algorithm requires the user to provide templates. Examples that require templates are the algorithms of Li, Dworkin, and Seshia [106] as well as Lemieux, Park, and Beschastnikh [105]. Note that providing templates is often a challenging task as it requires the user to have a good understanding of the data. By contrast, the algorithms presented in this chapter work without templates and can learn arbitrary formulas without any assistance from the user.

The third dimension distinguishes between algorithms that learn exact specifications and those that learn approximate ones. Like the majority of algorithms mentioned so far, the learning algorithms we have devised in this chapter are exact: all three algorithms learn specifications that describe the data perfectly. On the other hand, there also exists algorithms that use statistical methods to derive approximate formulas from noisy data. An example is a work by Kim et al. [96]. Extending our algorithm to handle noise in the data is a promising direction for future work, which we have partially addressed in recent work [68].

To learn from positive examples only, none of the above methods provide good results: they return a trivial solution that accepts all possible examples. However, the problem of learning a specification from a single type of examples has been studied in other contexts. One prominent example is process mining (see Aalst et al. [15] for an overview), where the problem is the following: given a log of events generated by some process, find a model that satisfies specific properties of interest. Typically, such properties include fitness (the model should be consistent with the examples from the log), precision (the model should not be overly general), generalization (the model should not be overly tight), and simplicity (the model should be simple). Different operationalizations of these four properties give rise to different problem formulations and solutions. By choosing UVW as our model, we get (structural) simplicity and connect it to the generalization property by the tightness value n, for which we require the tightest possible UVW consistent with the data.

Closely related to our algorithm for learning UVW is an algorithm by Avellaneda and Petrenko [25] for inferring deterministic automata over finite words from positive examples alone. This algorithm searches for an automaton $\mathcal{A}$ with a fixed number n of states that is consistent with the given positive examples and for which no n-state DFA $\mathcal{A}'$ exists such that the language of $\mathcal{A}'$ is a strict subset of the language of $\mathcal{A}$. Both our method and the one by Avellaneda and Petrenko identify the language to be learned in the limit and use a single additional parameter for choosing the complexity of the language to be learned. Unlike our approach, however, the resulting language in Avellaneda and Petrenko's algorithm is not unique for a given value of n. Furthermore,

while our approach is relatively simple to adapt to the finite-word setting, their approach is much harder to adapt to the infinite-word setting, which we support in our work.

Inverse reinforcement learning [118] is another related problem in which learning happens over (positive) demonstrations only. However, instead of human-understandable specifications of the task at hand, inverse reinforcement learning aims to infer reward functions, which numerically capture the rewards an agent receives while performing specific actions. Inspired by this setting, Vazquez-Chanlatte et al. [148] have developed an algorithm to learn LTL-like temporal descriptions of a reward function from demonstrations. The underlying idea is to proceed in two steps: first, a lattice of implications between all possible descriptions is pre-computed; second, demonstrations are used to rule out non-consistent descriptions, thereby "traversing" the lattice until all descriptions have been processed. While this approach has shown promising performance on real-world examples, its downside is that the first step is highly computationally expensive and only works if the number of syntactically distinct descriptions is finite. By contrast, our algorithm does not require such a pre-computation since we use the syntactic approximation of language inclusion between simple chains of UVWs. However, Vazquez-Chanlatte et al.'s algorithm successfully handles noise in the sample, which we currently cannot do.

5 CONCLUSION

Building safe and reliable systems is hard and involves a large number of highly complex, manual tasks. By using an innovative combination of inductive methods from the area of machine learning and deductive methods from the area of logic, this work has demonstrated how ***intelligent formal methods*** can significantly simplify or even entirely automate many of these tasks. In particular, we have covered three of the most essential topics in system safety and reliability: software verification, the synthesis of reactive systems and program code, as well as the challenging and error-prone task of writing formal specifications.

As a byproduct of our research, we have developed an extensive, learning-based toolbox that complements and improves existing, purely deductive methods. However, a host of challenges still exist that prevent the widespread adaptation of formal methods in practice. Let us conclude this work by sketching how our research can help mitigate the most important ones. As in the earlier parts of this work, we structure our discussion along the topics of *verification*, *synthesis*, and *formal specifications*.

In formal verification, one of the most pressing problems is to provide effective methods to prove the safety and reliability of artificial intelligence. Due to the ubiquitous use of machine learning technology in modern intelligent systems, this task is exceptionally challenging. On the one hand, machine learning models differ drastically from classical hard- and software, and the development of efficient tools for reasoning about statistical models is still in its infancy. On the other hand, intelligent systems are often an amalgamation of different components (including hard- and software, cyber-physical components, and machine learning models), and any end-to-end verification approach for intelligent systems needs to involve methods from various disciplines. However, the intelligent formal methods we have developed in this work serve as an ideal foundation to meet these challenges. In particular, our future research will be targeted at the definition of a standardized interface between the individual components of an intelligent system (exploiting the synergies between logic and learning) that allows (a) decomposing a global specification into local specifications for each component and (b) orchestrating specialized tools to verify each local specification individually. In this context, our ultimate goal is to develop effective methods for the end-to-end verification of intelligent systems that provide a high degree of automation.

In automated synthesis, an emerging trend is to synthesize reactive systems or programs that integrate machine learning models trained to perform specific, low-level tasks (e.g., perception). Initial results show the great potential of this novel approach, but its

fundamental properties, including its strengths, weaknesses, and limitations, are not yet well understood. Our future research will address this shortcoming. In particular, our overall goal is to develop a general framework that can model and explain similarities and differences of techniques that synthesize systems with integrated machine learning technology. Similar to our abstract learning frameworks for synthesis, this will help us identify a wide range of domains where such type of synthesis is applicable and simplify the design and analysis of synthesis algorithms through a common vocabulary.

Finally, a major obstacle in the area of formal specifications is the mismatch between an engineer's intuitive understanding of a complex system and the need to express this intuition in terms of formal logic. To bridge this gap, we plan to develop a completely novel, computer-aided approach to writing formal specifications, which we term *logic sketching*. The vision is that a future engineer writes a partial specification (a specification sketch), where parts that are difficult to formalize can be left unspecified. By interacting with the engineer (e.g., by querying for examples of the system's desired behavior), an automated tool would then infer the missing parts and complete the specification the engineer has in mind. While our learning algorithms for LTL and PSL specifications are helpful first steps in this direction, they clearly require fundamental extensions to fit into the much more demanding sketching scenario laid out above. Our future research will focus on this topic.

PRIMARY BIBLIOGRAPHY

[1] Rüdiger Ehlers, Ivan Gavran, and Daniel Neider. "Learning Properties in LTL ∩ ACTL from Positive Examples Only". In: *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*. IEEE, 2020, pp. 104–112. DOI: 10.34727/2020/isbn.978-3-85448-042-6_17.

[2] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. "Horn-ICE learning for synthesizing invariants and contracts". In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 131:1–131:25. DOI: 10.1145/3276501.

[3] Pranav Garg, Daniel Neider, P. Madhusudan, and Dan Roth. "Learning invariants using decision trees and implication counterexamples". In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. Ed. by Rastislav Bodík and Rupak Majumdar. ACM, 2016, pp. 499–512. DOI: 10.1145/2837614.2837664.

[4] Christof Löding, P. Madhusudan, and Daniel Neider. "Abstract Learning Frameworks for Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 167–185. DOI: 10.1007/978-3-662-49674-9_10.

[5] Oliver Markgraf, Chih-Duo Hong, Anthony W. Lin, Muhammad Najib, and Daniel Neider. "Parameterized Synthesis with Safety Properties". In: *Programming Languages and Systems - 18th Asian Symposium, APLAS 2020, Fukuoka, Japan, November 30 - December 2, 2020, Proceedings*. Ed. by Bruno C. d. S. Oliveira. Vol. 12470. Lecture Notes in Computer Science. Springer, 2020, pp. 273–292. DOI: 10.1007/978-3-030-64437-6_14.

[6] Daniel Neider, Pranav Garg, P. Madhusudan, Shambwaditya Saha, and Daejun Park. "Invariant Synthesis for Incomplete Verification Engines". In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 232–250. DOI: 10.1007/978-3-319-89960-2_13.

[7] Daniel Neider and Ivan Gavran. "Learning Linear Temporal Properties". In: *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*. Ed. by Nikolaj Bjørner and Arie Gurfinkel. IEEE, 2018, pp. 1–10. DOI: 10.23919/FMCAD.2018.8603016.

[8] Daniel Neider, P. Madhusudan, Shambwaditya Saha, Pranav Garg, and Daejun Park. "A Learning-Based Approach to Synthesizing Invariants for Incomplete Verification Engines". In: *J. Autom. Reason.* 64.7 (2020), pp. 1523–1552. DOI: 10.1007/s10817-020-09570-z.

[9] Daniel Neider and Oliver Markgraf. "Learning-Based Synthesis of Safety Controllers". In: *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*. Ed. by Clark W. Barrett and Jin Yang. IEEE, 2019, pp. 120–128. DOI: 10.23919/FMCAD.2019.8894254.

[10] Daniel Neider, Shambwaditya Saha, Pranav Garg, and P. Madhusudan. "Sorcar: Property-Driven Algorithms for Learning Conjunctive Invariants". In: *Static Analysis - 26th International Symposium, SAS 2019, Porto, Portugal, October 8-11, 2019, Proceedings*. Ed. by Bor-Yuh Evan Chang. Vol. 11822. Lecture Notes in Computer Science. Springer, 2019, pp. 323–346. DOI: 10.1007/978-3-030-32304-2_16.

[11] Daniel Neider, Shambwaditya Saha, and P. Madhusudan. "Compositional Synthesis of Piece-Wise Functions by Learning Classifiers". In: *ACM Trans. Comput. Log.* 19.2 (2018), 10:1–10:23. DOI: 10.1145/3173545.

[12] Daniel Neider, Shambwaditya Saha, and P. Madhusudan. "Synthesizing Piece-Wise Functions by Learning Classifiers". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 186–203. DOI: 10.1007/978-3-662-49674-9_11.

[13] Daniel Neider and Ufuk Topcu. "An Automaton Learning Approach to Solving Safety Games over Infinite Graphs". In: *Tools and Algorithms for the Construction and Analysis of Systems - 22nd International Conference, TACAS 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. Ed. by Marsha Chechik and Jean-François Raskin. Vol. 9636. Lecture Notes in Computer Science. Springer, 2016, pp. 204–221. DOI: 10.1007/978-3-662-49674-9_12.

[14] Rajarshi Roy, Dana Fisman, and Daniel Neider. “Learning Interpretable Models in the Property Specification Language”. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*. Ed. by Christian Bessiere. ijcai.org, 2020, pp. 2213–2219. DOI: 10.24963/ijcai.2020/306.

SECONDARY BIBLIOGRAPHY

[15] Wil M. P. van der Aalst, Josep Carmona, Thomas Chatain, and Boudewijn F. van Dongen. "A Tour in Process Mining: From Practice to Algorithmic Challenges". In: *Trans. Petri Nets Other Model. Concurr.* 14 (2019), pp. 1–35. DOI: 10.1007/978-3-662-60651-3_1.

[16] Keerthi Adabala and Rüdiger Ehlers. "A Fragment of Linear Temporal Logic for Universal Very Weak Automata". In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. Ed. by Shuvendu K. Lahiri and Chao Wang. Vol. 11138. Lecture Notes in Computer Science. Springer, 2018, pp. 335–351. DOI: 10.1007/978-3-030-01090-4_20.

[17] Aws Albarghouthi, Josh Berdine, Byron Cook, and Zachary Kincaid. "Spatial Interpolants". In: *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. Ed. by Jan Vitek. Vol. 9032. Lecture Notes in Computer Science. Springer, 2015, pp. 634–660. DOI: 10.1007/978-3-662-46669-8_26.

[18] Rajeev Alur, Rastislav Bodík, Eric Dallal, Dana Fisman, Pranav Garg, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shambwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. "Syntax-Guided Synthesis". In: *Dependable Software Systems Engineering*. Ed. by Maximilian Irlbeck, Doron A. Peled, and Alexander Pretschner. Vol. 40. NATO Science for Peace and Security Series, D: Information and Communication Security. IOS Press, 2015, pp. 1–25. DOI: 10.3233/978-1-61499-495-4-1.

[19] Rajeev Alur, Dana Fisman, Rishabh Singh, and Armando Solar-Lezama. "SyGuS-Comp 2017: Results and Analysis". In: *Proceedings Sixth Workshop on Synthesis, SYNT@CAV 2017, Heidelberg, Germany, 22nd July 2017*. Ed. by Dana Fisman and Swen Jacobs. Vol. 260. EPTCS. 2017, pp. 97–115. DOI: 10.4204/EPTCS.260.9.

[20] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. "Scaling Enumerative Program Synthesis via Divide and Conquer". In: *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings,*

Part I. Ed. by Axel Legay and Tiziana Margaria. Vol. 10205. Lecture Notes in Computer Science. 2017, pp. 319–336. DOI: 10.1007/978-3-662-54577-5_18.

[21] Rajeev Alur and Nimit Singhania. "Precise piecewise affine models from input-output data". In: *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*. Ed. by Tulika Mitra and Jan Reineke. ACM, 2014, 3:1–3:10. DOI: 10.1145/2656045.2656064.

[22] Dana Angluin. "Learning Regular Sets from Queries and Counterexamples". In: *Inf. Comput.* 75.2 (1987), pp. 87–106. DOI: 10.1016/0890-5401(87)90052-6.

[23] Roy Armoni, Limor Fix, Alon Flaisher, Rob Gerth, Boris Ginsburg, Tomer Kanza, Avner Landver, Sela Mador-Haim, Eli Singerman, Andreas Tiemeyer, Moshe Y. Vardi, and Yael Zbar. "The ForSpec Temporal Logic: A New Temporal Property-Specification Language". In: *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*. Ed. by Joost-Pieter Katoen and Perdita Stevens. Vol. 2280. Lecture Notes in Computer Science. Springer, 2002, pp. 296–211. DOI: 10.1007/3-540-46002-0_21.

[24] IEEE Standards Association. "IEEE Standard for Property Specification Language (PSL)". In: *IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005)* (2010), pp. 1–182. DOI: 10.1109/IEEESTD.2010.5446004.

[25] Florent Avellaneda and Alexandre Petrenko. "Inferring DFA without Negative Examples". In: *Proceedings of the 14th International Conference on Grammatical Inference, ICGI 2018, Wrocaw, Poland, September 5-7, 2018*. Ed. by Olgierd Unold, Witold Dyrka, and Wojciech Wieczorek. Vol. 93. Proceedings of Machine Learning Research. PMLR, 2018, pp. 17–29.

[26] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008. ISBN: 978-0-262-02649-9.

[27] Thomas Ball, Rupak Majumdar, Todd D. Millstein, and Sriram K. Rajamani. "Automatic Predicate Abstraction of C Programs". In: *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*. Ed. by Michael Burke and Mary Lou Soffa. ACM, 2001, pp. 203–213. DOI: 10.1145/378795.378846.

[28] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. "cvc5: A Versatile and Industrial-Strength SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as*

Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I. Ed. by Dana Fisman and Grigore Rosu. Vol. 13243. Lecture Notes in Computer Science. Springer, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24.

[29] Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs". In: *Formal Methods for Components and Objects, 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever. Vol. 4111. Lecture Notes in Computer Science. Springer, 2005, pp. 364–387. DOI: 10.1007/11804192_17.

[30] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The Satisfiability Modulo Theories Library (SMT-LIB)*. http://smtlib.cs.uiowa.edu/. 2016.

[31] Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4". In: *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Springer, 2011, pp. 171–177. DOI: 10.1007/978-3-642-22110-1_14.

[32] Alberto Bemporad, Andrea Garulli, Simone Paoletti, and Antonio Vicino. "A bounded-error approach to piecewise affine system identification". In: *IEEE Trans. Autom. Control.* 50.10 (2005), pp. 1567–1580. DOI: 10.1109/TAC.2005.856667.

[33] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. "GPUVerify: a verifier for GPU kernels". In: *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*. Ed. by Gary T. Leavens and Matthew B. Dwyer. ACM, 2012, pp. 113–132. DOI: 10.1145/2384616.2384625.

[34] Tewodros A. Beyene, Swarat Chaudhuri, Corneliu Popeea, and Andrey Rybalchenko. "A constraint-based approach to solving games on infinite graphs". In: *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. Ed. by Suresh Jagannathan and Peter Sewell. ACM, 2014, pp. 221–234. DOI: 10.1145/2535838.2535860.

[35] Tewodros A. Beyene, Corneliu Popeea, and Andrey Rybalchenko. "Solving Existentially Quantified Horn Clauses". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture

Notes in Computer Science. Springer, 2013, pp. 869–882. DOI: 10.1007/978-3-642-39799-8_61.

[36] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. "The software model checker Blast". In: *Int. J. Softw. Tools Technol. Transf.* 9.5-6 (2007), pp. 505–525. DOI: 10.1007/s10009-007-0044-z.

[37] Alan W. Biermann and Jerome A. Feldman. "On the Synthesis of Finite-State Machines from Samples of Their Behavior". In: *IEEE Trans. Computers* 21.6 (1972), pp. 592–597. DOI: 10.1109/TC.1972.5009015.

[38] Nikolaj Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. "On Solving Universally Quantified Horn Clauses". In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 105–125. DOI: 10.1007/978-3-642-38856-9_8.

[39] Roderick Bloem, Barbara Jobstmann, Nir Piterman, Amir Pnueli, and Yaniv Sa'ar. "Synthesis of Reactive(1) designs". In: *J. Comput. Syst. Sci.* 78.3 (2012), pp. 911–938. DOI: 10.1016/j.jcss.2011.08.007.

[40] Ahmed Bouajjani, Bengt Jonsson, Marcus Nilsson, and Tayssir Touili. "Regular Model Checking". In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 403–418. DOI: 10.1007/10722167_31.

[41] Aaron R. Bradley. "SAT-Based Model Checking without Unrolling". In: *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*. Ed. by Ranjit Jhala and David A. Schmidt. Vol. 6538. Lecture Notes in Computer Science. Springer, 2011, pp. 70–87. DOI: 10.1007/978-3-642-18275-4_7.

[42] Aaron R. Bradley and Zohar Manna. *The calculus of computation - decision procedures with applications to verification*. Springer, 2007. DOI: 10.1007/978-3-540-74113-8.

[43] Tomás Brázdil, Krishnendu Chatterjee, Jan Kretínský, and Viktor Toman. "Strategy Representation by Decision Trees in Reactive Synthesis". In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 385–407. DOI: 10.1007/978-3-319-89960-2_21.

[44] Marc Brockschmidt, Yuxin Chen, Pushmeet Kohli, Siddharth Krishna, and Daniel Tarlow. “Learning Shape Analysis”. In: *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. Ed. by Francesco Ranzato. Vol. 10422. Lecture Notes in Computer Science. Springer, 2017, pp. 66–87. DOI: 10.1007/978-3-319-66706-5_4.

[45] Cristiano Calcagno, Dino Distefano, Peter W. O’Hearn, and Hongseok Yang. “Compositional Shape Analysis by Means of Bi-Abduction”. In: *J. ACM* 58.6 (2011), 26:1–26:66. DOI: 10.1145/2049697.2049700.

[46] Alberto Camacho and Sheila A. McIlraith. “Learning Interpretable Models Expressed in Linear Temporal Logic”. In: *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS 2018, Berkeley, CA, USA, July 11-15, 2019*. Ed. by J. Benton, Nir Lipovetzky, Eva Onaindia, David E. Smith, and Siddharth Srivastava. AAAI Press, 2019, pp. 621–630.

[47] Pavol Cerný, Edmund M. Clarke, Thomas A. Henzinger, Arjun Radhakrishna, Leonid Ryzhyk, Roopsha Samanta, and Thorsten Tarrach. “From Non-preemptive to Preemptive Scheduling Using Synchronization Synthesis”. In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 180–197. DOI: 10.1007/978-3-319-21668-3_11.

[48] Adrien Champion, Tomoya Chiba, Naoki Kobayashi, and Ryosuke Sato. “ICE-Based Refinement Type Discovery for Higher-Order Functional Programs”. In: *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part I*. Ed. by Dirk Beyer and Marieke Huisman. Vol. 10805. Lecture Notes in Computer Science. Springer, 2018, pp. 365–384. DOI: 10.1007/978-3-319-89960-2_20.

[49] Yu-Fang Chen, Chih-Duo Hong, Anthony W. Lin, and Philipp Rümmer. “Learning to prove safety over parameterised concurrent systems”. In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. Ed. by Daryl Stewart and Georg Weissenbacher. IEEE, 2017, pp. 76–83. DOI: 10.23919/FMCAD.2017.8102244.

[50] Alonzo Church. “Application of Recursive Arithmetic to the Problem of Circuit Synthesis”. In: *Summaries of the Summer Institute of Symbolic Logic*. Vol. I. Cornell Univ., Ithaca, 1957, pp. 3–50.

[51] Alonzo Church. "Application of Recursive Arithmetic to the Problem of Circuit Synthesis". In: *Journal of Symbolic Logic* 28.4 (1963), pp. 289–290. DOI: 10.2307/2271310.

[52] Vasek Chvátal. "A Greedy Heuristic for the Set-Covering Problem". In: *Math. Oper. Res.* 4.3 (1979), pp. 233–235. DOI: 10.1287/moor.4.3.233.

[53] Michael Colón, Sriram Sankaranarayanan, and Henny Sipma. "Linear Invariant Generation Using Non-linear Constraint Solving". In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 420–432. DOI: 10.1007/978-3-540-45069-6_39.

[54] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". In: *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*. Ed. by Robert M. Graham, Michael A. Harrison, and Ravi Sethi. ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973.

[55] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976. ISBN: 013215871X.

[56] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. "Inductive invariant generation via abductive inference". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 443–456. DOI: 10.1145/2509136.2509511.

[57] William F. Dowling and Jean H. Gallier. "Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae". In: *J. Log. Program.* 1.3 (1984), pp. 267–284. DOI: 10.1016/0743-1066(84)90014-1.

[58] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in Property Specifications for Finite-State Verification". In: *Proceedings of the 1999 International Conference on Software Engineering, ICSE' 99, Los Angeles, CA, USA, May 16-22, 1999*. Ed. by Barry W. Boehm, David Garlan, and Jeff Kramer. ACM, 1999, pp. 411–420. DOI: 10.1145/302405.302672.

[59] Rüdiger Ehlers. "ACTL ∩ LTL Synthesis". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture

Notes in Computer Science. Springer, 2012, pp. 39–54. DOI: 10.1007/978-3-642-31424-7_9.

[60] Rüdiger Ehlers. "Computing the Complete Pareto Front". In: *CoRR* abs/1512.05207 (2015). arXiv: 1512.05207.

[61] Cindy Eisner and Dana Fisman. *A Practical Introduction to PSL*. Series on Integrated Circuits and Systems. Springer, 2006. ISBN: 978-0-387-35313-5. DOI: 10.1007/978-0-387-36123-9.

[62] Michael D. Ernst, Adam Czeisler, William G. Griswold, and David Notkin. "Quickly detecting relevant program invariants". In: *Proceedings of the 22nd International Conference on on Software Engineering, ICSE 2000, Limerick Ireland, June 4-11, 2000*. Ed. by Carlo Ghezzi, Mehdi Jazayeri, and Alexander L. Wolf. ACM, 2000, pp. 449–458. DOI: 10.1145/337180.337240.

[63] Grigory Fedyukovich, Samuel J. Kaufman, and Rastislav Bodík. "Sampling invariants from frequency distributions". In: *2017 Formal Methods in Computer Aided Design, FMCAD 2017, Vienna, Austria, October 2-6, 2017*. Ed. by Daryl Stewart and Georg Weissenbacher. IEEE, 2017, pp. 100–107. DOI: 10.23919/FMCAD.2017.8102247.

[64] Giancarlo Ferrari-Trecate, Marco Muselli, Diego Liberati, and Manfred Morari. "A clustering technique for the identification of piecewise affine systems". In: *Autom.* 39.2 (2003), pp. 205–217. DOI: 10.1016/S0005-1098(02)00224-8.

[65] Jean-Christophe Filliâtre and Andrei Paskevich. "Why3 - Where Programs Meet Provers". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8.

[66] Cormac Flanagan and K. Rustan M. Leino. "Houdini, an Annotation Assistant for ESC/Java". In: *FME 2001: Formal Methods for Increasing Software Productivity, International Symposium of Formal Methods Europe, Berlin, Germany, March 12-16, 2001, Proceedings*. Ed. by José Nuno Oliveira and Pamela Zave. Vol. 2021. Lecture Notes in Computer Science. Springer, 2001, pp. 500–517. DOI: 10.1007/3-540-45251-6_29.

[67] Robert W Floyd. "Assigning meanings to programs". In: *Program Verification*. Springer, 1993, pp. 65–81.

[68] Jean-Raphaël Gaglione, Daniel Neider, Rajarshi Roy, Ufuk Topcu, and Zhe Xu. "Learning Linear Temporal Properties from Noisy Data: A MaxSAT-based Approach". In: *Automated Technology for Verification and Analysis, 19th International Symposium, ATVA 2021, Gold Coast (Online), Australia, October 18-22, 2021. Proceedings*. Lecture Notes in Computer Science. Springer, 2021, to appear.

[69] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. "ICE: A Robust Framework for Learning Invariants". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 69–87. DOI: 10.1007/978-3-319-08867-9_5.

[70] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. "Learning Universally Quantified Invariants of Linear Data Structures". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 813–829. DOI: 10.1007/978-3-642-39799-8_57.

[71] Pranav Garg, Christof Löding, P. Madhusudan, and Daniel Neider. "Quantified data automata for linear data structures: a register automaton model with applications to learning invariants of programs manipulating arrays and lists". In: *Formal Methods Syst. Des.* 47.1 (2015), pp. 120–157. DOI: 10.1007/s10703-015-0231-6.

[72] Yeting Ge and Leonardo Mendonça de Moura. "Complete Instantiation for Quantified Formulas in Satisfiabiliby Modulo Theories". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 306–320. DOI: 10.1007/978-3-642-02658-4_25.

[73] E Mark Gold. "Complexity of automaton identification from given data". In: *Information and Control* 37.3 (1978), pp. 302–320. ISSN: 0019-9958. DOI: https://doi.org/10.1016/S0019-9958(78)90562-4.

[74] Erich Grädel, Wolfgang Thomas, and Thomas Wilke, eds. *Automata, Logics, and Infinite Games: A Guide to Current Research [outcome of a Dagstuhl seminar, February 2001]*. Vol. 2500. Lecture Notes in Computer Science. Springer, 2002. ISBN: 3-540-00388-6. DOI: 10.1007/3-540-36387-4.

[75] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. "Synthesizing software verifiers from proof rules". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. Ed. by Jan Vitek, Haibo Lin, and Frank Tip. ACM, 2012, pp. 405–416. DOI: 10.1145/2254064.2254112.

[76] Sumit Gulwani. "Automating string processing in spreadsheets using input-output examples". In: *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*. Ed. by Thomas Ball and Mooly Sagiv. ACM, 2011, pp. 317–330. DOI: 10.1145/1926385.1926423.

[77] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. "Synthesis of loop-free programs". In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 62–73. DOI: 10.1145/1993498.1993506.

[78] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. "Program analysis as constraint solving". In: *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. Ed. by Rajiv Gupta and Saman P. Amarasinghe. ACM, 2008, pp. 281–292. DOI: 10.1145/1375581.1375616.

[79] Ashutosh Gupta and Andrey Rybalchenko. "InvGen: An Efficient Invariant Generator". In: *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proceedings*. Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. Lecture Notes in Computer Science. Springer, 2009, pp. 634–640. DOI: 10.1007/978-3-642-02658-4_48.

[80] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. "The SeaHorn Verification Framework". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 343–361. DOI: 10.1007/978-3-319-21690-4_20.

[81] Peter Habermehl and Tomás Vojnar. "Regular Model Checking Using Inference of Regular Languages". In: *Proceedings of the 6th International Workshop on Verification of Infinite-State Systems, INFINITY 2004, London, UK, September 4, 2004*. Ed. by Julian C. Bradfield and Faron Moller. Vol. 138. Electronic Notes in Theoretical Computer Science 3. Elsevier, 2004, pp. 21–36. DOI: 10.1016/j.entcs.2005.01.044.

[82] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming". In: *Commun. ACM* 12.10 (1969), pp. 576–580. DOI: 10.1145/363235.363259.

[83] Gerard J. Holzmann. "The logic of bugs". In: *Proceedings of the Tenth ACM SIGSOFT Symposium on Foundations of Software Engineering 2002, Charleston, South Carolina, USA, November 18-22, 2002*. ACM, 2002, pp. 81–87. DOI: 10.1145/587051.587064.

[84] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004. ISBN: 978-0-321-22862-8.

[85] Marieke Huisman, Dilian Gurov, and Alexander Malkis. "Formal Methods: From Academia to Industrial Practice. A Travel Guide". In: *CoRR* abs/2002.07279 (2020). arXiv: 2002.07279.

[86] Laurent Hyafil and Ronald L. Rivest. "Constructing Optimal Binary Decision Trees is NP-Complete". In: *Inf. Process. Lett.* 5.1 (1976), pp. 15–17. DOI: 10.1016/0020-0190(76)90095-8.

[87] Shachar Itzhaky, Anindya Banerjee, Neil Immerman, Aleksandar Nanevski, and Mooly Sagiv. "Effectively-Propositional Reasoning about Reachability in Linked Data Structures". In: *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. Ed. by Natasha Sharygina and Helmut Veith. Vol. 8044. Lecture Notes in Computer Science. Springer, 2013, pp. 756–772. DOI: 10.1007/978-3-642-39799-8_53.

[88] Shachar Itzhaky, Nikolaj Bjørner, Thomas W. Reps, Mooly Sagiv, and Aditya V. Thakur. "Property-Directed Shape Analysis". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 35–51. DOI: 10.1007/978-3-319-08867-9_3.

[89] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. "Oracle-guided component-based program synthesis". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. Ed. by Jeff Kramer, Judith Bishop, Premkumar T. Devanbu, and Sebastián Uchitel. ACM, 2010, pp. 215–224. DOI: 10.1145/1806799.1806833.

[90] Susmit Jha and Sanjit A. Seshia. "A theory of formal synthesis via inductive learning". In: *Acta Informatica* 54.7 (2017), pp. 693–726. DOI: 10.1007/s00236-017-0294-5.

[91] Ranjit Jhala and Kenneth L. McMillan. "A Practical and Complete Approach to Predicate Refinement". In: *Tools and Algorithms for the Construction and Analysis of Systems, 12th International Conference, TACAS 2006 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS*

2006, Vienna, Austria, March 25 - April 2, 2006, Proceedings. Ed. by Holger Hermanns and Jens Palsberg. Vol. 3920. Lecture Notes in Computer Science. Springer, 2006, pp. 459–473. DOI: 10.1007/11691372_33.

[92] Xiaoqing Jin, Alexandre Donzé, Jyotirmoy V. Deshmukh, and Sanjit A. Seshia. "Mining requirements from closed-loop control models". In: *Proceedings of the 16th international conference on Hybrid systems: computation and control, HSCC 2013, April 8-11, 2013, Philadelphia, PA, USA*. Ed. by Calin Belta and Franjo Ivancic. ACM, 2013, pp. 43–52. DOI: 10.1145/2461328.2461337.

[93] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. "Property-Directed Inference of Universal Invariants or Proving Their Absence". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 583–602. DOI: 10.1007/978-3-319-21690-4_40.

[94] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[95] Michael J. Kearns and Umesh V. Vazirani. *An Introduction to Computational Learning Theory*. MIT Press, 1994. ISBN: 978-0-262-11193-5.

[96] Joseph Kim, Christian Muise, Ankit Shah, Shubham Agarwal, and Julie Shah. "Bayesian Inference of Linear Temporal Logic Specifications for Contrastive Explanations". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*. Ed. by Sarit Kraus. ijcai.org, 2019, pp. 5591–5598. DOI: 10.24963/ijcai.2019/776.

[97] James C. King. "Symbolic Execution and Program Testing". In: *Commun. ACM* 19.7 (1976), pp. 385–394. DOI: 10.1145/360248.360252.

[98] Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. "Deductive Program Repair". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9207. Lecture Notes in Computer Science. Springer, 2015, pp. 217–233. DOI: 10.1007/978-3-319-21668-3_13.

[99] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. "Synthesis modulo recursive functions". In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. Ed. by Antony L. Hosking, Patrick Th. Eugster, and Cristina V. Lopes. ACM, 2013, pp. 407–426. DOI: 10.1145/2509136.2509555.

[100] Zhaodan Kong, Austin Jones, and Calin Belta. "Temporal Logics for Learning and Detection of Anomalous Behavior". In: *IEEE Trans. Autom. Control.* 62.3 (2017), pp. 1210–1222. DOI: 10.1109/TAC.2016.2585083.

[101] Orna Kupferman, Nir Piterman, and Moshe Y. Vardi. "An Automata-Theoretic Approach to Infinite-State Systems". In: *Time for Verification, Essays in Memory of Amir Pnueli*. Ed. by Zohar Manna and Doron A. Peled. Vol. 6200. Lecture Notes in Computer Science. Springer, 2010, pp. 202–259. DOI: 10.1007/978-3-642-13754-9_11.

[102] Akash Lal and Shaz Qadeer. "Powering the static driver verifier using corral". In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 202–212. DOI: 10.1145/2635868.2635894.

[103] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. "A Solver for Reachability Modulo Theories". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 427–443. DOI: 10.1007/978-3-642-31424-7_32.

[104] Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. "Shape Analysis via Second-Order Bi-Abduction". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 52–68. DOI: 10.1007/978-3-319-08867-9_4.

[105] Caroline Lemieux, Dennis Park, and Ivan Beschastnikh. "General LTL Specification Mining (T)". In: *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. Ed. by Myra B. Cohen, Lars Grunske, and Michael Whalen. IEEE Computer Society, 2015, pp. 81–92. DOI: 10.1109/ASE.2015.71.

[106] Wenchao Li, Lili Dworkin, and Sanjit A. Seshia. "Mining assumptions for synthesis". In: *9th IEEE/ACM International Conference on Formal Methods and Models for Codesign, MEMOCODE 2011, Cambridge, UK, 11-13 July,*

2011. Ed. by Satnam Singh, Barbara Jobstmann, Michael Kishinevsky, and Jens Brandt. IEEE, 2011, pp. 43–50. DOI: 10.1109/MEMCOD.2011.5970509.

[107] Anthony W. Lin and Philipp Rümmer. "Liveness of Randomised Parameterised Systems under Arbitrary Schedulers". In: *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Vol. 9780. Lecture Notes in Computer Science. Springer, 2016, pp. 112–133. DOI: 10.1007/978-3-319-41540-6_7.

[108] Christof Löding, P. Madhusudan, and Lucas Peña. "Foundations for natural proofs and quantifier instantiation". In: *Proc. ACM Program. Lang.* 2.POPL (2018), 10:1–10:30. DOI: 10.1145/3158098.

[109] Monika Maidl. "The Common Fragment of CTL and LTL". In: *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*. IEEE Computer Society, 2000, pp. 643–652. DOI: 10.1109/SFCS.2000.892332.

[110] Kenneth L. McMillan. "Interpolation and SAT-Based Model Checking". In: *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*. Ed. by Warren A. Hunt Jr. and Fabio Somenzi. Vol. 2725. Lecture Notes in Computer Science. Springer, 2003, pp. 1–13. DOI: 10.1007/978-3-540-45069-6_1.

[111] Robert McNaughton. "Infinite Games Played on Finite Graphs". In: *Ann. Pure Appl. Logic* 65.2 (1993), pp. 149–184. DOI: 10.1016/0168-0072(93)90036-D.

[112] Sara Mohammadinejad, Jyotirmoy V. Deshmukh, Aniruddh G. Puranic, Marcell Vazquez-Chanlatte, and Alexandre Donzé. "Interpretable classification of time-series data using efficient enumerative techniques". In: *HSCC '20: 23rd ACM International Conference on Hybrid Systems: Computation and Control, Sydney, New South Wales, Australia, April 21-24, 2020*. Ed. by Aaron D. Ames, Sanjit A. Seshia, and Jyotirmoy Deshmukh. ACM, 2020, 9:1–9:10. DOI: 10.1145/3365365.3382218.

[113] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Efficient E-Matching for SMT Solvers". In: *Automated Deduction - CADE-21, 21st International Conference on Automated Deduction, Bremen, Germany, July 17-20, 2007, Proceedings*. Ed. by Frank Pfenning. Vol. 4603. Lecture Notes in Computer Science. Springer, 2007, pp. 183–198. DOI: 10.1007/978-3-540-73595-3_13.

[114] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest,*

Hungary, March 29-April 6, 2008. Proceedings. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24.

[115] Daniel Neider. "Reachability Games on Automatic Graphs". In: *Implementation and Application of Automata - 15th International Conference, CIAA 2010, Winnipeg, MB, Canada, August 12-15, 2010. Revised Selected Papers*. Ed. by Michael Domaratzki and Kai Salomaa. Vol. 6482. Lecture Notes in Computer Science. Springer, 2010, pp. 222–230. DOI: 10.1007/978-3-642-18098-9_24.

[116] Daniel Neider. "Small Strategies for Safety Games". In: *Automated Technology for Verification and Analysis, 9th International Symposium, ATVA 2011, Taipei, Taiwan, October 11-14, 2011. Proceedings*. Ed. by Tevfik Bultan and Pao-Ann Hsiung. Vol. 6996. Lecture Notes in Computer Science. Springer, 2011, pp. 306–320. DOI: 10.1007/978-3-642-24372-1_22.

[117] Daniel Neider and Nils Jansen. "Regular Model Checking Using Solver Technologies and Automata Learning". In: *NASA Formal Methods, 5th International Symposium, NFM 2013, Moffett Field, CA, USA, May 14-16, 2013. Proceedings*. Ed. by Guillaume Brat, Neha Rungta, and Arnaud Venet. Vol. 7871. Lecture Notes in Computer Science. Springer, 2013, pp. 16–31. DOI: 10.1007/978-3-642-38088-4_2.

[118] Andrew Y. Ng and Stuart J. Russell. "Algorithms for Inverse Reinforcement Learning". In: *Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000), Stanford University, Stanford, CA, USA, June 29 - July 2, 2000*. Ed. by Pat Langley. Morgan Kaufmann, 2000, pp. 663–670.

[119] ThanhVu Nguyen, Deepak Kapur, Westley Weimer, and Stephanie Forrest. "Using dynamic analysis to discover polynomial and array invariants". In: *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. Ed. by Martin Glinz, Gail C. Murphy, and Mauro Pezzè. IEEE Computer Society, 2012, pp. 683–693. DOI: 10.1109/ICSE.2012.6227149.

[120] José Oncina and Pedro Garcia. "Inferring regular languages in polynomial updated time". In: *Pattern recognition and image analysis: selected papers from the IVth Spanish Symposium*. Vol. 1. World Scientific. 1992, pp. 49–61.

[121] Saswat Padhi, Rahul Sharma, and Todd D. Millstein. "Data-driven precondition inference with learned features". In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery Berger. ACM, 2016, pp. 42–56. DOI: 10.1145/2908080.2908099.

[122] Simone Paoletti, Aleksandar Lj. Juloski, Giancarlo Ferrari-Trecate, and René Vidal. “Identification of Hybrid Systems: A Tutorial”. In: *Eur. J. Control* 13.2-3 (2007), pp. 242–260. DOI: 10.3166/ejc.13.242-260.

[123] Zvonimir Pavlinovic, Akash Lal, and Rahul Sharma. “Inferring annotations for device drivers from verification histories”. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, Singapore, September 3-7, 2016*. Ed. by David Lo, Sven Apel, and Sarfraz Khurshid. ACM, 2016, pp. 450–460. DOI: 10.1145/2970276.2970305.

[124] Edgar Pek, Xiaokang Qiu, and P. Madhusudan. “Natural proofs for data structure manipulation in C using separation logic”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’14, Edinburgh, United Kingdom - June 09 - 11, 2014*. Ed. by Michael F. P. O’Boyle and Keshav Pingali. ACM, 2014, pp. 440–451. DOI: 10.1145/2594291.2594325.

[125] Amir Pnueli. “The Temporal Logic of Programs”. In: *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 1977, pp. 46–57. DOI: 10.1109/SFCS.1977.32.

[126] Amir Pnueli and Elad Shahar. “Liveness and Acceleration in Parameterized Verification”. In: *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*. Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. Lecture Notes in Computer Science. Springer, 2000, pp. 328–343. DOI: 10.1007/10722167_26.

[127] Xiaokang Qiu, Pranav Garg, Andrei Stefanescu, and Parthasarathy Madhusudan. “Natural proofs for structure, data, and separation”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI ’13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 231–242. DOI: 10.1145/2491956.2462169.

[128] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993. ISBN: 1-55860-238-0.

[129] J. Ross Quinlan. “Induction of Decision Trees”. In: *Mach. Learn.* 1.1 (1986), pp. 81–106. DOI: 10.1023/A:1022643204877.

[130] Gunther Reissig, Alexander Weber, and Matthias Rungger. “Feedback Refinement Relations for the Synthesis of Symbolic Controllers”. In: *IEEE Trans. Autom. Control.* 62.4 (2017), pp. 1781–1796. DOI: 10.1109/TAC.2016.2593947.

[131] Henry Gordon Rice. “Classes of recursively enumerable sets and their decision problems”. In: *Transactions of the American Mathematical Society* 74 (1953), pp. 358–366. DOI: 10.2307/1990888.

[132] Heinz Riener. "Exact Synthesis of LTL Properties from Traces". In: *2019 Forum for Specification and Design Languages, FDL 2019, Southampton, United Kingdom, September 2-4, 2019*. Ed. by Tom J. Kazmierski, Reinhard von Hanxleden, and Terrence S. T. Mak. IEEE, 2019, pp. 1–6. DOI: 10.1109/FDL.2019.8876900.

[133] Ronald L. Rivest and Robert E. Schapire. "Inference of Finite Automata Using Homing Sequences". In: *Inf. Comput.* 103.2 (1993), pp. 299–347. DOI: 10.1006/inco.1993.1021.

[134] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. "Parametric Shape Analysis via 3-Valued Logic". In: *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*. Ed. by Andrew W. Appel and Alex Aiken. ACM, 1999, pp. 105–118. DOI: 10.1145/292540.292552.

[135] Shambwaditya Saha, Pranav Garg, and P. Madhusudan. "Alchemist: Learning Guarded Affine Functions". In: *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*. Ed. by Daniel Kroening and Corina S. Pasareanu. Vol. 9206. Lecture Notes in Computer Science. Springer, 2015, pp. 440–446. DOI: 10.1007/978-3-319-21690-4_26.

[136] Eric Schkufza, Rahul Sharma, and Alex Aiken. "Stochastic superoptimization". In: *Architectural Support for Programming Languages and Operating Systems, ASPLOS '13, Houston, TX, USA - March 16 - 20, 2013*. Ed. by Vivek Sarkar and Rastislav Bodík. ACM, 2013, pp. 305–316. DOI: 10.1145/2451116.2451150.

[137] Rahul Sharma and Alex Aiken. "From Invariant Checking to Invariant Inference Using Randomized Search". In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. Ed. by Armin Biere and Roderick Bloem. Vol. 8559. Lecture Notes in Computer Science. Springer, 2014, pp. 88–105. DOI: 10.1007/978-3-319-08867-9_6.

[138] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, Percy Liang, and Aditya V. Nori. "A Data Driven Approach for Algebraic Loop Invariants". In: *Programming Languages and Systems - 22nd European Symposium on Programming, ESOP 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. Ed. by Matthias Felleisen and Philippa Gardner. Vol. 7792. Lecture Notes in Computer Science. Springer, 2013, pp. 574–592. DOI: 10.1007/978-3-642-37036-6_31.

[139] Rahul Sharma, Saurabh Gupta, Bharath Hariharan, Alex Aiken, and Aditya V. Nori. "Verification as Learning Geometric Concepts". In: *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. Ed. by Francesco Logozzo and Manuel Fähndrich. Vol. 7935. Lecture Notes in Computer Science. Springer, 2013, pp. 388–411. DOI: 10.1007/978-3-642-38856-9_21.

[140] Rahul Sharma, Aditya V. Nori, and Alex Aiken. "Interpolants as Classifiers". In: *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 71–87. DOI: 10.1007/978-3-642-31424-7_11.

[141] Armando Solar-Lezama. "Program sketching". In: *Int. J. Softw. Tools Technol. Transf.* 15.5-6 (2013), pp. 475–495. DOI: 10.1007/s10009-012-0249-7.

[142] Armando Solar-Lezama. "Program synthesis by sketching". PhD thesis. University of California at Berkeley, 2008.

[143] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. "Combinatorial sketching for finite programs". In: *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*. Ed. by John Paul Shen and Margaret Martonosi. ACM, 2006, pp. 404–415. DOI: 10.1145/1168857.1168907.

[144] Wolfgang Thomas. "Church's Problem and a Tour through Automata Theory". In: *Pillars of Computer Science, Essays Dedicated to Boris (Boaz) Trakhtenbrot on the Occasion of His 85th Birthday*. Ed. by Arnon Avron, Nachum Dershowitz, and Alexander Rabinovich. Vol. 4800. Lecture Notes in Computer Science. Springer, 2008, pp. 635–655. DOI: 10.1007/978-3-540-78127-1_35.

[145] Wolfgang Thomas. "Facets of Synthesis: Revisiting Church's Problem". In: *Foundations of Software Science and Computational Structures, 12th International Conference, FOSSACS 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*. Ed. by Luca de Alfaro. Vol. 5504. Lecture Notes in Computer Science. Springer, 2009, pp. 1–14. DOI: 10.1007/978-3-642-00596-1_1.

[146] Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M. K. Martin, and Rajeev Alur. "TRANSIT: specifying protocols with concolic snippets". In: *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 287–296. DOI: 10.1145/2491956.2462174.

[147] BEL V, BfS, CNSC, CSN, ISTec, ONR, SSM, and STUK. *Licensing of safety critical software for nuclear reactors : Common position of seven European nuclear regulators and authorised technical support organisations.* Tech. rep. Revision 2014. Salzgitter, Germany: Bundesamt für Strahlenschutz (BfS), 2015.

[148] Marcell Vazquez-Chanlatte, Susmit Jha, Ashish Tiwari, Mark K. Ho, and Sanjit A. Seshia. "Learning Task Specifications from Demonstrations". In: *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada.* Ed. by Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett. 2018, pp. 5372–5382.

[149] Yakir Vizel, Arie Gurfinkel, Sharon Shoham, and Sharad Malik. "IC3 - Flipping the E in ICE". In: *Verification, Model Checking, and Abstract Interpretation - 18th International Conference, VMCAI 2017, Paris, France, January 15-17, 2017, Proceedings.* Ed. by Ahmed Bouajjani and David Monniaux. Vol. 10145. Lecture Notes in Computer Science. Springer, 2017, pp. 521–538. DOI: 10.1007/978-3-319-52234-0_28.

[150] Andrzej Wasylkowski and Andreas Zeller. "Mining Temporal Specifications from Object Usage". In: *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009.* IEEE Computer Society, 2009, pp. 295–306. DOI: 10.1109/ASE.2009.30.

[151] Pierre Wolper. "Temporal Logic Can Be More Expressive". In: *22nd Annual Symposium on Foundations of Computer Science, Nashville, Tennessee, USA, 28-30 October 1981.* IEEE Computer Society, 1981, pp. 340–348. DOI: 10.1109/SFCS.1981.44.

[152] Qiwen Xu, Willem P. de Roever, and Jifeng He. "The Rely-Guarantee Method for Verifying Shared Variable Concurrent Programs". In: *Formal Aspects Comput.* 9.2 (1997), pp. 149–174. DOI: 10.1007/BF01211617.

[153] He Zhu, Stephen Magill, and Suresh Jagannathan. "A data-driven CHC solver". In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018.* Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 707–721. DOI: 10.1145/3192366.3192416.

[154] He Zhu, Aditya V. Nori, and Suresh Jagannathan. "Learning refinement types". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.* Ed. by Kathleen Fisher and John H. Reppy. ACM, 2015, pp. 400–411. DOI: 10.1145/2784731.2784766.

[155] He Zhu, Gustavo Petri, and Suresh Jagannathan. “Automatically learning shape specifications”. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*. Ed. by Chandra Krintz and Emery Berger. ACM, 2016, pp. 491–507. DOI: 10.1145/2908080.2908125.

INDEX