DIPLOMATERVEZÉSI FELADAT

Tegzes Tamás
Mérnökinformatikus hallgató részére

Tanuló és Szintézis Algoritmusokkal Támogatott Szoftver Verifikáció

A szoftverek egyre nagyobb részt vállalnak napjaink rendszereinek működtetésében, amely során egyre több kritikus funkciót is rájuk bízunk. Ezen okból különösen fontos a szoftverek helyes működésének biztosítása. Ezt megtehetjük tervezési, fejlesztési vagy futási időben is. A fejlesztési idejű helyesség ellenőrzést szoftver verifikáció segítségével végezhetjük el.

Szoftverek verifikálása nehéz probléma algoritmikus szempontból: általános esetben eldönthetetlen egy programkód helyessége. Azonban egyre több praktikus módszer jelenik meg, amelyek segítségével egyre több szoftver helyessége vizsgálható.

Napjainkban trend, hogy a tradicionális megközelítések mellett tanuló és egyéb, szintézis alapú módszereket vetünk be a szoftver verifikáció támogatására. A hallgató feladata megvizsgálni ezen módszereket, és egy prototípus szoftver verifikációs algoritmusba integrálni őket.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a szoftver verifikáció módszereit.
- Vizsgálja meg a szakirodalomban fellelhető szintézis és tanuló algoritmusokat, amelyeket szoftver verifikáció támogatására ajánlanak.
- Tervezzen meg egy szoftver verifikációs megközelítést, amely kombinálja az irodalomban fellelhető megoldásokat.
- Implementálja a megtervezett rendszer prototípusát.
- Mérésekkel vizsgálja meg a megközelítés hatékonyságát és alkalmazhatóságát.

Tanszéki konzulens: Dr. Vörös András, docens

Budapest, 2020.03.10.

…………………………..
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár, DSc

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Méréstechnika és Információs Rendszerek Tanszék

1117 Budapest, Magyar Tudósok krt. 2. I. ép. I.E.444.
Telefon: 463-2057, Fax: 463-4112
http://www.mit.bme.hu • e-mail: mitadm@mit.bme.hu

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Learning and Synthesis Supported Software Verification

Master's Thesis

Author
Tamás Tegzes

Advisor
Dr. András Vörös

December 19, 2020

Contents

Kivonat **i**

Abstract **ii**

1 Introduction **1**

2 Background **3**

2.1 Formal logic 3

2.2 Control flow automata 5

2.2.1 Basics 5

2.2.2 Paths and loops 7

2.2.3 Structural nodes and edges 10

2.2.4 Invariant systems 11

2.3 Horn clauses 14

2.4 Constraints 15

3 The Algorithm **19**

3.1 Overview 19

3.2 Finding structural nodes 19

3.3 Teacher 21

3.3.1 Solver 21

3.3.2 Teacher algorithm 22

3.4 Learners 24

3.4.1 Datapoints 25

3.4.1.1 Checking constraint consistency 27

3.4.1.2 Keeping track of datapoints 30

3.4.2 Learner algorithms 39

3.4.2.1 Simple learner 40

3.4.2.2 Sorcar learner 41

3.4.2.3 Decision tree learner 45

4 Implementation **54**
4.1 Main modules . . . 54
4.2 Learner combinations . . . 55
4.3 Predicate patterns . . . 55
4.4 Configurability . . . 56
5 Evaluation **57**
5.1 Methodology . . . 57
5.2 Tested configurations . . . 57
5.3 Results . . . 59
6 Conclusion **65**
Bibliography **66**

HALLGATÓI NYILATKOZAT

Alulírott *Tegzes Tamás*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2020. december 19.

Tegzes Tamás
hallgató

Kivonat

Életünk egyre nagyobb részét automatizáljuk, egyre több problémát szoftverrendszerek segítségével oldunk meg. Akár olyan feladatokat is szofverrendszerekre bízunk, amelyek során ezek nem megfelelő működése végzetes következményekkel járhat. Míg a pénzügyi rendszereket vagy kritikus infrastruktúrát irányító szoftverek hibái jelentős gazdasági kárt okozhatnak, a repülőgépekben vagy orvosi eszközökben működő szoftverek hibája esetén életveszély állhat fenn. Az ilyen rendszerekben futó szoftverek hibáinak kiszűrése ezért kitüntetett figyelmet érdemel.

A fejlesztési folyamatba integrált verifikációs módszerek segítik a szoftverhibák felismerését és emelik a szoftver minőségét. Hagyományos verifikációs módszerekkel (mint amilyen a tesztelés) általában csak csökkenteni lehet a hibák jelenlétének valószínűségét, az ember nem lehet teljesen biztos benne, hogy a ki nem próbált esetekben is jól működik-e a tesztelt szoftver. A formális módszerek ezzel szemben lehetőséget adnak rá, hogy bizonyos formálisan megfogalmazott tulajdonságok teljesülését matematikailag bizonyítsuk.

A formális módszerek egyik ága a modellellenőrzés. A modellellenőrző algoritmusok jellemzője, hogy az ellenőrzendő rendszer egy formális modelljét, illetve a modell állapotterét vizsgálják.

A dolgozatban vizsgált algortmuscsalád szoftver modellek ellenőrzésére képes, az ellenőrzött program ciklusaihoz próbál invariánsokat szintetizálni. Ezek olyan logikai formulák, amelyek indukcióval bizonyítható lemmákat alkotnak, és együtt képesek a modell helyességét bizonyítani.

Az invariánsok szintézisének feladatát tekinthetjük egy speciális Horn-klóz halmaz megoldásának. A Horn-ICE verifikációs eszköz egy tanár és egy tanuló modul együttműködésének eredményeként képes Horn-klózokat megoldani. A tanuló invariáns-jelölteket szintetizál, a tanár pedig ellenőrzi őket. Ha a jelöltek nem bizonyulnak valódi invariánsoknak, a tanár mintákat ad a tanulónak, amik alapján a tanuló (akár a gépi tanulásból ismert algoritmusok használatával) javít a jelöltjein.

A dolgozatban néhány, az irodalomból megismert Horn-klóz megoldó algoritmust adaptálunk a feladatra, ötvözzük őket és mérésekkel vizsgáljuk az elkészült prototítpus hatékonyságát. Különlegessége a bemutatott megoldásnak, hogy míg a Horn-ICE eszköz esetében a tanár mintái legfeljebb a program egy állapotára vonatkoznak, a mi eszközünk olyan mintákat is képes adni, amely a program végtelen sok állapotát lefedi.

Abstract

An increasing part of our lives is being automated. We solve more and more problems with software systems. We even trust software systems with tasks where their potential improper operation can have catastrophic consequences. While bugs in software that control financial systems or critical infrastructure can lead to economic damage, bugs in the software of an aeroplane or a medical device may endanger life. Ensuring that there are no bugs in the software that runs on such systems deserves special attention.

Verification methods integrated into the development process support the detection of bugs and improve the quality of the software. Using traditional verification methods (such as testing), one can usually only reduce the probability of bugs being present, but they can never be completely sure that the software under test works correctly in the cases they did not try. Using Formal methods, on the other hand, one can prove some formally stated properties mathematically.

One of the branches of formal methods is model checking. Model checking systems work with a formal model of the system. They traverse the state space of the model and check if the property they are trying to prove is satisfied.

The family of algorithms we examine in the thesis can check software models by trying to synthesize loop invariants. These are logical formulae that form lemmas about the program. The lemmas are proven by induction, and together they prove that the model is correct.

We can view the task of synthesizing invariants as a special case of solving Horn clauses. The Horn-ICE verification toolkit solves Horn clauses through the collaboration of a teacher and a learner module. The learner synthesizes invariant candidates, and the teacher checks them. If the candidates are not invariants, the teacher gives samples to the learner, which the learner can use to improve the candidates—for example with algorithms known from machine learning.

In this thesis, we adapt some previously published Horn clause solver algorithms to invariant synthesis, combine them and measure the efficiency of the completed prototype. A special feature of the presented solution is that the samples that the teacher gives to the learner can represent infinitely many states of the program, as opposed to the Horn-ICE toolkit, where the samples only represent a single program state.

Chapter 1

Introduction

The more important a task we entrust to a software system, the more important it is to ensure that the system works correctly. We have to take more precaution developing the control software for a nuclear power plant than developing a game.

Errors in some systems, for example aeroplanes, railway control systems or medical systems, can have significant consequences, they may lead to death or significant damage to their environment. We call such systems *safety critical systems*.

Since many safety critical systems rely on software, the utmost care must be taken to make sure that the software they rely on is correct to prevent the consequences of a malfunction. A major part of that is *verification*. Verification is the process of ensuring the quality of software. We can differentiate *dynamic* verification methods, which work by executing the software to be verified, and *static* verification methods, which analyse the code without executing it.

Formal verification is a static verification method. It requires the software and the requirements both to be stated formally, and it either proves by formal methods that the software satisfies the requirements or offers an example of when it does not. While e.g. testing can only offer information on whether the program behaves correctly in the chosen test cases, and one must extrapolate from that to other cases, formal verification can reason about all scenarios that can occur when executing the software.

Our approach to formal verification is *model checking*. As seen in Figure 1.1, model checking algorithms work with the formal model of the system to be checked, and they check whether it adheres to some formally stated requirements. They either prove that all possible states in the state space of the model satisfy the requirements or provide an example of when the system fails.

In this thesis, we present a family of algorithms with the goal of synthesizing loop invariants for a software model. Loop invariants are logical formulae that evaluate true for every reachable configuration of the software model, and that fact can be proven by induction. Additionally, we aim to synthesize loop invariants that are strong enough to prove that the software system is safe.

Related work Invariant synthesis is a special case of solving Horn clauses. An overview of Horn clauses and existing techniques to solving them can be found in [2]. Our solution is based on the Horn-ICE toolkit [3, 6]. Other approaches to Horn-clause solving include [7] and [4]. We implemented the prototypes using the Theta [11] framework.

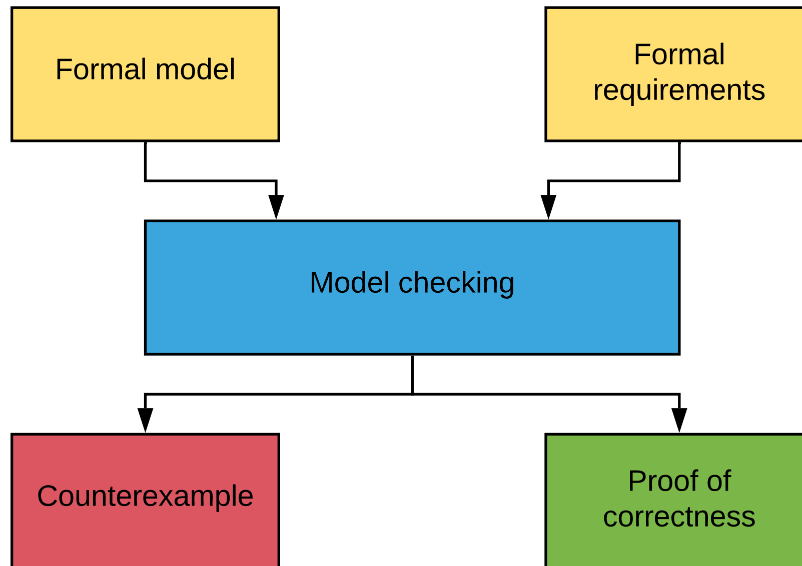

Figure 1.1: Overview of model checking

Structure of the thesis We introduce the mathematical background to the algorithms, define the used formalisms and formally state the task of invariant synthesis in Chapter 2. In Chapter 3, we present the algorithms and describe how they interact. Chapter 4 has details about our implementation of a prototype. In Chapter 5 we evaluate the proposed solutions and draw conclusions. Finally, Chapter 6 concludes the thesis.

Chapter 2

Background

2.1 Formal logic

In this thesis, we use first-order logic under the satisfiability modulo theory of linear integer arithmetic ($\mathcal{LA}(\mathbb{Z})$). Thus, the domain of terms is $\mathbb{Z}$. Formulae are quantifier-free, and the only uninterpreted symbols in them are variables unless explicitly noted otherwise.

By atoms, we mean a relation applied to terms. By literals, we mean an atom or the negation of an atom.

For a set of formulae $\mathbf{F}$,

$$\bigwedge(\mathbf{F}) \mathrel{\hat{=}} \left(\bigwedge_{\varphi \in \mathbf{F}} \varphi \right).$$

Moreover, $\bigwedge(\varnothing) \mathrel{\hat{=}} \top$.

Definition 1 (Clause). A clause is a disjunction of literals. ▪

Example 1. $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_n$, *where* φ_i *are literals, is a clause.*

By $\mathbb{F}(X)$, we mean the set of formulae over X.

Definition 2 (Valuation). A valuation over the set of variables X is a partial function $\alpha : X \nrightarrow \mathbb{Z}$, which assigns an integer value to some of the variables in X.

The value α assigns to $x \in X$ is noted $\alpha(x)$. The set of variables α assigns value to is noted $\mathcal{X}_\alpha$.

For valuations α and β over the set of variables X and one of its subsets $Y \subseteq X$,

- $\alpha[Y]$ is a valuation such that $\mathcal{X}_{\alpha[Y]} = Y \cap \mathcal{X}_\alpha$ and for all $x \in Y \cap \mathcal{X}_\alpha$, $\alpha[Y](x) = \alpha(x)$, i.e., $\alpha[Y]$ assigns the same value as α to variables in Y and omits other assignments;
- $\alpha \subseteq \beta$, if $\mathcal{X}_\beta \subseteq \mathcal{X}_\alpha$ and for all $x \in \mathcal{X}_\beta$, $\alpha(x) = \beta(x)$, i.e., if α assigns the same value to all of the variables β assigns value to, and α possibly assigns value to others;
- α and β are disjoint if there is an $x \in (\mathcal{X}_\alpha \cap \mathcal{X}_\beta)$ for which $\alpha(x) \neq \beta(x)$, i.e., if there is a variable they assign different values to;
- if α and β are not disjoint, then $\alpha \oplus \beta$ is a valuation such that $\mathcal{X}_{\alpha\oplus\beta} = \mathcal{X}_\alpha \cup \mathcal{X}_\beta$ and for all $x \in \mathcal{X}_\alpha \cup \mathcal{X}_\beta$

$$(\alpha \oplus \beta)(x) = \begin{cases} \alpha(x) & \text{if } x \in \mathcal{X}_\alpha \\ \beta(x) & \text{if } x \notin \mathcal{X}_\alpha. \end{cases}$$

We note the valuation that assigns a_1 to x_1, a_2 to x_2, etc. as $\{x_1 \to a_1, x_2 \to a_2, \ldots\}$. ▪

Definition 3 (Full valuation). A valuation α over a set of variables X is full if $\mathcal{X}_\alpha = X$, i.e., if it assigns a value to every $x \in X$. ▪

Example 2. *Over the set of variables* $X = \{x, y, z\}$, $\alpha = \{x \to 1, y \to 4\}$ *and* $\beta = \{x \to 2, z \to 32\}$ *are valuations. Neither is full, since* $\mathcal{X}_\alpha = \{x, y\} \neq X$ *and* $\mathcal{X}_\beta = \{x, z\} \neq X$. *In other words,* α *does not assign a value to* z *and* β *does not assign value to* y. *Moreover,* α *and* β *are disjoint, because they assign different values to* x. *Therefore,* $\alpha \oplus \beta$ *does not exist.*

For $Y = \{a, b, y, z\}$, $\alpha[Y] = \{y \to 4\}$ *and* $\beta[Y] = \{z \to 32\}$. *Now* α *and* $\beta[Y]$ *are not disjoint, therefore* $\gamma = \alpha \oplus (\beta[Y]) = \{x \to 1, y \to 4, z \to 32\}$ *exists, and it is a full valuation over* X. *Additionally,* $\gamma \subseteq \alpha$.

Definition 4. For a valuation α and a formula φ, $\alpha \models \varphi$ means that for every $\mathcal{LA}(\mathbb{Z})$-interpretation $\mathcal{M}$ that assigns $\alpha(x)$ to every $x \in \mathcal{X}_\alpha$, $\mathcal{M} \models \varphi$. ▪

Example 3. *Let* $\alpha = \{x \to 2, y \to (-2)\}$. *Then* $\alpha \models (x > y) \vee (z = 0)$, *because the formula evaluates to true regardless of the value of* z, *but* $\alpha \not\models (x > y) \wedge (z = 0)$, *since there are* $\mathcal{LA}(\mathbb{Z})$*-interpretations (where, e.g.,* $z = 1$*) that assign* 2 *to* x, *and* -2 *to* y *but for which the formula does not hold.*

Let $\beta = \alpha \oplus \{z \to 0\}$. $\beta \models (x > y) \wedge (z = 0)$.

Proposition 1. If, for a valuation α and a formula φ, $\alpha \models \varphi$, then for every valuation $\beta \subseteq \alpha$, $\beta \models \varphi$. ▪

Proof. Every $\mathcal{LA}(\mathbb{Z})$-interpretation $\mathcal{M}$ that is considered for $\beta \models \varphi$ (because it assigns $\beta(x)$ to every $x \in \mathcal{X}_\beta$) is also considered for $\alpha \models \varphi$, therefore $\mathcal{M} \models \varphi$. □

Proposition 2. For a full valuation $\alpha : X \to \mathbb{Z}$ and a formula φ over X, if every uninterpreted symbol in φ is a variable, then either $\alpha \models \varphi$ or $\alpha \models \neg\varphi$. ▪

Proof. The only uninterpreted symbols in φ are elements of X, and α sets their value. For a $\mathcal{LA}(\mathbb{Z})$-interpretation $\mathcal{M}$ that assigns $\alpha(x)$ to every $x \in \mathcal{X}_\alpha$, either $\mathcal{M} \models \varphi$, or $\mathcal{M} \models \neg\varphi$. Other such $\mathcal{LA}(\mathbb{Z})$ interpretations can only differ in the value assigned to symbols not present in φ, therefore, their evaluation of φ cannot differ. □

Proposition 3. If, for a valuation α and a formula φ over the set of variables X, $\alpha \not\models \varphi$ and $\alpha \not\models \neg\varphi$, then there are full valuations $\alpha_1 \subseteq \alpha$ and $\alpha_2 \subseteq \alpha$ over X such that $\alpha_1 \models \varphi$ and $\alpha_2 \models \neg\varphi$. ▪

Proof. For every $\mathcal{LA}(\mathbb{Z})$-interpretation $\mathcal{M}$, either $\mathcal{M} \models \varphi$, or $\mathcal{M} \models \neg\varphi$. From $\alpha \not\models \varphi$ and $\alpha \not\models \neg\varphi$, we know that φ does not evaluate the same for all of the $\mathcal{LA}(\mathbb{Z})$-interpretations that assign $\alpha(x)$ to every $x \in \mathcal{X}_\alpha$. Therefore, there must be interpretations $\mathcal{M}_1$ and $\mathcal{M}_2$ that assign $\alpha(x)$ to every $x \in \mathcal{X}_\alpha$ for which $\mathcal{M}_1 \models \varphi$ and $\mathcal{M}_2 \models \neg\varphi$. We can build full valuations α_1 from $\mathcal{M}_1$ and α_2 from $\mathcal{M}_2$ such that they assign the same value to every variable in X as their respective interpretations. This process ensures that $\alpha_1 \subseteq \alpha$ and $\alpha_2 \subseteq \alpha$. By Proposition 2, $\alpha_1 \models \varphi$ and $\alpha_2 \models \neg\varphi$. □

For a variable x and $i \in \mathbb{N}$, $x^{(i)}$ denotes x with i primes applied. For $i \neq j$, $x^{(i)}$ and $x^{(j)}$ are considered different variables. $x \equiv x^{(0)}$, $x' \equiv x^{(1)}$, $x'' \equiv x^{(2)}$, etc. For a set of variables X, $X^{(i)} = \left\{x^{(i)} : x \in X\right\}$. For a formula φ, $\varphi^{(i)}$ is φ with i primes applied to every occurrence of its every variable. For a valuation α, $\alpha^{(i)}$ is a valuation, for which $\mathcal{X}_{\alpha^{(i)}} = (\mathcal{X}_\alpha)^{(i)}$, and for all $x^{(i)} \in (\mathcal{X}_\alpha)^{(i)}$, $\alpha^{(i)}\left(x^{(i)}\right) = \alpha(x)$.

2.2 Control flow automata

For our purposes, the formal model of the software to verify is a control flow automaton, and the formal requirements are also stated within that.

2.2.1 Basics

The following definitions are based on similar definitions in [10].

Definition 5 (Control flow automata). A control flow automaton (CFA) is a tuple $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ where

- L is a finite set of locations, the nodes of the control flow graph,
- $E \subseteq L \times \mathit{Stmt} \times L$ is a finite set of edges, where for an edge $(\ell_a, s, \ell_b) \in E$,
  - $\ell_a \in L$ is the source location,
  - $s \in \mathit{Stmt}$ is the statement to be executed upon traversing the edge,
  - $\ell_b \in L$ is the target location;
- X is a set of first order logic variables,
- $\ell_s \in L$ is the initial location,
- $\ell_e \in L$ is the error location.

A statement $s \in \mathit{Stmt}$ is

- an assignment $x \coloneqq t$ where $x \in X$ is a variable and t is a term over X,
- an assumption $[\varphi]$ where φ is a formula over X
- or a statement of the form `havoc` x where $x \in X$ is a variable. ▪

Algorithm 1: Example algorithm

Input: Integers x and y

```
1  u ← x;
2  z ← y;
3  while u ≠ y do
4  |  if x < y then
5  |  |  z ← z + 1;
6  |  |  u ← u + 1;
7  |  else
8  |  |  z ← z − 1;
9  |  |  u ← u − 1;
10 |  end
11 end
12 assert(x + z = 2 · y);                    /* Requirement */
```

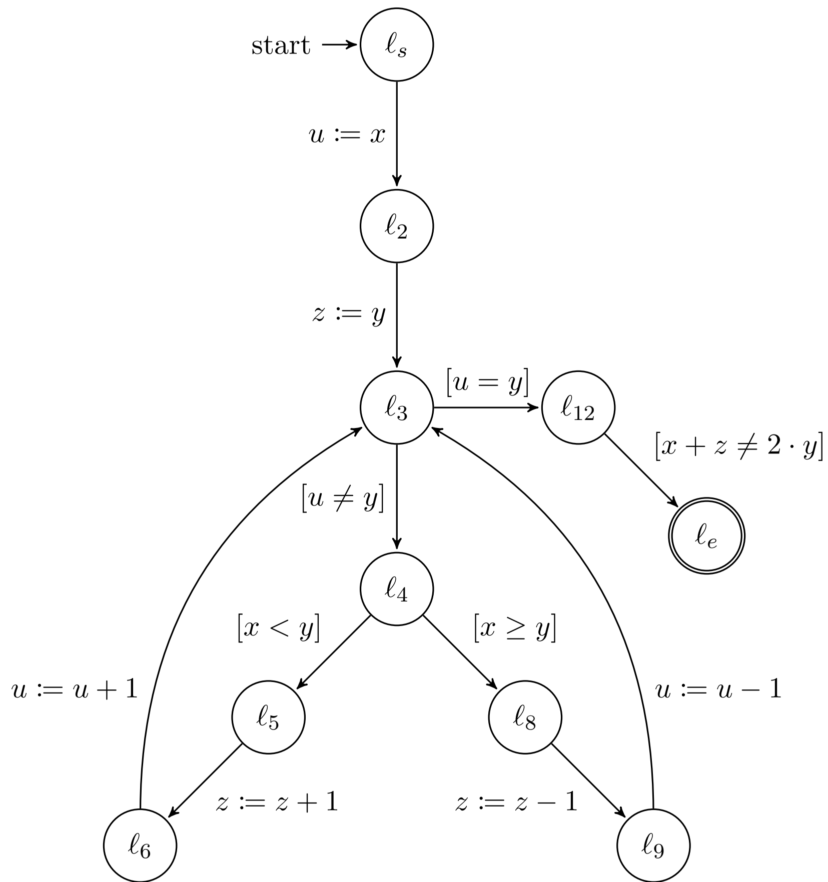

Figure 2.1: A graphical representation of the CFA created from Algorithm 1

Example 4. *The tuple* $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$*, where*

$$\begin{aligned}
L &= \{\ell_s, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6, \ell_8, \ell_9, \ell_{12}, \ell_e\},\\
E &= \{(\ell_s, u := x, \ell_2),\ (\ell_2, z := y, \ell_3),\ (\ell_3, [u \neq y], \ell_4),\ (\ell_3, [u = y], \ell_{12}),\\
&\quad (\ell_4, [x < y], \ell_5),\ (\ell_4, [x \geq y], \ell_8),\ (\ell_5, z := z + 1, \ell_6),\ (\ell_6, u := u + 1, \ell_3),\\
&\quad (\ell_8, z := z - 1, \ell_9),\ (\ell_9, u := u - 1, \ell_3),\ (\ell_{12}, [x + z \neq 2 \cdot y], \ell_e)\ \}, \textit{ and}\\
X &= \{x, y, u, z\}
\end{aligned}$$

is a CFA based on Algorithm 1. A graphical representation of $\mathcal{A}$ *is depicted in Figure 2.1.*

The configuration of the CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ is a pair (ℓ, α), where $\ell \in L$ is the current location and α, a full valuation over X, is the current value of the variables. The configuration evolves in steps. Initially, it is (ℓ_s, α_s) with α_s chosen nondeterministically. Then in every step, the CFA chooses one of the traversable edges $(\ell_a, s, \ell_b) \in E$ leading out of its current location, executes the statement s—which possibly changes the value of the variables—and changes its location to the target of the edge, ℓ_b.

An assumption statement $[\varphi]$ can only be executed—and edges that it appears on can only be traversed—if $\alpha \models \varphi$. Upon its execution, $[\varphi]$ does not change the valuation α.

Assignments $x := t$ and statements of the form $\texttt{havoc}\ x$ can always be executed, and edges they appear on can always be traversed. Upon their execution, they change the value of x in α. In the case of $\texttt{havoc}\ x$, the new value of x can be any integer chosen nondeterministically. In the case of $x := t$, the new value of x is the value of t by α.

To describe this more formally, we define the transition formula of a statement.

Definition 6 (Transition formula of a statement). For a statement $s \in \mathit{Stmt}$ in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, we define δ_s as its transition formula.

Let $\mathrm{same}(X) \mathrel{\hat{=}} (\bigwedge_{x \in X} x' = x)$.

$$\delta_s \mathrel{\hat{=}} \begin{cases} \varphi \wedge \mathrm{same}(X) & \text{if } s = [\varphi] \\ (x' = t) \wedge \mathrm{same}(X \setminus \{x\}) & \text{if } s = (x := t) \\ \mathrm{same}(X \setminus \{x\}) & \text{if } s = \texttt{havoc}\ x. \end{cases}$$

∎

In a step, the configuration of a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ can change from (ℓ_a, α_a) to (ℓ_b, α_b) if there is an edge $(\ell_a, s, \ell_b) \in E$ such that $(\alpha_a \oplus \alpha_b{}') \models \delta_s$.

Definition 7 (Set of possibly changed variables in a statement). For a statement $s \in \mathit{Stmt}$ in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, we define $\Delta_s \subseteq X$ as the set of variables possibly changed by s.

$$\begin{aligned}
\Delta_s &= \begin{cases} \varnothing & \text{if } s = [\varphi] \\ \{x\} & \text{if } s = (x := t) \\ \{x\} & \text{if } s = \texttt{havoc}\ x, \end{cases}\\
\overline{\Delta_s} &= X \setminus \Delta_s.
\end{aligned}$$

∎

2.2.2 Paths and loops

We now move from discussing single edges to sequences of edges: paths and loops.

Definition 8 (Path in a control flow automaton). A *path* (of length k) in a control flow automaton is a sequence of k edges

$$\Phi = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2) \dots (\ell_{k-1}, s_k, \ell_k)$$

such that after the first edge, the source location of every subsequent edge is the same as the target location of the previous edge.

We say that Φ starts at ℓ_0, and that it leads from ℓ_0 to ℓ_k. We introduce the notation $\mathrm{src}(\Phi)$ for the location Φ starts at, the notation $\mathrm{tgt}(\Phi)$ for the location it leads to and the notation $\mathrm{len}(\Phi)$ for its length. By the internal locations of Φ, we mean $\ell_1, \ell_2, \dots, \ell_{k-1}$. ▪

Definition 9 (Error path). An *error path* in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ is a path that leads from ℓ_s to ℓ_e. ▪

We can generalize the definitions we made for a single statement to a path, i.e., a sequence of statements.

Definition 10 (Transition formula of a path). For a path

$$\Phi = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2) \dots (\ell_{k-1}, s_k, \ell_k)$$

in a CFA, we define δ_Φ to be its transition formula.

$$\delta_\Phi \mathrel{\hat{=}} \bigwedge_{i=1}^{k} \delta_{s_i}^{(i-1)}.$$ ▪

Definition 11 (Set of variables possibly changed by a path). For a path

$$\Phi = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2) \dots (\ell_{k-1}, s_k, \ell_k)$$

in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, we define Δ_Φ as the set of variables possibly changed by its statements.

$$\Delta_\Phi = \bigcup_{i=1}^{k} \Delta_{s_i}$$

$$\overline{\Delta_\Phi} = X \setminus \Delta_\Phi = \bigcap_{i=1}^{k} \overline{\Delta_{s_i}}$$ ▪

Proposition 4. If there is a path Φ of length k and a set of valuations $\alpha_0, \alpha_1, \dots \alpha_k$ over X such that

$$\alpha_0 \oplus (\alpha_1)' \oplus \dots \oplus (\alpha_k)^{(k)} \models \delta_\Phi,$$

then

$$\alpha_0\left[\overline{\Delta_\Phi}\right] = \alpha_1\left[\overline{\Delta_\Phi}\right] = \dots = \alpha_k\left[\overline{\Delta_\Phi}\right].$$ ▪

Proof. Let $\Phi = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2) \dots (\ell_{k-1}, s_k, \ell_k)$.

Assume indirectly that there is an $i \in [0..k-1]$[1] such that $\alpha_i\left[\overline{\Delta_\Phi}\right] \neq \alpha_{i+1}\left[\overline{\Delta_\Phi}\right]$. Since $\delta_{s_{i+1}}^{(i)}$ is included in δ_Φ, $\alpha_i \oplus \alpha_{i+1} \models \delta_{s_i}$. By Definition 6, $\text{same}\left(\overline{\Delta_{s_i}}\right)$ is included in δ_{s_i}, and by Definition 11, $\overline{\Delta_\Phi} \subseteq \overline{\Delta_{s_i}}$.

If there is a variable $x \in \overline{\Delta_\Phi}$ such that $\alpha_i(x) \neq \alpha_{i+1}(x)$, then $\alpha_i \oplus \alpha_{i+1} \not\models \text{same}\left(\overline{\Delta_{s_i}}\right)$ and $\alpha_i \oplus \alpha_{i+1} \not\models \delta_{s_i}$.

If there is a variable $x \in \overline{\Delta_\Phi}$ for which either α_i or α_{i+1} assigns value, but not both, then the one that does not can be extended to assign a different value to x, therefore $\alpha_i \oplus \alpha_{i+1} \not\models \text{same}\left(\overline{\Delta_{s_i}}\right)$ and $\alpha_i \oplus \alpha_{i+1} \not\models \delta_{s_i}$.

Therefore, $\alpha_i\left[\overline{\Delta_\Phi}\right] = \alpha_{i+1}\left[\overline{\Delta_\Phi}\right]$, and we reached a contradiction. □

Similarly to how we defined when an edge can be traversed, we now define when a path is feasible.

Definition 12 (Path feasability). We say that a path Φ is feasible if and only if δ_Φ is satisfiable. ▪

Equivalently, a path Φ is feasible, if there is a starting configuration $(\text{src}(\Phi), \alpha_0)$ from which the edges of the path are traversable in order.

Example 5. *In the CFA $\mathcal{A}$ defined in Example 4 and depicted in Figure 2.1,*

$$\begin{aligned}\Phi = {} & (\ell_2, z \coloneqq y, \ell_3)\ (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x < y], \ell_5) \\ & (\ell_5, z \coloneqq z+1, \ell_6)\ (\ell_6, u \coloneqq u+1, \ell_3)\ (\ell_3, [u = y], \ell_{12})\end{aligned}$$

is a path that leads from ℓ_2 to ℓ_{12}. It is not an error path. Its set of possibly changed variables is $\Delta_\Phi = \{u, z\}$. Its transition formula is

$$\begin{aligned}\delta_\Phi \Leftrightarrow {} & z' = y \ \wedge\ \text{same}(\{x, y, u\}) \\ & \wedge\ u' \neq y' \ \wedge\ \text{same}(\{x', y', z', u'\}) \\ & \wedge\ x'' < y'' \ \wedge\ \text{same}(\{x'', y'', z'', u''\}) \\ & \wedge\ z^{(4)} = z^{(3)} + 1 \ \wedge\ \text{same}\left(\left\{x^{(3)}, y^{(3)}, u^{(3)}\right\}\right) \\ & \wedge\ u^{(5)} = u^{(4)} + 1 \ \wedge\ \text{same}\left(\left\{x^{(4)}, y^{(4)}, z^{(4)}\right\}\right) \\ & \wedge\ u^{(5)} = y^{(5)} \ \wedge\ \text{same}\left(\left\{x^{(5)}, y^{(5)}, z^{(5)}, u^{(5)}\right\}\right).\end{aligned}$$

It is feasible. A sample assignment of variables along the steps of its traversal can be seen in Table 2.1.

Definition 13 (Reachability of a node). We say that a node ℓ is reachable in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ if and only if there is a feasible path leading from ℓ_s to ℓ. ▪

Equivalently, a node ℓ is reachable if starting from one of the initial configurations, we can choose a finite number of edges to traverse that get the CFA to a configuration whose location is ℓ.

The decision problem of control flow automata is whether the error location ℓ_e is reachable. A counterexample is a feasible path leading from the initial location to the final location.

[1]For integers a and b, let $[a..b] \mathrel{\hat=} \{a, a+1, a+2, \ldots, b\}$

Location	x	y	z	u	Next statement
ℓ_2	10	48	-9	47	$z := y$
ℓ_3	10	48	48	47	$[u \neq y]$
ℓ_4	10	48	48	47	$[x < y]$
ℓ_5	10	48	48	47	$z := z + 1$
ℓ_6	10	48	49	47	$u := u + 1$
ℓ_3	10	48	49	48	$[u = y]$
ℓ_{12}	10	48	49	47	

Table 2.1: Example assignments to the variables of $\mathcal{A}$ in Example 5, showing the feasibility of Φ

Definition 14 (Loop in a CFA). A loop (of length k) is a path (of length k)

$$\Lambda = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2) \dots (\ell_{k-1}, s_k, \ell_0),$$

where the source location of the first edge is the same as the target location of the last edge. ▪

Example 6. *In the CFA $\mathcal{A}$ defined in Example 4 and depicted in Figure 2.1,*

$$\Lambda_1 = (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x < y], \ell_5)\ (\ell_5, z := z + 1, \ell_6)\ (\ell_6, u := u + 1, \ell_3) \text{ and}$$
$$\Lambda_2 = (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x \geq y], \ell_8)\ (\ell_8, z := z - 1, \ell_9)\ (\ell_9, u := u - 1, \ell_3)$$

are loops.

2.2.3 Structural nodes and edges

Loops are the primary obstacle to traversing the state space of a control flow automaton. Any path that goes through one of the locations of a loop can be extended by traversing the loop an arbitrary number of times, which leads to an infinite number of paths. Our approach is to choose a subset of the locations as structural nodes, such that at least one location is chosen from every loop. Then we only consider paths connecting structural nodes.

Definition 15 (Structural nodes). In a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, a set $N \subseteq L$ of locations is a set of structural nodes if there is an $i \in [0..k-1]$ for every loop $(\ell_0, s_1, \ell_1)(\ell_1, s_2, ell_2) \dots (\ell_{k-1}, s_k, \ell_0)$ in $\mathcal{A}$ such that $\ell_i \in N$. ▪

Definition 16 (Structural edges). In a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, for a set of structural nodes $N \subseteq L$, a structural edge is a path

$$\Phi = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2) \dots (\ell_{k-1}, s_k, \ell_k),$$

such that $\ell_0 \in N \cup \{\ell_s\}$, $\ell_k \in N \cup \{\ell_e\}$ and for all $i \in [1..k-1]$, $\ell_i \notin N$.

We permit $\ell_0 = \ell_k$ in a structural edge.

The set of structural edges in a CFA $\mathcal{A}$ for a set of structural nodes N is noted by $\mathrm{SE}(\mathcal{A}, N)$. ▪

The number of structural edges is fortunately finite, which makes them easier to work with.

Proposition 5. For a CFA $\mathcal{A}$ and a set of structural nodes N, the set of structural edges $\mathrm{SE}(\mathcal{A}, N)$ is finite. ▪

Proof. If in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, for a structural edge

$$\Phi = (\ell_0, s_1, \ell_1) \dots (\ell_a, s_{a+1}, \ell_{a+1}) \dots (\ell_{b-1}, s_b, \ell_b) \dots (\ell_{k-1}, s_k, \ell_k) \in \mathrm{SE}(\mathcal{A}, N),$$

$\ell_a = \ell_b$ for some $a, b \in [1..k-1]$, $a \neq b$, then the section $(\ell_a, s_{a+1}, \ell_{a+1}) \dots (\ell_{b-1}, s_b, \ell_b)$ would form a loop, and hence one of its locations would have to be a structural node. Structural edges cannot have structural nodes among their internal locations. Therefore, every edge can only appear at most once in a structural edge, and that makes structural edges permutations of subsets of E. Since E is a finite set, the set of its subsets is also finite, and the set of permutations for each subset is finite as well. Therefore, $\mathrm{SE}(\mathcal{A}, N)$ is the subset of a finite set. □

Example 7. *In the CFA $\mathcal{A}$ defined in Example 4 and depicted in Figure 2.1, $N_1 = \{\ell_3\}$ is a set of structural nodes, and the set of structural edges for it is $\mathrm{SE}(\mathcal{A}, N_1) = \{\Phi_1, \Phi_2, \Phi_3, \Phi_4\}$, where*

$$\begin{aligned}
\Phi_1 =& (\ell_s, u \coloneqq x, \ell_2)\ (\ell_2, z \coloneqq y, \ell_3),\\
\Phi_2 =& (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x < y], \ell_5)\ (\ell_5, z \coloneqq z + 1, \ell_6)\ (\ell_6, u \coloneqq u + 1, \ell_3),\\
\Phi_3 =& (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x \geq y], \ell_8)\ (\ell_8, z \coloneqq z - 1, \ell_9)\ (\ell_9, u \coloneqq u - 1, \ell_3),\\
\Phi_4 =& (\ell_3, [u = y], \ell_{12})\ (\ell_{12}, [x + z \neq 2 \cdot y], \ell_e).
\end{aligned}$$

$N_2 = \{\ell_5, \ell_8\}$ is also a set of structural nodes, and the set of structural edges for it is

$$\begin{aligned}
&\mathrm{SE}(\mathcal{A}, N_2) =\\
&\quad = \{(\ell_s, u \coloneqq x, \ell_2)\ (\ell_2, z \coloneqq y, \ell_3)\ (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x < y], \ell_5),\\
&\qquad (\ell_s, u \coloneqq x, \ell_2)\ (\ell_2, z \coloneqq y, \ell_3)\ (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x \geq y], \ell_8),\\
&\qquad (\ell_5, z \coloneqq z + 1, \ell_6)\ (\ell_6, u \coloneqq u + 1, \ell_3)\ (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x < y], \ell_5),\\
&\qquad (\ell_5, z \coloneqq z + 1, \ell_6)\ (\ell_6, u \coloneqq u + 1, \ell_3)\ (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x \geq y], \ell_8),\\
&\qquad (\ell_8, z \coloneqq z - 1, \ell_9)\ (\ell_9, u \coloneqq u - 1, \ell_3)\ (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x \geq y], \ell_8),\\
&\qquad (\ell_8, z \coloneqq z - 1, \ell_9)\ (\ell_9, u \coloneqq u - 1, \ell_3)\ (\ell_3, [u \neq y], \ell_4)\ (\ell_4, [x < y], \ell_5),\\
&\qquad (\ell_5, z \coloneqq z + 1, \ell_6)\ (\ell_6, u \coloneqq u + 1, \ell_3)\ (\ell_3, [u = y], \ell_{12})\ (\ell_{12}, [x + z \neq 2 \cdot y], \ell_e),\\
&\qquad (\ell_8, z \coloneqq z - 1, \ell_9)\ (\ell_9, u \coloneqq u - 1, \ell_3)\ (\ell_3, [u = y], \ell_{12})\ (\ell_{12}, [x + z \neq 2 \cdot y], \ell_e)\}.
\end{aligned}$$

2.2.4 Invariant systems

Given a set of structural nodes, we aim to synthesize a formula for every structural node that overapproximates[2] the set of reachable configurations in that location, and we try to make these formulae strong enough to prove that the error location is unreachable. In the following section, we formalize that goal and prove that reaching it is sufficient to prove that the error location is unreachable.

[2]Here, by overapproximation we mean that the set of reachable valuations is a subset of the set of valuations that satisfy the formula.

Definition 17 (Invariant system). In a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, for a set of structural nodes $N \subseteq L$, an invariant system is a function

$$\lambda : (N \cup \{\ell_s, \ell_e\}) \to \mathbb{F}(X)$$

for which $\lambda[\ell_s] \doteq \top$ and $\lambda[\ell_e] \doteq \bot$. It can map an arbitrary formula to members of N. ▪

Definition 18 (Satisfactory invariant system). In a CFA $\mathcal{A}$, for a set of structural nodes N, an invariant system λ is satisfactory if for all $\Phi \in \mathrm{SE}(\mathcal{A}, N)$, the formula

$$\lambda[\mathrm{src}(\Phi)] \wedge \delta_\Phi \to (\lambda[\mathrm{tgt}(\Phi)])^{(\mathrm{len}(\Phi))}$$

is a tautology, i.e., if the formula

$$\lambda[\mathrm{src}(\Phi)] \wedge \delta_\Phi \wedge \neg(\lambda[\mathrm{tgt}(\Phi)])^{(\mathrm{len}(\Phi))} \tag{2.1}$$

is unsatisfiable. ▪

Example 8. *In the CFA $\mathcal{A}$ defined in Example 4 and depicted in Figure 2.1, for the set of structural nodes $N_1 = \{\ell_3\}$, λ is a satisfactory invariant system, where*

$$\begin{aligned} &\lambda[\ell_0] \Leftrightarrow \top, \\ &\lambda[\ell_3] \Leftrightarrow (u - x = z - y), \\ &\lambda[\ell_s] \Leftrightarrow \bot. \end{aligned}$$

Initially, both sides of the $\lambda[\ell_3]$ equation are 0. Therefore,

$$\begin{aligned} \top \wedge (u' = x) &\wedge \mathrm{same}(\{x, y, z\}) \\ &\wedge (z'' = y') \wedge \mathrm{same}(\{x', y', u'\}) \\ &\wedge (u'' - x'' \neq z'' - y'') \end{aligned}$$

is unsatisfiable.

During a pass through the loop, both sides of the equation either increase or decrease by one, therefore they remain equal. The formulae

$$\begin{aligned} (u - x = z - y) &\wedge (u \neq y) \wedge \mathrm{same}(\{x, y, z, u\}) \\ &\wedge (x' < y') \wedge \mathrm{same}(\{x', y', z', u'\}) \\ &\wedge \left(z^{(3)} = z'' + 1\right) \wedge \mathrm{same}(\{x'', y'', u''\}) \\ &\wedge \left(u^{(4)} = u^{(3)} + 1\right) \wedge \mathrm{same}\left(\left\{x^{(3)}, y^{(3)}, z^{(3)}\right\}\right) \\ &\wedge \neg\left(u^{(4)} - x^{(4)} = z^{(4)} - y^{(4)}\right) \text{ and} \\ (u - x = z - y) &\wedge (u \neq y) \wedge \mathrm{same}(\{x, y, z, u\}) \\ &\wedge (x' \geq y') \wedge \mathrm{same}(\{x', y', z', u'\}) \\ &\wedge \left(z^{(3)} = z'' - 1\right) \wedge \mathrm{same}(\{x'', y'', u''\}) \\ &\wedge \left(u^{(4)} = u^{(3)} - 1\right) \wedge \mathrm{same}\left(\left\{x^{(3)}, y^{(3)}, z^{(3)}\right\}\right) \\ &\wedge \neg\left(u^{(4)} - x^{(4)} = z^{(4)} - y^{(4)}\right) \end{aligned}$$

are unsatisfiable.

The formula

$$\begin{aligned}(u - x = z - y) \wedge (u = y) \wedge \mathrm{same}(\{x, y, z, u\}) \\ \wedge (x' + z' \neq 2 \cdot y') \wedge \mathrm{same}(\{x', y', z', u'\}) \\ \wedge \top\end{aligned}$$

is unsatisfiable because

$$(u - x = z - y) \wedge (u = y) \Rightarrow (y - x = z - y) \Rightarrow (2 \cdot y = x + z).$$

Proposition 6. If a CFA has a satisfactory invariant system for a set of structural nodes, then it is safe, i.e., its error location is unreachable. ▪

Proof. Assume indirectly that in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ there is a set of structural nodes N for which λ is a satisfactory invariant system and there is a feasible path Φ leading from ℓ_s to ℓ_e and making $\mathcal{A}$ unsafe.

If $\Phi \in \mathrm{SE}(\mathcal{A}, N)$, then the formula

$$\lambda[\ell_s] \wedge \delta_\Phi \wedge \neg\lambda[\ell_e] \tag{2.2}$$

must be unsatisfiable, because λ is a satisfactory invariant system. But since Φ is feasible, the formula δ_Φ is satisfiable, and since $\lambda[\ell_s] \Leftrightarrow \top$ and $\lambda[\ell_e] \Leftrightarrow \bot$, the formula in Equation 2.2 is also satisfiable, and we reached contradiction.

If $\Phi \notin \mathrm{SE}(\mathcal{A}, N)$, then some of its internal locations are structural nodes. Let us note the internal locations of Φ by $\ell_1, \ell_2, \ldots, \ell_{k-1}$ where $k = \mathrm{len}(\Phi)$. Then for some $n_1, n_2, \ldots, n_{m-1} \in [1..k-1]$, $\ell_{n_1}, \ell_{n_2}, \ldots, \ell_{n_{m-1}} \in N$. Let $\ell_{n_0} = \ell_s$ and $\ell_{n_m} = \ell_e$. Divide Φ along these nodes into m structural edges $\Phi_1, \Phi_2, \ldots \Phi_m$ such that Φ_j ($j \in [1..m]$) leads from $\ell_{n_{j-1}}$ to ℓ_{n_j}. Let

$$X_j = \left(\bigcup_{i=n_{j-1}}^{n_j} X^{(i)} \right).$$

For all Φ_j, the formula

$$\lambda\left[\ell_{n_{j-1}}\right] \wedge \delta_{\Phi_j} \wedge \neg\left(\lambda\left[\ell_{n_j}\right]\right)^{(\mathrm{len}(\Phi_j))} \tag{2.3}$$

must be unsatisfiable, because λ is a satisfactory invariant system.

Φ is feasible, i.e., there is a full valuation

$$\alpha : \left(\bigcup_{i=0}^{k} X^{(i)} \right) \to \mathbb{Z},$$

for which $\alpha \models \delta_\Phi$, and more specifically,

$$\alpha[X_j] \models \left(\delta_{\Phi_j}\right)^{(n_{j-1})}.$$

Let $\alpha_i = \alpha\left[X^{(i)}\right]$. Since $\alpha_{n0} \models \lambda[\ell_{n0}]$ and $\alpha_{n_m} \models \neg(\lambda[\ell_{n_m}])^{(n_m)}$, and since every α_i is a full valuation, there must be a $p \in [1..m]$, such that

$$\alpha_{n_{p-1}} \models (\lambda[\ell_{n_{p-1}}])^{(n_{p-1})}, \text{but}$$
$$\alpha_{n_p} \models \neg(\lambda[\ell_{n_p}])^{(n_p)}.$$

We reach a contradiction by showing that the formula in Equation 2.3 is satisfiable for $j = p$.

$$\alpha[X_{n_p}] \models \lambda[\ell_{n_{p-1}}] \wedge \delta_{\Phi_p} \wedge \neg(\lambda[\ell_{n_p}])^{n_p - n_{(p-1)}}. \qquad \square$$

Therefore, finding a satisfactory invariant system is sufficient to prove that a CFA is safe.

2.3 Horn clauses

The problem of reasoning about the safety of a CFA through invariant systems lends itself to be stated using Horn-clauses.

The following definitions are based on similar definitions in [2, 3, 5].

Definition 19 (Horn clause). A Horn clause is a clause in which at most one of the literals are positive, and the others are negated. It can have one of the following three forms:

$$\forall X.(\neg\beta_1 \vee \neg\beta_2 \vee \cdots \vee \neg\beta_n \vee \varphi) \Leftrightarrow \forall X.((\beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_n) \rightarrow \varphi)$$
$$\forall X.(\neg\beta_1 \vee \neg\beta_2 \vee \cdots \vee \neg\beta_n) \Leftrightarrow \forall X.((\beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_n) \rightarrow \bot),$$
$$\forall X.\ \varphi \Leftrightarrow \forall X.(\top \rightarrow \varphi),$$

where X is a set of variables, β_i and φ are atoms over X and for $X = \{x_1, x_2, \ldots, x_k\}$, $\forall X$ means $\forall x_1.\forall x_2.\cdots\forall x_k$. ▪

From now on we will use the implication form of Horn clauses.

Definition 20 (Constrained Horn clause). A constrained Horn clause (CHC) is a Horn clause of one of three forms:

$$\forall X.(B(X) \wedge (\beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_n) \rightarrow H(X)), \tag{2.4}$$
$$\forall X.(B(X) \wedge (\beta_1 \wedge \beta_2 \wedge \cdots \wedge \beta_n) \rightarrow \bot), \tag{2.5}$$
$$\forall X.(\top \rightarrow H(X)), \tag{2.6}$$

where

- X is a set of variables,
- $B(X)$[3] and $H(X)$ are uninterpreted predicates applied to (a subset of) X, and
- β_i are interpreted formulae over X.

Equation 2.4 is called an inductive clause, Equation 2.5 is called a query, and Equation 2.6 is called a fact. ▪

[3] Other definitions of constrained Horn clauses, e.g. in [2] allow multiple uninterpreted predicates on their left side, but for the purposes of verifying control flow automata, one is enough.

The decision problem for a set of constrained Horn clauses is whether there is an interpretation of the uninterpreted predicate symbols that satisfies all of the clauses.

The task of searching for satisfactory invariant systems for a set of structural nodes N in a CFA $\mathcal{A}$ can be stated as a task of searching for an interpretation for a set of CHCs.

To every structural node $\ell \in N$, we assign a predicate symbol L_ℓ. Similarly to invariant systems, $L_{\ell_s} \doteq \top$ and $L_{\ell_e} \doteq \bot$.

From every structural edge $\Phi \in \mathrm{SE}(\mathcal{A}, N)$, we derive a CHC

$$\forall X^{(0..\mathrm{len}(\Phi))}.\Big(\Big(L_{\mathrm{src}(\Phi)}(X) \wedge \delta_\Phi\Big) \rightarrow L_{\mathrm{tgt}(\Phi)}\Big(X^{(\mathrm{len}(\Phi))}\Big)\Big),$$

where $X^{(0..k)} = \bigcup_{i=0}^{k} X^{(i)}$.

In this representation, the requirements of $\lambda[\ell]$ for λ to be satisfactory are the same as the requirements of the interpretation of L_ℓ.

This means that we can use CHC solving algorithms such as [3, 6] for CFA verification.

Example 9. *In the CFA $\mathcal{A}$ defined in Example 4 and depicted in Figure 2.1, the task of searching for a satisfactory invariant system for the set of structural nodes $N_1 = \{\ell_3\}$ defined in Example 7 corresponds to the task of searching for an interpretation for the following CHCs:*

$$\begin{aligned}
&\forall X.(\top \wedge \delta_{\Phi_1} \rightarrow L_{\ell_3}(X''))\\
&\forall X.\Big(L_{\ell_3}(X) \wedge \delta_{\Phi_2} \rightarrow L_{\ell_3}\Big(X^{(4)}\Big)\Big)\\
&\forall X.\Big(L_{\ell_3}(X) \wedge \delta_{\Phi_3} \rightarrow L_{\ell_3}\Big(X^{(4)}\Big)\Big)\\
&\forall X.(L_{\ell_3}(X) \wedge \delta_{\Phi_4} \rightarrow \bot)
\end{aligned}$$

2.4 Constraints

When searching for invariants, we consolidate the information learned about them in *constraints.* Constraints are based on *implication counterexamples* in [3].

Definition 21 (Constraint). A constraint in a CFA $\mathcal{A}$ with structural nodes N is a tuple $(\alpha_0, \Phi, \alpha_k, \beta)$ where

- $\Phi \in \mathrm{SE}(\mathcal{A}, N)$ is a structural edge of length k,
- $\alpha_0 : \Delta_\Phi \nrightarrow \mathbb{Z}$ and $\alpha_k : \Delta_\Phi \nrightarrow \mathbb{Z}$ are valuations over the set of variables that might change upon execution of Φ,
- $\beta : \Big(\overline{\Delta_\Phi}\Big) \nrightarrow \mathbb{Z}$ is a valuation over the set of variables that do not change upon execution of Φ and
- for every $\hat{\alpha}_0 \subseteq \alpha_0$, $\hat{\alpha}_k \subseteq \alpha_k$ and $\hat{\beta} \subseteq \beta$, where $\hat{\alpha}_0$ and $\hat{\alpha}_k$ are full valuations over Δ_Φ, and $\hat{\beta}$ is a full valuation over $\overline{\Delta_\Phi}$, there are full valuations $\alpha_1, \alpha_2, \ldots, \alpha_{k-1} : X \rightarrow \mathbb{Z}$ such that

$$\Big(\hat{\beta} \oplus \hat{\alpha}_0 \oplus (\alpha_1)^{(1)} \oplus (\alpha_2)^{(2)} \oplus \cdots \oplus (\alpha_{k-1})^{(k-1)} \oplus (\hat{\alpha}_k)^{(k)} \oplus \hat{\beta}^{(k)}\Big) \models \delta_\Phi. \quad (2.7)$$

▪

Example 10. *Assume that in a CFA where the set of variables is* $\{x, y, z\}$, *there is structural edge* Φ *and that the statements along it are* $[x < y], (x := x + 1), (\texttt{havoc}\ z)$. *Then* $\Delta_\Phi = \{x, z\}$, $\overline{\Delta_\Phi} = \{y\}$, *and*

$$\begin{aligned}\delta_\Phi \Leftrightarrow & (x < y) \land (x' = x) \land (y' = y) \land (z' = z) \\ & \land (x'' = x' + 1) \land (y'' = y') \land (z'' = z') \\ & \land (x''' = x'') \land (y''' = y'').\end{aligned}$$

Consider the tuple $(\alpha_0, \Phi, \alpha_3, \beta)$, *where* $\alpha_0 = \{x \to 2\}$, $\alpha_3 = \{x \to 3\}$ *and* $\beta = \{y \to 4\}$, *is a constraint. Full valuations over* $\{x, z\}$ *that are subsets of* α_0 *or* α_3 *are of the form* $\hat{\alpha_0} = \{x \to 2, z \to \mathbf{a}\}$ *or* $\hat{\alpha_3} = \{x \to 3, z \to \mathbf{b}\}$ *respectively, where* $\mathbf{a}$ *and* $\mathbf{b}$ *are chosen arbitrarily. The valuation* β *is already full over* $\{y\}$, *its single full subset is itself.*

For every choice of $\mathbf{a}$ *and* $\mathbf{b}$, *the valuations*

$$\begin{aligned}\alpha_1 &= \{x \to 2, y \to 4, z \to \mathbf{a}\} \text{ and} \\ \alpha_2 &= \{x \to 3, y \to 4, z \to \mathbf{a}\}\end{aligned}$$

satisfy the requirement that

$$\left(\beta \oplus \hat{\alpha}_0 \oplus (\alpha_1)' \oplus (\alpha_2)'' \oplus (\hat{\alpha}_3)''' \oplus \beta'''\right) \models \delta_\Phi.$$

Therefore, $(\alpha_0, \Phi, \alpha_3, \beta)$ *is a constraint in this CFA.*

Example 11. *In the CFA* $\mathcal{A}$ *defined in Example 4 and depicted in Figure 2.1, for the set of structural nodes* $N_1 = \{\ell_3\}$ *defined in Example 7 the following are all constraints:*

$$\begin{gathered}(\{\}, \Phi_1, \{u \to 32, z \to 56\}, \{x \to 32, y \to 56\}), \\ (\{u \to 43, z \to -52\}, \Phi_2, \{u \to 44, z \to -51\}, \{x \to 1, y \to 100\}), \\ (\{u \to 43, z \to -52\}, \Phi_3, \{u \to 42, z \to -52\}, \{x \to 21, y \to 18\}), \\ (\{\}, \Phi_4, \{\}, \{x \to 21, y \to 18, z \to -1000, u \to 18\}).\end{gathered}$$

Since the valuations in a constraint are not necessarily full, a constraint can be viewed as a template which can be filled in by choosing values for some variables the valuations do not assign a value to.

Corollary 1. If $(\alpha_0, \Phi, \alpha_k, \beta)$ is a constraint in a CFA, then for every $\tilde{\alpha}_0 \subseteq \alpha_0$, $\tilde{\alpha}_k \subseteq \alpha_k$ and $\tilde{\beta} \subseteq \beta$, where $\mathcal{X}_{\tilde{\alpha}_0} \subseteq \Delta_\Phi$, $\mathcal{X}_{\tilde{\alpha}_k} \subseteq \Delta_\Phi$ and $\mathcal{X}_{\tilde{\beta}} \subseteq \overline{\Delta_\Phi}$, $\left(\tilde{\alpha}_0, \Phi, \tilde{\alpha}_k, \tilde{\beta}\right)$ is also a constraint. ▪

By Corollary 1, some constraints imply the existence of many. A set of constraints can therefore imply the existence of a larger set.

Definition 22 (Constraint system). The constraint system generated by a set of constraints $\mathbf{C}$ is the set of constraints $\mathrm{CS}(\mathbf{C})$, where $C' \in \mathrm{CS}(\mathbf{C})$ if and only if there is a $C \in \mathbf{C}$ such that by Corollary 1, the existence of C implies the existence of C'.

We say that a set of constraints $\mathcal{C}$ is a constraint system if $\mathrm{CS}(\mathcal{C}) = \mathcal{C}$. ▪

Constraints convey information about the CFA under examination. This information can be used when searching for invariants. Let Φ be a structural edge leading from ℓ_0 to ℓ_k A constraint $(\alpha_0, \Phi, \alpha_k, \beta)$ poses a requirement that every invariant system must adhere to in order to be satisfactory. The following sentence summarizes the requirement for an invariant system λ.

If $(\alpha_0 \oplus \beta) \models \lambda[\ell_0]$, then $\lambda[\ell_k]$ must be chosen such that $(\alpha_k \oplus \beta) \models \lambda[\ell_k]$.

Proposition 7. If $(\alpha_0, \Phi, \alpha_k, \beta)$ is a constraint and λ is a satisfactory invariant system in a CFA, then if $(\alpha_0 \oplus \beta) \models \lambda[\mathrm{src}(\Phi)]$, then $(\alpha_k \oplus \beta) \models \lambda[\mathrm{tgt}(\Phi)]$. ▪

Proof. Assume indirectly that there is a constraint $(\alpha_0, \Phi, \alpha_k, \beta)$ and a satisfactory invariant system λ in a CFA, where $(\alpha_0 \oplus \beta) \models \lambda[\mathrm{src}(\Phi)]$, but $(\alpha_k \oplus \beta) \not\models \lambda[\mathrm{tgt}(\Phi)]$.

Let $k = \mathrm{len}(\Phi)$, $\ell_0 = \mathrm{src}(\Phi)$ and $\ell_k = \mathrm{tgt}(\Phi)$.

Since λ is satisfactory and Φ is a structural edge, the formula

$$\lambda[\ell_0] \wedge \delta_\Phi \wedge \neg(\lambda[\ell_k])^{(k)} \tag{2.8}$$

is unsatisfiable.

Since $(\alpha_k \oplus \beta) \not\models \lambda[\ell_k]$, there is a full valuation $\gamma \subseteq (\alpha_k \oplus \beta)$ over X, for which $\gamma \models \neg\lambda[\ell_k]$. Let $\hat{\alpha}_k = \gamma[\Delta_\Phi]$, let $\hat{\beta} = \gamma\left[\overline{\Delta_\Phi}\right]$, and let $\hat{\alpha}_0 \subseteq \alpha_0$ be a full valuation over Δ_Φ.

By Definition 21, there are full valuations $\alpha_1, \alpha_2, \ldots, \alpha_{k-1} : X \to \mathbb{Z}$ such that

$$\left(\hat{\beta} \oplus \hat{\alpha}_0 \oplus (\alpha_1)^{(1)} \oplus (\alpha_2)^{(2)} \oplus \cdots \oplus (\alpha_{k-1})^{(k-1)} \oplus (\alpha_k)^{(k)} \oplus \hat{\beta}^{(k)}\right) \models \delta_\Phi.$$

We constructed $\hat{\alpha}_0$ and $\hat{\beta}$ such that $\left(\hat{\alpha}_0 \oplus \hat{\beta}\right) \subseteq (\alpha_0 \oplus \beta)$. Since $(\alpha_0 \oplus \beta) \models \lambda[\ell_0]$, we conclude that $\left(\hat{\alpha}_0 \oplus \hat{\beta}\right) \models \lambda[\ell_0]$. Therefore,

$$\left(\hat{\beta} \oplus \hat{\alpha}_0 \oplus (\alpha_1)^{(1)} \oplus (\alpha_2)^{(2)} \oplus \cdots \oplus (\alpha_{k-1})^{(k-1)} \oplus (\hat{\alpha}_k)^{(k)} \oplus \hat{\beta}^{(k)}\right) \models \lambda[\ell_0] \wedge \delta_\Phi \wedge \neg(\lambda[\ell_k])^{(k)}.$$

Which means that Equation 2.8 is satisfiable, and we reached a contradiction. □

Definition 23 (Contradictory constraint system). We call a constraint system $\mathcal{C}$ in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ contradictory if there is a chain

$$(\alpha_0, \Phi_1, \alpha_1, \beta_1), (\tilde{\alpha}_1, \Phi_2, \alpha_2, \beta_2), (\tilde{\alpha}_2, \Phi_3, \alpha_3, \beta_3), \ldots, (\tilde{\alpha}_{n-1}, \Phi_n, \alpha_n, \beta_n) \in \mathcal{C}$$

such that

- Φ_1 starts at ℓ_s,
- for all $i \in [1..n-1]$, $\mathrm{tgt}(\Phi_i) = \mathrm{src}(\Phi_{i+1})$,
- Φ_n leads to ℓ_e and
- for all $i \in [1..n-1]$, $(\tilde{\alpha}_i \oplus \beta_{i+1}) \subseteq (\alpha_i \oplus \beta_i)$. ▪

Proposition 8. If there is a contradictory constraint system in a CFA, then there is a feasible error path. ▪

Proof. Let there be a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ with a contradictory constraint system. Let the chain making the set contradictory be

$$(\alpha_0, \Phi_1, \alpha_1, \beta_1), (\tilde{\alpha}_1, \Phi_2, \alpha_2, \beta_2), (\tilde{\alpha}_2, \Phi_3, \alpha_3, \beta_3), \ldots, (\tilde{\alpha}_{n-1}, \Phi_n, \alpha_n, \beta_n).$$

Let Φ be the concatenation of $\Phi_1, \Phi_2, \ldots, \Phi_n$. Then Φ is an error path, since Φ_1 starts at ℓ_s and Φ_n leads to ℓ_e. We will show that it is feasible.

Let $k_i = \text{len}(\Phi_i)$ for all $i \in [1..n]$.

We will create full valuations from the valuations in the constraints such that they still form a chain. Let ζ_n be a full valuation over X such that $\zeta_n \subseteq (\alpha_n \oplus \beta_n)$. Then, for all $i \in [1..n-1]$, let ζ_i be a full valuation over X such that

$$\zeta_i \subseteq \left(\tilde{\alpha}_i \oplus \zeta_{i+1}\left[\overline{\Delta_{\Phi_{i+1}}}\right]\right) \subseteq (\alpha_i \oplus \beta_i).$$

We can create them in reverse order: ζ_n first, then ζ_{n-1}, ζ_{n-2} and so on. Finally, let ζ_0 be a full valuation over X such that $\zeta_0 \subseteq \left(\alpha_0 \oplus \zeta_1\left[\overline{\Delta_{\Phi_1}}\right]\right)$.

Then for all $i \in [1..n]$,

- $\zeta_{i-1}[\Delta_{\Phi_i}] = \zeta_i[\Delta_{\Phi_i}]$,
- by Corollary 1, $\left(\zeta_{i-1}[\Delta_{\Phi_i}], \Phi_i, \zeta_i[\Delta_{\Phi_i}], \zeta_i\left[\overline{\Delta_{\Phi_i}}\right]\right)$ is a constraint, and
- by Definition 21, there are full valuations[4] $\alpha_{i_1}, \alpha_{i_2}, \ldots, \alpha_{i_{k_i-1}} : X \to \mathbb{Z}$, such that

$$\left(\zeta_{i-1} \oplus \gamma_i \oplus (\zeta_i)^{k_i}\right) \models \delta_{\Phi_i},$$
$$\text{where } \gamma_i = (\alpha_{i_1})' \oplus (\alpha_{i_2})'' \oplus \cdots \oplus \left(\alpha_{i_{k_i-1}}\right)^{(k_i-1)}.$$

For $i \in [1..n-1]$, let l_i be the number of edges in Φ before Φ_i starts, i.e., $l_i = \sum_{j=1}^{i-1} k_j$. Let $l_n = \sum_{i=1}^{n} k_i$, the length of Φ.

Φ is feasible because

$$\left(\bigoplus_{i=1}^{n} (\zeta_{i-1} \oplus \gamma_i)^{(l_i)}\right) \oplus (\zeta_n)^{(l_n)} \models \delta_\Phi.$$

□

[4] Since ζ_{i-1} and ζ_i are full valuations, we can substitute $\zeta_{i-1}[\Delta_{\Phi_i}]$ in place of $\hat{\alpha}_0$, $\zeta_i[\Delta_{\Phi_i}]$ in place of $\hat{\alpha}_k$ and $\zeta_i\left[\overline{\Delta_{\Phi_i}}\right]$ in place of $\hat{\beta}$ into Equation 2.7.

Chapter 3

The Algorithm

3.1 Overview

The goal of the algorithm is to prove that in a control flow automaton $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ the error state ℓ_e is unreachable. It first finds a set of structural nodes N (Section 3.2), then searches for a satisfactory invariant system λ for it.

The search for invariants is an iterative collaboration between *teacher* (Section 3.3) and *learner* (Section 3.4) modules. Learners suggest candidate invariant systems, and teachers then check if they are satisfactory. If one of the learners finds a satisfactory invariant system, then by Proposition 6, the error state is proven unreachable. Otherwise, the teacher gives constraints to the learners which highlight the reason for the failure of the candidate system and provide information about the checked CFA. The learners then suggest candidates which adhere to the constraints, if possible. If the constraints generate a contradictory constraint system, then by Proposition 8, the error state is proven reachable.

Figure 3.1 shows an overview of the algorithm architecture.

3.2 Finding structural nodes

In order to find a set of structural nodes, we have to find loops and choose locations until at least one location is chosen from every loop. Instead of searching for loops, we will search for *strongly connected components*.

Definition 24 (Strongly connected component). In a directed graph a strongly connected component (SCC) is a maximal subgraph in which there is a path from every vertex to every other vertex. ▪

Corollary 2. Every loop in the graph is either a strongly connected component or can be extended to be a strongly connected component. Every strongly connected component has a loop in it. ▪

A CFA is structured as a directed graph where the vertices are the CFA locations and the edges are the CFA edges without the statements. We call this graph the control flow graph (CFG). Using Tarjan's algorithm [9] we can find the strongly connected components in it. There may be many loops in every strongly connected component, therefore we choose a vertex from each of them, remove it along with its edges, and see if there are strongly connected components in the remaining vertices. We describe the process in Algorithm 2.

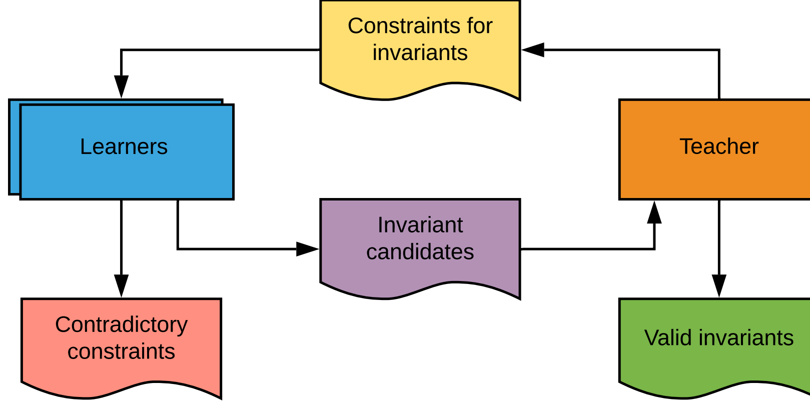

Figure 3.1: Overview of the algorithm architecture

Algorithm 2: Find a set of structural nodes

Input: $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, a control flow automaton
Output: N, a set of structural nodes

```
1  N ← ∅;
2  sccs ← the SCCs Tarjan's algorithm finds in the CFG as sets of vertices;
3  repeat
4      scc ← an element of sccs;
5      sccs ← sccs \ scc;
6      ℓ ← a vertex (location) in scc;
7      if there is an edge leading from ℓ to itself or another vertex in scc then
8          N ← N ∪ {ℓ};
9          newSCCs ← the SCCs Tarjan's algorithm finds in the subgraph induced by
            scc \ {ℓ};
10         sccs ← sccs ∪ newSCCs;
11     else
           // scc = {ℓ} and ℓ does not have a loop edge
12     end
13 until sccs = ∅;
14 return N;
```

Proposition 9. The set of locations that Algorithm 2 returns is a set of structural nodes by Definition 15. ▪

Proof. Assume indirectly that there is a loop in $L \setminus N$. By Corollary 2, the vertices of the loop are in a strongly connected component. At line 2, Tarjan's algorithm adds that to `sccs`. When it gets chosen at line 4, two cases are possible. If the ℓ chosen at line 6 is one of the locations in the loop, the algorithm adds ℓ to N at line 8, and we reach a contradiction. Otherwise, every location in the loop is in $\texttt{scc} \setminus \{\ell\}$, therefore at line 9, Tarjan's algorithm adds a set with all of them in it to `sccs`. When that set is chosen at line 4, the same reasoning can be repeated until one of the locations in the loop gets chosen. □

3.3 Teacher

The teacher module receives a candidate invariant system from one of the learners and checks if it is satisfactory. If it is not, then it produces a set of constraints that highlight the problems with the candidate system.

3.3.1 Solver

By Definition 18, the teacher has to check the satisfiability of formulae like Equation 2.1 to decide if an invariant system is satisfactory. Our algorithm relies on an SMT solver to achieve that. The solver can check the satisfiability of a formula and give a valuation that satisfies it.

We defined the transition formula of a statement (δ_s) such that it describes the relation between variables in X and in X'. We introduce x' for each $x \in X$, even those whose value does not change. To make sure that the new value is the same as the old for the variables that s does not change, we add $\text{same}\left(\overline{\Delta_s}\right)$. We chose this notation because we find it more intuitive because every variable at every point has the same number of primes applied.

The solver, however, does not work like this. Using our notation, for all $x \in \overline{\Delta_s}$, it treats x and x' as the same variable. By doing so, it can give more useful valuations as the following example illustrates.

Example 12. *If the set of variables is $X = \{x, y, z\}$, the transition formula of the statement $s = (x \coloneqq x + y)$ is*

$$\delta_s \Leftrightarrow (x' = x + y) \wedge (y' = y) \wedge (z' = z).$$

In a valuation $\alpha : (X \cup X') \nrightarrow \mathbb{Z}$, such that $\alpha \models \delta_s$, the value of z and z' can be anything, but their value must be the same. If α assigns value to only one of z and z', or if it assigns value to neither, then there is a valuation $\hat{\alpha} \subseteq \alpha$ that assigns different values to them. Since $\hat{\alpha} \models \neg\delta_s$, $\alpha \not\models \delta_s$. Therefore, α must assign a value to both z and z', even though the number it chooses for that value is irrelevant.

At the same time, α must also assign the same value to y and y', but this value cannot be chosen arbitrarily, because it is related to the value of x and x' through $x' = x + y$. It is useful to know when we can choose arbitrary values, and when we cannot. Thankfully, the solver can tell us that.

The solver, in this case, would treat y and y' as the same variable, and it would also treat z and z' as the same variable (but different from y and y'). It would assign a value to y and y' (to make sure that their relation with x and x' is satisfied), but it would not assign a value to z and expect us to choose the same value for both z and z'. It works similarly for longer statement sequences in paths.

We will describe our expectation of the solver more formally by stating our requirements for its two functions, $\mathrm{SAT}(\Phi, \lambda)$ and $\mathrm{VAL}(\Phi, \lambda)$. For an invariant system λ and a structural edge

$$\Phi = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2) \ldots (\ell_{k-1}, s_k, \ell_k),$$

(R1) $\mathrm{SAT}(\Phi, \lambda)$ is true if and only if the formula $\lambda[\mathrm{src}(\Phi)] \wedge \delta_\Phi \wedge (\lambda[\mathrm{tgt}(\Phi)])^{(\mathrm{len}(\Phi))}$ is satisfiable, and

(R2) $\mathrm{VAL}(\Phi, \lambda) : \left(X \cup X' \cup \cdots \cup X^{(\mathrm{len}(\Phi))}\right) \nrightarrow \mathbb{Z}$ is a valuation such that

(R2.1) for all $i \in [1..\mathrm{len}(\Phi)]$, for all $x \in \overline{\Delta_{s_i}}$, if $\mathrm{VAL}(\Phi, \lambda)$ assigns value to either $x^{(i-1)}$ or $x^{(i)}$, then it assigns the same value to the other, and

(R2.2) $\mathrm{VAL}(\Phi, \lambda) \models \left(\mathrm{sames}(\Phi) \rightarrow \left(\lambda[\mathrm{src}(\Phi)] \wedge \delta_\Phi \wedge (\lambda[\mathrm{tgt}(\Phi)])^{(\mathrm{len}(\Phi))}\right)\right)$,

where

$$\mathrm{sames}(\Phi) \Leftrightarrow \bigwedge_{i=1}^{k} \left(\mathrm{same}\left(\overline{\Delta_{s_i}}\right)\right)^{(i-1)}.$$

Corollary 3. By R2.1, $\mathrm{VAL}(\Phi, \lambda) \not\models \neg\mathrm{sames}(\Phi)$, which means that there is a valuation $\theta \subseteq \mathrm{VAL}(\Phi, \lambda)$ such that $\theta \models \mathrm{sames}(\Phi)$, and therefore by R2.2,

$$\theta \models \lambda[\mathrm{src}(\Phi)] \wedge \delta_\Phi \wedge (\lambda[\mathrm{tgt}(\Phi)])^{(\mathrm{len}(\Phi))}.$$

We also expect $\mathrm{VAL}(\Phi, \lambda)$ to be minimal in the sense that we cannot omit any of the variable assignments in it without violating either R2.1 or R2.2. This, however, is not a requirement, the algorithm is sound even if the solver does not adhere to this.

3.3.2 Teacher algorithm

Using a solver whose implementation satisfies our requirements, the teacher works as described in Algorithm 3.

Proposition 10. If Algorithm 3 returns an empty set, the checked invariant system λ is satisfactory.

Proof. If $\mathbf{C}$ is empty at line 13, then the algorithm never executed line 10, meaning $\mathrm{SAT}(\Phi, \lambda)$ was always false at line 4. Therefore, by R1, for all $\Phi \in \mathrm{SE}(\mathcal{A}, N)$, the formula

$$\lambda[\mathrm{src}(\Phi)] \wedge \delta_\Phi \wedge \neg(\lambda[\mathrm{tgt}(\Phi)])^{(\mathrm{len}(\Phi))}$$

is unsatisfiable, and by Definition 18, λ is satisfactory. □

Proposition 11. Every element of the set Algorithm 3 returns is a constraint in the CFA $\mathcal{A}$ for the structural nodes N.

Algorithm 3: Check if an invariant system is satisfactory

Input: The set of structural edges $\mathrm{SE}(\mathcal{A}, N)$ for a set of structural nodes N in a CFA $\mathcal{A}$

Input: An invariant system λ for N

Output: A set of constraints $\mathbf{C}$ or an empty set if λ is satisfactory

```
1  C ← ∅;
2  forall Φ ∈ SE(A, N) do
3  |  k ← len(Φ);
4  |  if SAT(Φ, λ) then
5  |  |  γ ← VAL(Φ, λ);                       // γ : (X ∪ X′ ∪ ··· ∪ X^(k)) ⇸ ℤ
   |  |  /* For a constraint we only need the values γ assigns to
   |  |     variables in X and in X^(k).                                  */
6  |  |  α_0 ← γ[Δ_Φ];                                   // α_0 : Δ_Φ ⇸ ℤ
7  |  |  α̇_k ← γ[(Δ_Φ)^(k)];                        // α̇_k : (Δ_Φ)^(k) ⇸ ℤ
8  |  |  Let α_k be such that (𝒳_{α_k})^(k) = 𝒳_{α̇_k} and ∀x ∈ 𝒳_{α_k} α_k(x) = α̇_k(x^(k));
   |  |  // α_k : (Δ_Φ) ⇸ ℤ
9  |  |  β ← γ[overline(Δ_Φ)];                     // β : overline(Δ_Φ) ⇸ ℤ
   |  |  // By R2.1, ∀x ∈ 𝒳_β β(x) = γ(x) = γ(x′) = ··· = γ(x^(k))
10 |  |  C ← C ∪ (α_0, Φ, β, α_k);
11 |  end
12 end
13 return C;                                   // Empty if λ is satisfactory.
```

Proof. The algorithm returns $\mathbf{C}$, and it only adds elements to it at line 10. Therefore, we have to show that the tuple we add in that line is a constraint.

At line 3, let

$$\Phi = (\ell_0, s_1, \ell_1)(\ell_1, s_2, \ell_2)\ldots(\ell_{k-1}, s_k, \ell_k).$$

At line 5, γ satisfies R2.1 and R2.2.

Let $\gamma_0, \gamma_1, \ldots, \gamma_k : \Delta_\Phi \nrightarrow \mathbb{Z}$ and $\zeta : \overline{\Delta_\Phi} \nrightarrow \mathbb{Z}$ be valuations such that

$$\gamma = \left(\gamma_0 \oplus \zeta \oplus (\gamma_1)' \oplus \zeta' \oplus \cdots \oplus (\gamma_k)^{(k)} \oplus \zeta^{(k)}\right).$$

This division is possible because of R2.1.

At line 10, $\alpha_0 = \gamma_0$, $\alpha_k = \gamma_k$ and $\beta = \zeta$. We will show that for each triple of full valuations $\hat{\alpha}_0 : \Delta_\Phi \to \mathbb{Z}$, $\hat{\alpha}_k : \Delta_\Phi \to \mathbb{Z}$ and $\hat{\beta} : \overline{\Delta_\Phi} \to \mathbb{Z}$ such that $\hat{\alpha}_0 \subseteq \alpha_0$, $\hat{\alpha}_k \subseteq \alpha_k$ and $\hat{\beta} \subseteq \beta$, there are full valuations $\alpha_1, \alpha_2, \ldots, \alpha_{k-1} : X \to \mathbb{Z}$ for which

$$\left(\hat{\beta} \oplus \hat{\alpha}_0 \oplus (\alpha_1)^{(1)} \oplus (\alpha_2)^{(2)} \oplus \cdots \oplus (\alpha_{k-1})^{(k-1)} \oplus (\hat{\alpha}_k)^{(k)} \oplus \hat{\beta}^{(k)}\right) \models \delta_\Phi.$$

Let $\hat{\gamma}_0 = \hat{\alpha}_0$ and $\hat{\gamma}_k = \hat{\alpha}_k$. For all $i \in [1..k-i]$, let $\hat{\gamma}_i : \Delta_\Phi \to \mathbb{Z}$ be a full valuation such that

- $\hat{\gamma}_i \subseteq \gamma_i$,
- $\hat{\gamma}_i$ assigns the same value to every $x \in \overline{\Delta_{s_i}}$ as $\hat{\gamma}_{i-1}$,
- $\hat{\gamma}_i$ assigns the same value to every $x \in \overline{\Delta_{s_{i+1}}}$ as $\hat{\gamma}_{i+1}$.

This is possible because γ adheres to R2.1. Then to get $\hat{\alpha}_0$ and $\hat{\alpha}_k$, we assign value to new variables. We can propagate the new variable assignments as required. These propagations will not overlap and cause potential contradictions (assigning different values to the same variable), since they are valuations over Δ_Φ. If any of the valuations we get are not full yet, we can keep choosing a value for a non-assigned variable and propagating it until they are all full.

Let

$$\hat{\gamma} = \hat{\alpha}_0 \oplus \hat{\beta} \oplus (\hat{\gamma}_1)' \oplus \hat{\beta}' \oplus (\hat{\gamma}_2)'' \oplus \hat{\beta}'' \oplus \cdots \oplus (\hat{\alpha}_k)^{(k)} \oplus \hat{\beta}^{(k)}.$$

Then $\hat{\gamma} \subseteq \gamma$, and $\hat{\gamma} \models \text{sames}(\Phi)$. Therefore, since γ satisfies R2.2, $\hat{\gamma} \models \delta_\Phi$. □

3.4 Learners

Learners synthesize invariant systems that adhere to a set of constraints they receive from the teacher. They also have to generalize the constraints in order to eventually synthesize a satisfactory invariant system.

We can view invariant systems as classifications. An invariant system λ classifies the potential configurations (ℓ, α) of the CFA (where ℓ is a structural node) based on whether $\alpha \models \lambda[\ell]$. We solve the problem of synthesizing invariant systems by searching for classifications. The properties required for an invariant system to adhere to a set of constraints—or even to be satisfactory—can be stated in terms of configurations and their classification. However, in order to prevent having to consider the infinite set of configurations, we only aim to find a classification of a set of *datapoints* adheres to the constraints. We then generalize that classification to an invariant system.

3.4.1 Datapoints

The definition of datapoints is based on [3].

Definition 25 (Datapoint). In a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ for a set of structural nodes $N \subseteq L$, a datapoint is a pair (ℓ, α), where $\ell \in N \cup \{\ell_s, \ell_e\}$, and $\alpha : X \nrightarrow \mathbb{Z}$ is a (potentially partial) valuation.

For datapoints $D_1 = (\ell_1, \alpha_1)$ and $D_2 = (\ell_2, \alpha_2)$, we say that

- $D_1 \subseteq D_2$ if $\ell_1 = \ell_2$ and $\alpha_1 \subseteq \alpha_2$;
- D_1 and D_2 are disjoint if $\ell_1 \neq \ell_2$ or α_1 and α_2 are disjoint;
- if D_1 and D_2 are not disjoint, $D_1 \cap D_2 = (\ell_1, \alpha_1 \oplus \alpha_2)$. ▪

Our definition differs from that in [3] in that we allow the valuation in the datapoint to not be full. This allows a datapoint to potentially represent a set of multiple CFA configurations instead of a single one.

Invariant candidates can be viewed as a classification of datapoints, and constraints can be viewed as restrictions on how they can be classified. However, since we allow datapoints to represent multiple configurations, the classification is not always unequivocal.

Definition 26 (Datapoint classification). An invariant system λ classifies (or labels) the datapoint (ℓ, α) as

- `true` if $\alpha \models \lambda[\ell]$,
- `false` if $\alpha \models \neg\lambda[\ell]$,
- `indeterminate` otherwise. ▪

Since every invariant system by definition assigns $\top$ to the initial location of the CFA, datapoints with the initial location are always classified as `true`. Similarly, datapoints with the error location are always classified as `false`.

Example 13. *Let* $D_1 = (\ell, \{x \to 2, y \to 3\})$, $D_2 = (\ell, \{y \to 4\})$ *and* $D_3 = (\ell, \{x \to 2\})$ *be datapoints.*

They assign different values to y*, therefore* D_1 *and* D_2 *are disjoint. However,* D_2 *and* D_3 *are not disjoint and* $D_2 \cap D_3 = \{x \to 4, y \to 5\}$*. Moreover,* $D_1 \subseteq D_3$*.*

Let λ *be an invariant system for which* $\lambda[\ell] \Leftrightarrow (y \leq 3)$*. The invariant system* λ *classifies* D_1 *as* `true`*,* D_2 *as* `false` *and* D_3 *as* `indeterminate`*.*

Proposition 12. If an invariant system λ classifies the datapoint D as `true` or `false`, then λ classifies every $\hat{D} \subseteq D$ the same.

If λ classifies the datapoint D as `indeterminate`, then there is a $\hat{D}_1 \subseteq D$ that it classifies `true` and a $\hat{D}_2 \subseteq D$ that it classifies `false`. ▪

Proof. If λ classifies the datapoint $D = (\ell, \alpha)$ as `true`, then $\alpha \models \lambda[\ell]$. Let $\hat{D} = (\ell, \hat{\alpha})$ be a datapoint such that $\hat{D} \subseteq D$. By Proposition 1, since $\hat{\alpha} \subseteq \alpha$, $\hat{\alpha} \models \lambda[\ell]$, and λ classifies $\hat{D}$ as `true`.

The same argument can be made for $\neg\lambda[\ell]$ when D is classified as `false`.

If λ classifies the datapoint $D = (\ell, \alpha)$ as `indeterminate`, then $\alpha \not\models \lambda[\ell]$ and $\alpha \not\models \neg\lambda[\ell]$. By Proposition 3, we know that there are $\alpha_1 \subseteq \alpha$ and $\alpha_2 \subseteq \alpha$ such that $\alpha_1 \models \lambda[\ell]$ and $\alpha_2 \models \neg\lambda[\ell]$. Therefore, the datapoint $\hat{D}_1 = (\ell, \alpha_1)$ is classified as `true` and the datapoint $\hat{D}_2 = (\ell, \alpha_2)$ is classified `false`. Also, $\hat{D}_1 \subseteq D$ and $\hat{D}_2 \subseteq D$. □

We originally defined constraints in terms of structural edges and valuations, but for the algorithm, we will consider a constraint a relation between datapoint pairs: if one datapoint is classified `true`, another must also be.

Definition 27 (Datapoints in a constraint). For a constraint $C = (\alpha_0, \Phi, \alpha_k, \beta)$, we say its source datapoint is $\mathrm{src}(C) = (\mathrm{src}(\Phi), \alpha_0 \oplus \beta)$, and its target datapoint is $\mathtt{tgt}(C) = (\mathrm{tgt}(\Phi), \alpha_k \oplus \beta)$. ▪

When the valuations in a constraint are not full, the constraint is a template that allows us to fill in arbitrary values for the other variables and the relationship remains as stated in Corollary 1. For datapoints, the process is slightly different. When we acquire new datapoints by filling in values in the source or the target datapoint, its pair might change as well. The reason for this is that we have to ensure that we give the same value in both datapoints to the variables that cannot change upon executing the structural edge.

The following definitions show how to find the pair of a datapoint with respect to a constraint.

Definition 28 (Positive deduction based on a constraint). For a constraint $C = (\alpha_0, \Phi, \alpha_k, \beta)$ and a datapoint $D = (\mathrm{src}(\Phi), \gamma)$ such that $D \subseteq \mathrm{src}(C)$, we define the datapoint $\mathtt{positiveDeduction}(C, D) = \left(\mathrm{tgt}(\Phi), \alpha_k \oplus \gamma\left[\overline{\Delta_\Phi}\right]\right)$ as the datapoint that C pairs with D. ▪

Definition 29 (Negative deduction based on a constraint). For a constraint $C = (\alpha_0, \Phi, \alpha_k, \beta)$ and a datapoint $D = (\mathrm{tgt}(\Phi), \gamma)$ such that $D \subseteq \mathrm{tgt}(C)$, we define the datapoint $\mathtt{negativeDeduction}(C, D) = \left(\mathrm{tgt}(\Phi), \alpha_0 \oplus \gamma\left[\overline{\Delta_\Phi}\right]\right)$ as the datapoint that C pairs with D. ▪

Proposition 13. Let there be a constraint C and an invariant system λ that is satisfactory. If λ classifies some $D_s \subseteq \mathrm{src}(C)$ as `true`, then it also classifies $\mathtt{positiveDeduction}(C, D_s)$ as `true`. ▪

Proof. Assume that $C = (\alpha_0, \Phi, \alpha_k, \beta)$, $D_s = (\mathrm{src}(\Phi), \gamma_s) \subseteq \mathrm{src}(C)$, and λ is a satisfactory invariant system that classifies D_s as `true`.

By Corollary 1, $C_1 = \left(\gamma_s[\Delta_\Phi], \Phi, \alpha_k, \gamma_s\left[\overline{\Delta_\Phi}\right]\right)$ is also a constraint. Since λ classifies D_s as `true`, $\gamma_s \models \lambda[\mathrm{src}(\Phi)]$, therefore by Proposition 7, $\left(\alpha_k \oplus \gamma_s\left[\overline{\Delta_\Phi}\right]\right) \models \lambda[\mathrm{tgt}(\Phi)]$. In other words, λ classifies $\mathtt{positiveDeduction}(C, D_1)$ as `true`. □

Proposition 14. Let there be a constraint C and an invariant system λ that is satisfactory. If λ classifies some $D_t \subseteq \mathrm{tgt}(C)$ as `false`, then it also classifies $\mathtt{negativeDeduction}(C, D_t)$ as `false`. ▪

Proof. Assume that $C = (\alpha_0, \Phi, \alpha_k, \beta)$, $D_t = (\mathrm{tgt}(\Phi), \gamma_t) \subseteq \mathrm{tgt}(C)$, and λ is a satisfactory invariant system that classifies D_t as `false`. Assume indirectly that λ does not classify $\mathtt{negativeDeduction}(C, D_t)$ as `false`. Then

$$\left(\alpha_0 \oplus \gamma_t\left[\overline{\Delta_\Phi}\right]\right) \not\models \neg\lambda[\mathrm{src}(\Phi)].$$

Which means that there is some valuation $\zeta \subseteq \left(\alpha_0 \oplus \gamma_t\left[\overline{\Delta_\Phi}\right]\right)$ for which $\zeta \models \lambda[\mathrm{src}(\Phi)]$. By Corollary 1, $C_2 = \left(\zeta[\Delta_\Phi], \Phi, \gamma_t[\Delta_\Phi], \zeta\left[\overline{\Delta_\Phi}\right]\right)$ is also a constraint. Therefore, by Proposition 7,

$$\left(\gamma_t[\Delta_\Phi] \oplus \zeta\left[\overline{\Delta_\Phi}\right]\right) \models \lambda[\mathrm{tgt}(\Phi)]. \tag{3.1}$$

Since λ classifies D_t as `false`, $\gamma_t \models \neg\lambda[\mathrm{tgt}(\Phi)]$. That contradicts Equation 3.1 because

$$\left(\gamma_t[\Delta_\Phi] \oplus \zeta\left[\overline{\Delta_\Phi}\right]\right) \subseteq \gamma_t.$$

□

3.4.1.1 Checking constraint consistency

Every time, a teacher module produces new constraints, they might reveal enough information to prove that the system is unsafe. In order to notice when that happens, we need to check if the constraint system generated by the constraints is consistent. The defining feature of a contradictory constraint system is a chain of constraints as defined in Definition 23. We will search for this chain with a method similar to a breadth-first search with datapoints being nodes and constraints being directed edges between them. We will call the datapoints reachable from a datapoint with the initial location *forced true*, and we will call those that are reachable from one with the error location *forced false*.

These concepts and the related algorithms are based on [3].

Definition 30 (Forced-true datapoints). A datapoint $D = (\ell, \alpha)$ is forced true in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ under the constraint system $\mathcal{C}$ if at least one of the following conditions holds:

(Condition 1) $\ell = \ell_s$,

(Condition 2) there is a forced-true datapoint $\hat{D}$ for which $D \subseteq \hat{D}$,

(Condition 3) there is a constraint $C \in \mathcal{C}$ and a forced-true datapoint $D_0 \subseteq \mathrm{src}(C)$ for which `positiveDeduction`$(C, D_0) = D$. ▪

Only datapoints forced true because of Condition 1 are inherently forced true, others refer to another forced-true datapoint. That datapoint may be inherently forced true or may refer to a third one which in turn may refer to fourth etc. We can trace these references from any forced-true datapoint through a number of datapoints that are forced true because of Condition 2 or Condition 3, but we would always eventually reach a datapoint forced true because of Condition 1.

Corollary 4. Every forced-true datapoint is either forced true by Condition 1 or can be traced back through transitive references via Condition 2 or Condition 3 to a datapoint that is forced true by Condition 1. ▪

Based on a constraint system, forced-true datapoints provide a simple restriction on the datapoint classification.

Proposition 15. A satisfactory invariant system classifies all forced-true datapoints `true`. ▪

Proof. Let there be a satisfactory invariant system λ and a datapoint $D = (\ell, \alpha)$ that is forced true.

By Corollary 4, we can trace D back to a datapoint with the initial location. Let the datapoints in the trace be $D_1, D_2, \ldots, D$, such that D_1 is forced true because of Condition 1.

We will prove by induction that λ classifies every datapoint in the trace as `true`. For the base case, it classifies D_1 as `true` by definition, since it has the initial location.

For the inductive step, assume that λ classifies D_i as `true`. If the next datapoint refers to D_i via Condition 2, then by Proposition 12, λ classifies it as `true`. Otherwise, the next datapoint refers to D_i via Condition 3, and by Proposition 13, λ classifies it as `true`.

Therefore, λ classifies every datapoint in the trace `true`, including D. □

When a datapoint with the error location is forced true, it is impossible to produce satisfactory invariant systems, because all invariant systems would classify it as `false`. We will prove that in this case, the CFA is unsafe.

Proposition 16. In a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$, the constraint system $\mathcal{C}$ is contradictory if and only if there is a datapoint (ℓ_e, ζ) that is forced true under $\mathcal{C}$. ▪

Proof. First we prove that if the constraint system is contradictory, then there is a datapoint (ℓ_e, ζ) that is forced true. Let the chain making $\mathcal{C}$ contradictory be

$$(\hat{\alpha}_0, \Phi_1, \alpha_1, \beta_1), (\hat{\alpha}_1, \Phi_2, \alpha_2, \beta_2), \ldots, (\hat{\alpha}_{n-1}, \Phi_n, \alpha_n, \beta_n) = C_1, C_2, \ldots, C_n$$

We will prove by induction that for all $i \in [1..n]$, $\mathrm{tgt}(C_i)$ is forced true. For the base case we will prove that $\mathrm{tgt}(C_1)$ is forced true. Since $\mathrm{src}(\Phi_1) = \ell_s$, the datapoint $\mathrm{src}(C_1) = (\ell_s, \alpha_0 \oplus \beta_1)$ is forced true by Condition 1. Therefore, $\mathrm{tgt}(C_1) = (\mathrm{tgt}(\Phi_1), \alpha_1 \oplus \beta_1)$ is forced true by Condition 3.

For the inductive step, we will assume that $\mathrm{tgt}(C_i)$ is forced true and prove that then $\mathrm{tgt}(C_{i+1})$ is forced true as well. By Definition 23, $\mathrm{tgt}(\Phi_i) = \mathrm{src}(\Phi_{i+1})$ and $(\hat{\alpha}_i \oplus \beta_{i+1}) \subseteq (\alpha_i \oplus \beta_i)$. Therefore, $\mathrm{src}(\Phi_{i+1}) \subseteq \mathrm{tgt}(\Phi_i)$, and $\mathrm{src}(\Phi_{i+1})$ is forced true by Condition 2, and $\mathrm{tgt}(C_{i+1})$ is forced true by Condition 3.

Thus, $\mathrm{tgt}(C_n)$, a datapoint whose location is ℓ_e, is forced true.

Now we will prove that if there is a datapoint (ℓ_e, ζ) forced true, then there is a contradictory chain of constraints in $\mathcal{C}$. Since $\ell_e \neq \ell_s$, (ℓ_e, ζ) cannot be forced true because of Condition 1. Therefore, by Corollary 4, it can be traced back to a datapoint (ℓ_s, γ). We will show that this sequence of references implies the existence of a contradictory chain of constraints.

We arrange the datapoints in the trace in order of references starting from (ℓ_s, γ) to (ℓ_e, ζ). Then we put the constraints from references that use Condition 3 into the chain: when a datapoint D_1 in the trace is forced true because there is a constraint C and a forced-true datapoint D_2 such that $D_1 = \texttt{positiveDeduction}(C, D_2)$, we include C in the chain. These constraints form a contradictory chain. When there are references using Condition 2 (subsets) between two references using constraints, it does not contradict with the list being a contradictory chain, since Definition 23 allows two consecutive constraints C_1 and C_2 as long as $\mathrm{tgt}(C_1) \subseteq \mathrm{src}(C_2)$, which the intermediate Condition 2 references ensure.

Thus, we have a contradictory chain of constraints and $\mathcal{C}$ is contradictory. □

In the breadth-first search, we calculate a set of forced-true datapoints under a constraint system. If there is a datapoint in the set with the error location, then by Proposition 16, the constraint system is inconsistent. The process is described in Algorithm 4.

Algorithm 4: Check constraint consistency

Input: $\mathbf{C}$, a set of constraints
Input: ℓ_s, the initial location of the CFA
Input: ℓ_e, the error location of the CFA
Output: Whether the constraint system $\mathrm{CS}(\mathbf{C})$ is consistent

```
1  D_found ← ∅;
2  forall (α_0, Φ, α_k, β) ∈ C do
3      C ← (α_0, Φ, α_k, β);
4      if src(Φ) = ℓ_s then
5          D_found ← D_found ∪ {src(C), tgt(C)};  // Condition 1 and Condition 3.
6      end
7  end
8  D_true ← D_found;
9  while D_found ≠ ∅ do
10     D_found ← ∅;
11     forall C ∈ C do
12         D_0 ← src(C);
13         D_k ← tgt(C);
14         if D_k ∉ D_true then
15             if D_0 ∈ D_true then
16                 D_found ← D_found ∪ {D_k};
17             else
18                 forall D ∈ D_true such that D and D_0 are not disjoint do
19                     D̂_0 ← D ∩ D_0;                              // Condition 2
20                     D̂_k ← positiveDeduction(C, D̂_0);            // Condition 3
21                     if D̂_k ∉ D_true then
22                         D_found ← D_found ∪ {D̂_k};
23                     end
24                 end
25             end
26         end
27     end
28     D_true ← D_true ∪ D_found;
29 end
30 if for a valuation γ, (ℓ_e, γ) ∈ D_true then
31     return false;                       // Contradictory by Proposition 16.
32 else
33     return true;                                            // Consistent.
34 end
```

Proposition 17. At line 30 of Algorithm 4, every datapoint in $\mathbf{D}_{\texttt{true}}$ is forced true under CS($\mathbf{C}$). ▪

Proof. At line 5, src(C) is forced true because of Condition 1, and tgt(C) is forced true because of Condition 3. Therefore, at line 8 we only add forced-true datapoints to $\mathbf{D}_{\texttt{true}}$.

We will prove by induction that every time, the execution gets to line 10, $\mathbf{D}_{\texttt{true}}$ contains only forced-true datapoints. For the base case, we already showed that for the initial run, that is true. For the inductive step, we will show that if the datapoints in $\mathbf{D}_{\texttt{true}}$ are forced true, then we add forced-true datapoints to $\mathbf{D}_{\texttt{found}}$ and then at line 28 to $\mathbf{D}_{\texttt{true}}$. At line 16 we simply apply Condition 3: D_0 is forced true, therefore $\texttt{positiveDeduction}(C, D_0) = D_k$ is also forced true. At line 22, $\hat{D}_0$ is forced true by Condition 2, because D is forced true, and $\hat{D}_0 = D \cap D_0 \subseteq D$. Thus, $\hat{D}_k = \texttt{positiveDeduction}\left(C, \hat{D}_0\right)$ is also forced true by Condition 3. Since we do not add datapoints to $\mathbf{D}_{\texttt{found}}$ in any other places, all datapoints we add to $\mathbf{D}_{\texttt{true}}$ at line 28 are forced true.

Therefore, we get to line 30 with only forced-true datapoints in $\mathbf{D}_{\texttt{true}}$. □

Proposition 18. Algorithm 4 returns `false` if and only if CS($\mathbf{C}$) is contradictory. ▪

Proof. If Algorithm 4 returns `false`, then there is a datapoint $(\ell_e, \gamma) \in \mathbf{D}_{\texttt{true}}$ at line 30, which by Proposition 17 is forced true under CS($\mathbf{C}$), therefore by Proposition 16, CS($\mathbf{C}$) is contradictory.

If CS($\mathbf{C}$) is contradictory, then by Definition 23, there is a contradictory chain of constraints $C_1, C_2, \ldots, C_n \in \text{CS}(\mathbf{C})$. Let $C'_i \in \mathbf{C}$ be the constraint that $C_i \in \text{CS}(\mathbf{C})$ is derived from using Corollary 1.

We will prove by induction that for all $i \in [1..n]$, a datapoint D_i for which $\text{tgt}(C_i) \subseteq D_i$ is added to $\mathbf{D}_{\texttt{true}}$. The first constraint in the chain, C_1, starts at ℓ_s, therefore when the loop at line 2 gets to C'_1, $\text{tgt}(C'_1)$ is added to $\mathbf{D}_{\texttt{found}}$ at line 5 and then to $\mathbf{D}_{\texttt{true}}$ at line 8. Trivially, $\text{tgt}(C_i) \subseteq \text{tgt}(C'_i)$.

For the inductive step, we assume that D_i is added and show that a valid D_{i+1} is also added to $\mathbf{D}_{\texttt{true}}$. Let us consider the first iteration of the main loop at line 9 after D_i is added to $\mathbf{D}_{\texttt{true}}$. When the nested loop at line 11 gets to C'_{i+1}, there are two possible courses.

If $\text{src}\left(C'_{i+1}\right) = D_i$, then the algorithm adds $D_{i+1} = \text{tgt}\left(C'_{i+1}\right)$ to $\mathbf{D}_{\texttt{found}}$ at line 16. Again, $\text{tgt}(C_{i+1}) \subseteq \text{tgt}\left(C'_{i+1}\right)$.

Otherwise, since $\text{src}(C_{i+1}) \subseteq \text{tgt}(C_i) \subseteq D_i$, the loop at line 18 does get to D_i and then

$$D_{i+1} = \texttt{positiveDeduction}\left(C'_{i+1}, \text{src}\left(C'_{i+1}\right) \cap D_i\right)$$

is added to $\mathbf{D}_{\texttt{found}}$ at line 22. Since $\text{src}(C_{i+1}) \subseteq \left(\text{src}\left(C'_{i+1}\right) \cap D_i\right)$, $\text{tgt}(C_{i+1}) \subseteq D_{i+1}$. At the end of both courses, D_{i+1} is added to $\mathbf{D}_{\texttt{true}}$ at line 28.

Therefore, by line 30, some $D_n = (\ell_e, \gamma)$ is added to $\mathbf{D}_{\texttt{true}}$ and the algorithm therefore returns `false`. □

3.4.1.2 Keeping track of datapoints

While Algorithm 4 does check if a constraint system is consistent, for learner algorithms, it is useful to extend it to find more datapoints with forced classification. Similarly to forced-true datapoints defined in Definition 30, we define forced-false datapoints.

Definition 31 (Forced-false datapoints). A datapoint $D = (\ell, \alpha)$ is forced false in a CFA $\mathcal{A} = (L, E, X, \ell_s, \ell_e)$ under the constraint system $\mathcal{C}$ if at least one of the following conditions holds:

(Condition 1) $\ell = \ell_e$,

(Condition 2) there is a forced-false datapoint $\hat{D}$ for which $D \subseteq \hat{D}$,

(Condition 3) there is a constraint $C \in \mathcal{C}$ and a forced-false datapoint $D_k \subseteq \mathrm{tgt}(C)$ for which $\texttt{negativeDeduction}(C, D_k) = D$. ▪

Similarly to Corollary 4 for forced-true datapoints, forced-false datapoints can also be traced back to one with the error location.

Corollary 5. Every forced-false datapoint is either forced true by Condition 1 or can be traced back through transitive references via Condition 2 or Condition 3 to a datapoint that is forced true by Condition 1. ▪

Similarly to Proposition 15 for forced-true datapoints, satisfactory invariant systems classify forced-false datapoints as `false`.

Proposition 19. A satisfactory invariant system classifies all forced-false datapoints `false`. ▪

Proof. Let there be a satisfactory invariant system λ and a datapoint $D = (\ell, \alpha)$ that is forced false.

By Corollary 5, we can trace D back to a datapoint with the error location. Let the datapoints in the trace be $D_1, D_2, \ldots, D$, such that D_1 is forced true because of Condition 1.

We will prove by induction that λ classifies every datapoint in the trace as `false`. For the base case, it classifies D_1 as `false` by definition, since it has the error location.

For the inductive step, assume that λ classifies D_i as `false`. If the next datapoint refers to D_i via Condition 2, then by Proposition 12, λ classifies it as `false`. Otherwise, the next datapoint refers to D_i via Condition 3, and by Proposition 14, λ classifies it as `false`. There is no other way for the trace to go to the next datapoint. ☐

Our algorithms calculate a set of forced-true and forced-false datapoints for the current constraint system. The learner algorithms then use these sets to create an invariant system consistent with the constraints.

Some of our learner algorithms do not consider the classification of every configuration of the CFA, they only consider the classification of a set of datapoints, and they only reject an invariant system if its classification of those datapoints contradicts the constraint system. If an invariant system classifies a datapoint in the sat as `indeterminate`, however, depending on which subsets of the datapoint the invariant system classifies `true` and which `false`, it may not be consistent with the constraint system.

Example 14. *If we have a single constraint*

$$C = (\{x \to 2\}, \Phi, \{x \to 3\}, \{y \to 4\}),$$

then the invariant system λ, for which

$$\lambda[\mathrm{src}(\Phi)] \Leftrightarrow (z < 4) \textit{ and}$$
$$\lambda[\mathrm{tgt}(\Phi)] \Leftrightarrow (x + z = y),$$

classifies both src(C) *and* tgt(C) *as* `indeterminate`*. When we only consider the classification of these datapoints, we do not see a contradiction. However, if we also consider the classification of*

$$D_a = (\text{src}(\Phi), \{x \to 2, y \to 4, z \to -1\}) \textit{ and}$$
$$D_b = \texttt{positiveDeduction}(C, D_a) = (\text{tgt}(\Phi), \{x \to 3, y \to 4, z \to -1\})$$

(assuming that $z \in \overline{\Delta_\Phi}$*), we can see that* λ *classifies* D_a *as* `true` *and* D_b *as* `false`*, therefore* λ *does not adhere to the constraints.*

In case of complex formulae, however, it is relatively expensive to check if a datapoint classified as `indeterminate` has such subsets. We did not find an efficient way to eliminate the problem completely, we would have to make calls to the underlying SMT solver to check if such subsets exist. It is not catastrophic, however, if a learner suggests an invariant system that contradicts the constraints in a non-obvious way. The teacher checks if the invariant system is satisfactory, and it gives more specific valuations which turn into more specific constraints and more specific datapoints for our algorithm to notice the inconsistent classification of. We therefore deemed using the SMT solver to check if an invariant system is consistent with the constraint system unnecessary.

To reduce the occurrence of such an occasion, we check the classification of a larger set of datapoints which includes the datapoints we extract from the constraints, and if D_1 and D_2 are in the set, and they are not disjoint, then we also put $D_1 \cup D_2$ in the set.

Moreover, to prevent iterating over every forced true datapoint at line 18, we keep a list of its subsets in the set of datapoints.

In Algorithm 5, we present the way we construct this data structure or update it with new datapoints. This procedure is called whenever a new datapoint emerges either from new constraints or from a deduction.

Proposition 20. If for every non-disjoint pair of datapoints $D_a, D_b \in \mathbf{D}$, $D_a \cap D_b \in \mathbf{D}$ when Algorithm 5 starts, then when the algorithm terminates, $\mathbf{D}$ still has that property. Moreover, every datapoint initially in $\mathbf{D}_{\texttt{new}}$ is added to $\mathbf{D}$. ▪

Proof. When the algorithm terminates, $\mathbf{D}_{\texttt{new}}$ is empty. We only take a datapoint out of it at line 3, and we always add that datapoint to $\mathbf{D}$ at line 17. Therefore, every datapoint either initially in $\mathbf{D}_{\texttt{new}}$ or subsequently added to $\mathbf{D}_{\texttt{new}}$ is eventually added to $\mathbf{D}$.

Let D_a and D_b two non-disjoint elements of $\mathbf{D}$ when the algorithm terminates. We will show that $D_a \cap D_b \in \mathbf{D}$.

If both $D_a \in \mathbf{D}$ and $D_b \in \mathbf{D}$ when the algorithm starts then the precondition ensures that $(D_a \cap D_b) \in \mathbf{D}$.

If one of them is in $\mathbf{D}$ when the algorithm starts, but the other is not, we assume without loss of generality that $D_a \notin \mathbf{D}$ and $D_b \in \mathbf{D}$. When $D = D_a$ at line 2, three cases are possible.

- If $D_a \subseteq D_b$, then $D_a \cap D_b = D_a$, and D_a is added to $\mathbf{D}$ at line 17.
- If $D_b \subseteq D_a$, then $D_a \cap D_b = D_b$, and D_b is already in $\mathbf{D}$.
- Otherwise, when the loop at line 6 gets to $D' = D_b$, $D_a \cap D_b$ is added to $\mathbf{D}_{\texttt{new}}$ at line 14 and eventually to $\mathbf{D}$.

Algorithm 5: Add datapoints

Data: $\mathbf{D}$, the set of datapoints. If $D_a \in \mathbf{D}$ and $D_b \in \mathbf{D}$ are non-disjoint, then $D_a \cap D_b \in \mathbf{D}$

Data: $\texttt{subsets}[D] = \{D' \in \mathbf{D} \mid D' \subseteq D\}$ for all $D \in \mathbf{D}$

Data: $\mathbf{D}_{\texttt{true}}$ the set of forced-true datapoints

Data: $\mathbf{D}_{\texttt{false}}$ the set of forced-false datapoints

Input: $\mathbf{D}_{\texttt{new}}$, the set of datapoints to add to $\mathbf{D}$ and subsets

1. **while** $\mathbf{D}_{\texttt{new}} \neq \varnothing$ **do**
2. $D \leftarrow$ some element of $\mathbf{D}_{\texttt{new}}$;
3. $\mathbf{D}_{\texttt{new}} \leftarrow \mathbf{D}_{\texttt{new}} \setminus \{D\}$;
4.   **if** $D \notin \mathbf{D}$ **then**
5. $\texttt{subsets}[D] \leftarrow \{D\}$;
6.   **forall** $D' \in \mathbf{D}$ **do**
7.   **if** $D' \subseteq D$ **then**
8. $\texttt{subsets}[D] \leftarrow \texttt{subsets}[D] \cup \{D'\}$;
9.   **else if** $D \subseteq D'$ **then**
10. $\texttt{subsets}[D'] \leftarrow \texttt{subsets}[D'] \cup \{D\}$;
11.   **if** $D' \in \mathbf{D}_{\texttt{true}}$ **then** $\mathbf{D}_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{true}} \cup \{D\}$;   `// Condition 2`
12.   **if** $D' \in \mathbf{D}_{\texttt{false}}$ **then** $\mathbf{D}_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{false}} \cup \{D\}$;   `// Condition 2`
13.   **else if** D *and* D' *are not disjoint* **then**
14. $\mathbf{D}_{\texttt{new}} \leftarrow \mathbf{D}_{\texttt{new}} \cup \{D \cap D'\}$;
15.   **end**
16.   **end**
17. $\mathbf{D} \leftarrow \mathbf{D} \cup \{D\}$;
18.   **end**
19. **end**

If both $D_a \notin \mathbf{D}$ and $D_b \notin \mathbf{D}$ when the algorithm starts, then whichever gets selected first at line 2 gets added to $\mathbf{D}$ first and when the other gets selected, one of the previous cases happens. ☐

Proposition 21. If for all $D \in \mathbf{D}$, $\texttt{subsets}[D] = \{D' \in \mathbf{D} \mid D' \subseteq D\}$ when Algorithm 5 starts, then the same is true when the algorithm terminates. ▪

Proof. Assume indirectly that for a datapoint $D \in \mathbf{D}$, $\texttt{subsets}[D] \neq \{D' \in \mathbf{D} \mid D' \subseteq D\}$ when the algorithm terminates.

1. If there is a $D' \in \texttt{subsets}[D]$ for which $D' \nsubseteq D$, then it was added at line 8 or line 10, since `subsets` is consistent before the algorithm started. However, the algorithm checks the relationship before executing either of these lines, therefore we have a contradiction.

2. If there is a $D' \in \mathbf{D}$ for which $D' \subseteq D$, but $D' \notin \texttt{subsets}[D]$, then one of the following cases applies.

   (a) If $D = D'$, then at line 5, $\texttt{subsets}[D]$ is initialized to include D'.

   (b) If initially $D \in \mathbf{D}$ and $D' \in \mathbf{D}$, then initially $D' \in \texttt{subsets}[D]$, and since the algorithm does not remove elements from `subsets`, we have a contradiction.

   (c) If initially $D \notin \mathbf{D}$ and $D' \in \mathbf{D}$, then D' is added at line 8.

   (d) If initially $D \in \mathbf{D}$ and $D' \notin \mathbf{D}$, then D' is added at line 10.

   (e) If initially $D \notin \mathbf{D}$ and $D' \notin \mathbf{D}$, then when the second one of them gets selected at line 2, the other is already in $\mathbf{D}$ and the loop at line 6 selects it eventually, and then D' is added. ☐

Corollary 6. If $\mathbf{D}_{\texttt{true}}$ only contains forced-true and $\mathbf{D}_{\texttt{false}}$ only contains forced-false datapoints when Algorithm 5 is started, then it only adds forced-true and forced-false datapoints to them respectively. ▪

Using Algorithm 5, we present two modified versions of Algorithm 4: Algorithm 6 and Algorithm 7. They are both based on [3]. They calculate the set of forced-true and the set of forced-false datapoints.

Algorithm 6: Finding forced-true datapoints

Data: $\mathbf{D}$, the set of datapoints. For every $C \in \mathbf{C}$, $\mathrm{src}(C) \in \mathbf{D}$ and $\mathrm{tgt}(C) \in \mathbf{D}$.
For every non-disjoint $D_a \in \mathbf{D}$ and $D_b \in \mathbf{D}$, $D_a \cap D_b \in \mathbf{D}$.

Data: $\texttt{subsets}[D] = \{D' \in \mathbf{D} \mid D' \subseteq D\}$ for all $D \in \mathbf{D}$

Input: $\mathbf{C}$, a set of constraints

Input: ℓ_s, the initial location of the CFA

Input: ℓ_e, the error location of the CFA

Input: $\mathbf{D}_{\texttt{prevTrue}}$, a set of previously calculated forced-true datapoints.
If $D \in \mathbf{D}_{\texttt{prevTrue}}$ then $\texttt{subsets}[D] \subseteq \mathbf{D}_{\texttt{prevTrue}}$.

Output: $\mathbf{D}_{\texttt{true}}$, the set of forced-true datapoints

1 $\mathbf{D}_{\texttt{found}} \leftarrow \mathbf{D}_{\texttt{prevTrue}}$;
2 **forall** $(\alpha_0, \Phi, \alpha_k, \beta) \in \mathbf{C}$ **do**
3     $C \leftarrow (\alpha_0, \Phi, \alpha_k, \beta)$;
4   **if** $\mathrm{src}(\Phi) = \ell_s$ **then**
5         $\mathbf{D}_{\texttt{found}} \leftarrow \mathbf{D}_{\texttt{found}} \cup \texttt{subsets}[\mathrm{src}(C)] \cup \texttt{subsets}[\mathrm{tgt}(C)]$;
6   **end**
7 **end**
8 $\mathbf{D}_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{found}}$;
9 **while** $\mathbf{D}_{\texttt{found}} \neq \varnothing$ **do**
10     $\mathbf{D}_{\texttt{found}} \leftarrow \varnothing$;
11   **forall** $C \in \mathbf{C}$ **do**
12         $D_0 \leftarrow \mathrm{src}(C)$;
13         $D_k \leftarrow \mathrm{tgt}(C)$;
14   **if** $D_k \notin \mathbf{D}_{\texttt{true}}$ **then**
15   **if** $D_0 \in \mathbf{D}_{\texttt{true}}$ **then**
16                 $\mathbf{D}_{\texttt{found}} \leftarrow \mathbf{D}_{\texttt{found}} \cup \texttt{subsets}[D_k]$;
17   **else**
18   **forall** $\hat{D}_0 \in \texttt{subsets}[D_0] \cap \mathbf{D}_{\texttt{true}}$ **do**
19                     $\hat{D}_k \leftarrow \texttt{positiveDeduction}\left(C, \hat{D}_0\right)$;
20   **if** $\hat{D}_k \notin \mathbf{D}$ **then**
21   Call Algorithm 5 with $\mathbf{D}_{\texttt{new}} = \left\{\hat{D}_k\right\}$;
22   **end**
23   **if** $\hat{D}_k \notin \mathbf{D}_{\texttt{true}}$ **then**
24                         $\mathbf{D}_{\texttt{found}} \leftarrow \mathbf{D}_{\texttt{found}} \cup \texttt{subsets}\left[\hat{D}_k\right]$;
25   **end**
26   **end**
27   **end**
28   **end**
29   **end**
30     $\mathbf{D}_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{true}} \cup \mathbf{D}_{\texttt{found}}$;
31 **end**
32 **return** $\mathbf{D}_{\texttt{true}}$;

Example 15. *Take the set of constraints* $\mathbf{C} = \{C_1, C_2, C_3\}$*, where the set of variables is* $X = \{x, y, z\}$*,*

$$\begin{aligned} C_1 &= (\{\}, \Phi_1, \{x \to 1\}, \{z \to 3\}), \\ C_2 &= (\{x \to 1\}, \Phi_2, \{\}, \{y \to 2\}), \\ C_3 &= (\{x \to 3\}, \Phi_3, \{y \to 4, z \to 2\}, \{\}), \end{aligned}$$

$\mathrm{src}(\Phi_1) = \ell_s$, $\mathrm{tgt}(\Phi_1) = \mathrm{src}(\Phi_2)$, $\mathrm{tgt}(\Phi_2) = \mathrm{src}(\Phi_1)$ *and* $\mathrm{tgt}(\Phi_3) = \ell_e$*. Elements of the set of datapoints (*$\mathbf{D}$*, as Algorithm 5 returns it) are*

$$\begin{aligned} D_1 =& (\ell_s, \{z \to 3\}) = \mathrm{src}(C_1), \\ D_2 =& (\mathrm{tgt}(\Phi_1), \{x \to 1, z \to 3\}) = \mathrm{tgt}(C_1), \\ D_3 =& (\mathrm{src}(\Phi_2), \{x \to 1, y \to 2\}) = \mathrm{src}(C_2), \\ D_2 \cap D_3 =& (\mathrm{src}(\Phi_2), \{x \to 1, y \to 2, z \to 3\}), \\ D_4 =& (\mathrm{tgt}(\Phi_2), \{y \to 2\}) = \mathrm{tgt}(C_2), \\ D_5 =& (\mathrm{src}(\Phi_3), \{x \to 3\}) = \mathrm{src}(C_3), \\ D_4 \cap D_5 =& (\mathrm{src}(\Phi_3), \{x \to 3, y \to 2\}), \\ D_6 =& (\ell_e, \{y \to 4, z \to 2\}) = \mathrm{tgt}(C_3). \end{aligned}$$

When we run Algorithm 6, and the loop at line 2 gets to C_1*, it adds* D_1*,* D_2 *and* $D_1 \cap D_2$ *to* $\mathbf{D}_{\texttt{found}}$ *at line 5.*

Then, when the loop at line 11 gets to C_2*, it will go to line 18, and that loop will find* $D_2 \cap D_3$ *as a forced-true subset of* D_3*. It will find*

$$D_7 = \texttt{positiveDeduction}(C_2, D_2 \cap D_3) = (\mathrm{tgt}(\Phi_2), \{y \to 2, z \to 3\}),$$

and run Algorithm 5 at line 21. That will lead to D_7 *and*

$$D_7 \cap D_5 = D_7 \cap (D_4 \cap D_5) = (\mathrm{src}(\Phi_3), \{x \to 3, y \to 2, z \to 3\})$$

being added to $\mathbf{D}$*. Then at line 24, the algorithm adds* D_7 *and* $D_7 \cap D_5$ *to* $\mathbf{D}_{\texttt{found}}$ *and eventually to* $\mathbf{D}_{\texttt{true}}$*.*

In the next iteration of the main loop, the algorithm finds $D_7 \cap D_5$ *as a forced-true subset of* D_5*, deduces*

$$D_6 = \texttt{positiveDeduction}(C_3, D_7 \cap D_5) = (\ell_e, \{y \to 4, z \to 2\}),$$

and adds it to $\mathbf{D}_{\texttt{found}}$ *and eventually to* $\mathbf{D}_{\texttt{true}}$*.*

The final iteration of the main loop does not add any more datapoints in $\mathbf{D}_{\texttt{found}}$*.*

Proposition 22. If $\mathbf{D}_{\texttt{prevTrue}}$ contains only forced-true datapoints, then every datapoint in the set that Algorithm 6 returns is forced true under $\mathrm{CS}(\mathbf{C})$ ▪

Proof. We apply a similar logic as for Proposition 17.

At line 5, $\mathrm{src}(C)$ is forced true because of Condition 1, and $\mathrm{tgt}(C)$ is forced true because of Condition 3. Therefore, at line 8, we initialize $\mathbf{D}_{\texttt{true}}$ with only forced-true datapoints.

We use induction to prove that we only add forced-true datapoints to $\mathbf{D}_{\texttt{true}}$ in the loop at line 9.

For the base case, we have already shown that the at the first execution of the loop, $\mathbf{D}_{\texttt{true}}$ only contains forced-true datapoints.

Assuming that every datapoint in $\mathbf{D}_{\texttt{true}}$ is forced true, we will show that the algorithm only adds forced-true datapoints to $\mathbf{D}_{\texttt{found}}$.

At line 16, and at line 24, we apply Condition 3 to determine that D_k or $\hat{D}_k$ is forced true, and their subsets are forced true because of Condition 2. Therefore, at line 30, we add only forced-true datapoints to $\mathbf{D}_{\texttt{true}}$.

Since we do not extend $\mathbf{D}_{\texttt{true}}$ anywhere else, it only contains forced-true datapoints. □

Proposition 23. The set of datapoints that Algorithm 6 returns contains all forced-true datapoints in $\mathbf{D}$. ▪

Proof. Assume that there is a forced-true datapoint not in $\mathbf{D}_{\texttt{true}}$. By Corollary 4, it can be traced back to a datapoint with the initial location. Let the datapoints in the trace be $D_1, D_2, \ldots, D_n$. Since D_1 has the initial location, the algorithm adds it at line 5.

Assuming D_i is added to $\mathbf{D}_{\texttt{true}}$, we show that D_{i+1} is added.

If D_{i+1} is forced true because $D_{i+1} \subseteq D_i$ (Condition 2), then $D_{i+1} \in \texttt{subsets}[D_i]$, and the algorithm also adds D_{i+1} to $\mathbf{D}_{\texttt{true}}$ when it adds D_i.

If D_{i+1} is forced true because there is a constraint C such that $D_{i+1} = \texttt{positiveDeduction}(C, D_i)$ (Condition 3), then in the first iteration of the loop at line 9 after the algorithm adds D_i to $\mathbf{D}_{\texttt{true}}$, when the loop at line 11 gets to C, it will add D_{i+1}.

Therefore, we have a contradiction. □

Proposition 24. When Algorithm 6 terminates, if there is a constraint $C \in \mathrm{CS}(\mathbf{C})$ and a datapoint $D \in \mathbf{D}_{\texttt{true}}$ such that $D \subseteq \mathrm{src}(C)$, then there is a datapoint $\tilde{D} \in \mathbf{D}_{\texttt{true}}$ such that $\texttt{positiveDeduction}(C, D) \subseteq \tilde{D}$. ▪

Proof. Let $\hat{C} \in \mathbf{C}$ be the constraint that caused C to be in $\mathrm{CS}(\mathbf{C})$. Let us consider the iteration of the loop at line 9 after D is added to $\mathbf{D}_{\texttt{true}}$ and the iteration of the loop at line 11 when it gets to $\hat{C}$.

If $\mathrm{src}\left(\hat{C}\right) \in \mathbf{D}_{\texttt{true}}$, then $\mathrm{tgt}\left(\hat{C}\right)$ is added to $\mathbf{D}_{\texttt{found}}$ at line 16. The datapoint $\mathrm{tgt}\left(\hat{C}\right)$ suffices as $\tilde{D}$, because $\texttt{positiveDeduction}(C, D) \subseteq \mathrm{tgt}\left(\hat{C}\right)$.

Since $D \in \mathbf{D}_{\texttt{true}}$, $D \in \mathbf{D}$ and since $D \subseteq \mathrm{src}\left(\hat{C}\right)$, $D \in \texttt{subsets}\left[\mathrm{src}\left(\hat{C}\right)\right]$. Therefore, if $\mathrm{src}\left(\hat{C}\right) \notin \mathbf{D}_{\texttt{true}}$, then when the loop at line 11 gets to $\hat{C}$, D is eventually selected at line 18, and $\texttt{positiveDeduction}\left(\hat{C}, D\right)$ is added at line line 24.

Since $\texttt{positiveDeduction}(C, D) \subseteq \texttt{positiveDeduction}\left(\hat{C}, D\right)$, the datapoint $\texttt{positiveDeduction}\left(\hat{C}, D\right)$ suffices as $\tilde{D}$. □

Algorithm 7: Finding forced-false datapoints

Data: $\mathbf{D}$, the set of datapoints. For every $C \in \mathbf{C}$, $\mathrm{src}(C) \in \mathbf{D}$ and $\mathrm{tgt}(C) \in \mathbf{D}$. For every non-disjoint $D_a \in \mathbf{D}$ and $D_b \in \mathbf{D}$, $D_a \cap D_b \in \mathbf{D}$.
Data: $\texttt{subsets}[D] = \{D' \in \mathbf{D} \mid D' \subseteq D\}$ for all $D \in \mathbf{D}$
Input: $\mathbf{C}$, a set of constraints
Input: ℓ_s, the initial location of the CFA
Input: ℓ_e, the error location of the CFA
Input: $\mathbf{D}_{\texttt{prevFalse}}$, a set of previously calculated forced-false datapoints. If $D \in \mathbf{D}_{\texttt{prevFalse}}$ then $\texttt{subsets}[D] \subseteq \mathbf{D}_{\texttt{prevFalse}}$.
Output: $\mathbf{D}_{\texttt{false}}$, the set of forced-false datapoints

```
1  D_found ← D_prevFalse;
2  forall (α_0, Φ, α_k, β) ∈ C do
3      C ← (α_0, Φ, α_k, β);
4      if tgt(Φ) = ℓ_e then
5          D_found ← D_found ∪ subsets[src(C)] ∪ subsets[tgt(C)];
6      end
7  end
8  D_false ← D_found;
9  while D_found ≠ ∅ do
10     D_found ← ∅;
11     forall C ∈ C do
12         D_0 ← src(C);
13         D_k ← tgt(C);
14         if D_0 ∉ D_false then
15             if D_k ∈ D_false then
16                 D_found ← D_found ∪ subsets[D_0];
17             else
18                 forall D̂_k ∈ subsets[D_k] ∩ D_false do
19                     D̂_0 ← negativeDeduction(C, D̂_k);
20                     if D̂_0 ∉ D then
21                         Call Algorithm 5 with D_new = {D̂_0};
22                     end
23                     if D̂_0 ∉ D_false then
24                         D_found ← D_found ∪ subsets[D̂_0];
25                     end
26                 end
27             end
28         end
29     end
30     D_false ← D_false ∪ D_found;
31 end
32 return D_false
```

Proposition 25. If $\mathbf{D}_{\texttt{prevFalse}}$ contains only forced-false datapoints, then every datapoint in the set that Algorithm 7 returns is forced false under CS($\mathbf{C}$) ▪

Proof. We apply a similar logic as for Proposition 22.

At line 5, src(C) is forced true because of Condition 1, and tgt(C) is forced true because of Condition 3. Therefore, at line 8, we initialize $\mathbf{D}_{\texttt{false}}$ with only forced-false datapoints.

We use induction to prove that we only add forced-false datapoints to $\mathbf{D}_{\texttt{false}}$ in the loop at line 9.

For the base case, we have already shown that the at the first execution of the loop, $\mathbf{D}_{\texttt{false}}$ only contains forced-true datapoints.

Assuming that every datapoint in $\mathbf{D}_{\texttt{false}}$ is forced false, we will show that the algorithm only adds forced-false datapoints to $\mathbf{D}_{\texttt{found}}$.

At line 16, and at line 24, we apply Condition 3 to determine that D_0 or $\hat{D}_0$ is forced false, and their subsets are forced false because of Condition 2. Therefore, at line 30, we add only forced-false datapoints to $\mathbf{D}_{\texttt{false}}$.

Since we do not extend $\mathbf{D}_{\texttt{false}}$ anywhere else, it only contains forced-false datapoints. □

Proposition 26. The set of datapoints that Algorithm 7 returns contains all forced-false datapoints in $\mathbf{D}$. ▪

Proof. Assume that there is a forced-false datapoint not in $\mathbf{D}_{\texttt{false}}$. By Corollary 5, it can be traced back to a datapoint with the error location. Let the datapoints in the trace be $D_1, D_2, \ldots, D_n$. Since D_1 has the error location, the algorithm adds it at line 5.

Assuming D_i is added to $\mathbf{D}_{\texttt{false}}$, we show that D_{i+1} is added.

If D_{i+1} is forced false because $D_{i+1} \subseteq D_i$ (Condition 2), then $D_{i+1} \in \texttt{subsets}[D_i]$, and the algorithm also adds D_{i+1} to $\mathbf{D}_{\texttt{false}}$ when it adds D_i.

If D_{i+1} is forced false because there is a constraint C such that $D_{i+1} = \texttt{negativeDeduction}(C, D_i)$ (Condition 3), then in the first iteration of the loop at line 9 after the algorithm adds D_i to $\mathbf{D}_{\texttt{false}}$, when the loop at line 11 gets to C, it will add D.

Therefore, we have a contradiction. □

These algorithms can be used incrementally, calling them whenever the teacher gives new constraints.

The set of forced-true datapoints that Algorithm 4 finds is a subset of what Algorithm 6 finds. Whenever Algorithm 4 adds a datapoint, Algorithm 6 adds that and all of its subsets. Therefore, we can also use Algorithm 6 to check if the constraint system is contradictory.

3.4.2 Learner algorithms

In this section, we will discuss the algorithms we use for synthesizing candidate invariant systems. Every algorithm relies on the datapoint data structure discussed in Section 3.4.1.2. When the teacher gives new constraints, we construct that data structure before running any of the following algorithms.

First, we run Algorithm 5 with $\mathbf{D}_{\texttt{new}} = \{\text{src}(C) \mid C \in \mathbf{C}\} \cup \{\text{tgt}(C) \mid C \in \mathbf{C}\}$. Then we run Algorithm 6 and check if the set of constraints is consistent. If they are, then we proceed with Algorithm 7 and one of the following algorithms.

3.4.2.1 Simple learner

Every satisfactory invariant system must classify forced-true datapoints as `true`. However, since the constraints are similar to implications—if the invariant system classifies one configuration as `true`, then it must also classify another one `true`—if it classifies every other configuration as `false`, then it adheres to them.

When given a set of constraints $\mathbf{C}$ and the set of forced-true datapoints $\mathbf{D}_{\texttt{true}}$ (the result of Algorithm 6), the simple learner returns the invariant system λ, where

$$\lambda[\ell] \Leftrightarrow \bigvee_{(\ell,\alpha)\in\mathbf{D}_{\texttt{true}}} \left(\bigwedge_{x\in\mathcal{X}_\alpha} x = \alpha(x) \right) \tag{3.2}$$

This invariant system classifies every datapoint in $\mathbf{D}_{\texttt{true}}$ as `true`, and every other datapoint as either `indeterminate` or `false` depending on whether they are disjoint from every datapoint in $\mathbf{D}_{\texttt{true}}$.[1]

We describe the process in Algorithm 8.

Algorithm 8: Simple learner

Data: $\mathbf{D}_{\texttt{true}}$, the set of forced-true datapoints as calculated by Algorithm 6
Input: ℓ_s, the start location
Input: ℓ_e, the error location
Input: N, the set of structural nodes
Output: λ, an invariant system that adheres to the constraint system

1 $\lambda[\ell_s] \leftarrow \top$;
2 $\lambda[\ell_e] \leftarrow \bot$;
3 **forall** $\ell \in N$ **do**
4 | $\lambda[\ell] \leftarrow \bot$;
5 **end**
6 **forall** $(\ell, \alpha) \in \mathbf{D}_{\texttt{true}}$ **do**
7 | $\lambda[\ell] \leftarrow \lambda[\ell] \vee (\bigwedge_{x\in\mathcal{X}_\alpha} x = \alpha(x))$; `// Equation 3.2`
8 **end**
9 **return** λ;

Corollary 7. The invariant system λ that Algorithm 8 returns classifies every configuration (ℓ, α) for which there is a datapoint $(\ell, \tilde{\alpha}) \in \mathbf{D}_{\texttt{true}}$ such that $\alpha \subseteq \tilde{\alpha}$ as `true`. It classifies every other configuration as `false`. ▪

Proposition 27. The invariant system λ that Algorithm 8 returns adheres to the constraint system $\mathrm{CS}(\mathbf{C})$. ▪

Proof. Assume that it does not. Then there is a constraint $C \in \mathrm{CS}(\mathbf{C})$, such that λ classifies $\mathrm{src}(C)$ as `true` and $\mathrm{tgt}(C)$ as `false`. By Corollary 7, there is a datapoint $D \in \mathbf{D}_{\texttt{true}}$ for which $\mathrm{src}(C) \subseteq D$. Therefore, by Proposition 24, there is a $\tilde{D} \in \mathbf{D}_{\texttt{true}}$ such that $\mathrm{tgt}(C) \subseteq \tilde{D}$. Then λ classifies $\tilde{D}$ as well as its subsets as `true`. □

---

[1] Configurations, on the other hand, are never classified as `indeterminate`.

Example 16. *For the set of forced-true datapoints* $\mathbf{D}_{\texttt{true}} = \{D_1, D_2, D_3, D_4\}$, *where*

$$\begin{aligned} D_1 =& (\ell_1, \{x \to 2\}), \\ D_2 =& (\ell_2, \{y \to 3, z \to 4\}), \\ D_3 =& (\ell_3, \{x \to 2, z \to 4\}), \\ D_4 =& (\ell_3, \{y \to 3, z \to 16\}), \end{aligned}$$

Algorithm 8 returns the invariant system λ, *where*

$$\begin{aligned} &\lambda[\ell_s] \Leftrightarrow \top, \\ &\lambda[\ell_1] \Leftrightarrow (x = 2), \\ &\lambda[\ell_2] \Leftrightarrow (y = 3) \wedge (z = 4), \\ &\lambda[\ell_3] \Leftrightarrow ((x = 2 \wedge z = 4) \vee (y = 3 \wedge z = 16)) \textit{ and} \\ &\lambda[\ell_e] \Leftrightarrow \bot. \end{aligned}$$

While Algorithm 8 finds an invariant system that adheres to the constraints, it does not generalize the constraints at all. In order for the learner to be successful, it cannot wait for enough constraints that the only solution to them is a satisfactory invariant system. It should be able to deduce generic properties of the CFA from the constraints.

3.4.2.2 Sorcar learner

Based on the HOUDINI and SORCAR algorithms in [6], we implemented Algorithm 9 as a learner. It tries to generate invariant systems by choosing a subset of a predetermined set $\mathcal{P}$ of predicates and forming the conjunction of its elements for each structural node. However, it does not always succeed. Sometimes it is not possible to express an invariant system that satisfies the set of constraints only using conjunctions of subsets of $\mathcal{P}$.

The algorithm starts from a large set of *relevant* predicates—the conjunction of which classifies a small set of datapoints as `true`—for each structural node. Then it removes predicates from the set—thereby making the conjunction classify more datapoints as `true`—if they contradict forced-true datapoints. Removing these predicates may cause not only forced-true datapoints to be classified `true`, but also other datapoints. These datapoints may appear as the source of some constraints and force the algorithm to label the target of those constraints `true` as well (by removing the predicates that contradict them) in order to adhere to those constraints.

We use Algorithm 6 to find these datapoints. As its input, we give the set of datapoints that the current conjunction labels `true`. Since they are not necessarily all forced-true datapoints, the output may contain datapoints that are not forced true by Definition 30. However, Algorithm 9 must classify these datapoints as `true` because every $\mathcal{P}$-subset conjunction that is consistent with the constraints does so. If there is a datapoint with the final location among them, it does not necessarily mean that the set of constraints is contradictory. It only means that Algorithm 9 is unable to generate an invariant system that adheres to the constraints.

The algorithm, therefore, is an iteration of having to remove predicates because they contradict one of the datapoints we have to label as `true`, removing those predicates causing other datapoints to be labelled `true`, and those datapoints in turn forcing us to label yet more datapoints as `true` through constraints.

Algorithm 9: Sorcar learner

Data: $\mathbf{D}$, the set of datapoints. If $D_a \in \mathbf{D}$ and $D_b \in \mathbf{D}$ are non-disjoint, then $D_a \cap D_b \in \mathbf{D}$

Data: $\texttt{subsets}[D] = \{D' \in \mathbf{D} \mid D' \subseteq D\}$ for all $D \in \mathbf{D}$

Input: $\mathbf{D}_{\texttt{true}}$, the set of forced-true datapoints as found by Algorithm 6

Input: $\mathcal{P}$, the set of predicates to build invariants from

Input: N, ℓ_s, ℓ_e, the set of structural nodes, the initial and the error location

Input: $\mathbf{C}$, the set of constraints

Output: λ, an invariant system that adheres to CS($\mathbf{C}$)

1 **forall** $\ell \in N$ **do**
2 | $\texttt{candidates}[\ell] \leftarrow \{\xi \in \mathcal{P} \mid \xi$ is relevant to $\ell\}$; `// Definition 32`
3 **end**
4 $\mathbf{D}_{\texttt{labelled}} \leftarrow \varnothing$; `// The set of datapoints labelled true`
5 $\mathbf{D}_{\texttt{toLabel}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{true}} \mid \ell \neq \ell_s\}$; `// Datapoints yet to be labelled true`
6 **repeat**
7 | **forall** $\ell \in N$ **do**
8 | | $\texttt{toRemove} \leftarrow \varnothing$;
9 | | **forall** $\xi \in \texttt{candidates}[\ell]$ **do**
10 | | | **forall** $(\ell, \alpha) \in \mathbf{D}_{\texttt{toLabel}}$ **do**
11 | | | | **if** $\alpha \not\models \xi$ **then** we must remove ξ to classify (ℓ, α) as `true`
12 | | | | | $\texttt{toRemove} \leftarrow \texttt{toRemove} \cup \{\xi\}$;
13 | | | | **end**
14 | | | **end**
15 | | **end**
16 | | $\texttt{candidates}[\ell] \leftarrow \texttt{candidates}[\ell] \setminus \texttt{toRemove}$;
17 | | **forall** $(\ell, \alpha) \in \mathbf{D} \setminus \mathbf{D}_{\texttt{labelled}}$ **do**
18 | | | **if** $\alpha \models \bigwedge(\texttt{candidates}[\ell])$ **then** we will classify (ℓ, α) as `true`
19 | | | | $\mathbf{D}_{\texttt{labelled}} \leftarrow \mathbf{D}_{\texttt{labelled}} \cup \texttt{subsets}[(\ell, \alpha)]$;
20 | | | **end**
21 | | **end**
22 | **end**
| `// Which datapoints get forced true by datapoints in` $\mathbf{D}_{\texttt{labelled}}$?
23 | $\mathbf{D}'_{\texttt{toLabel}} \leftarrow$ the result of Algorithm 6 with $\mathbf{D}_{\texttt{prevTrue}} = \mathbf{D}_{\texttt{labelled}}$;
24 | **if** *there is a datapoint* $(\ell_e, \alpha) \in \mathbf{D}'_{\texttt{toLabel}}$ **then**
25 | | **terminate**; `// The learner cannot express an invariant`
26 | **end**
27 | $\mathbf{D}_{\texttt{toLabel}} = \{(\ell, \alpha) \in \mathbf{D}'_{\texttt{toLabel}} \setminus \mathbf{D}_{\texttt{labelled}} \mid \ell \neq \ell_s\}$;
28 **until** $\mathbf{D}_{\texttt{toLabel}} = \varnothing$;
29 $\lambda[\ell_s] \leftarrow \top$;
30 $\lambda[\ell_e] \leftarrow \bot$;
31 **forall** $\ell \in N$ **do**
32 | $\lambda[\ell] \leftarrow \bigwedge(\texttt{candidates}[\ell])$
33 **end**
34 **return** λ;

We consider a predicate ξ relevant to a structural node ℓ if there is a constraint C such that $\text{src}(C) = (\ell, \alpha)$ and $\alpha \not\models \xi$. This means that by adding ξ to the conjunction, we can rule out at least one datapoint ($D \subseteq \text{src}(C)$) that would require $\texttt{positiveDeduction}(C, D)$ and possibly other datapoints to be forced true. On the other hand, predicates that are not relevant restrict the set of configurations classified as `true` in a way that does not help us adhere to the constraints.

Definition 32 (Relevant formula). In the context of a constraint system $\mathcal{C}$ and a set of datapoints $\mathbf{D}$, we say that a formula ξ is relevant to a structural node ℓ if there is a datapoint $(\ell, \alpha) \in \mathbf{D}$, and a constraint $C \in \mathcal{C}$ such that $(\ell, \alpha) \subseteq \text{src}(C)$ and $\alpha \not\models \xi$. ▪

Proposition 28. Datapoints in $\mathbf{D}_{\texttt{labelled}}$ are labelled `true` by the invariant system that Algorithm 9 returns. ▪

Proof. The algorithm only adds datapoints to $\mathbf{D}_{\texttt{labelled}}$ at line 19. The check before that line ensures that $\alpha \models (\bigwedge(\texttt{candidates}[\ell]))$. As the algorithm progresses, we only remove predicates from $\texttt{candidates}[\ell]$, and that maintains the previous relation. Finally, when the algorithm constructs λ, at line 32 $\lambda[\ell] \Leftrightarrow (\bigwedge(\texttt{candidates}[\ell]))$. Therefore, $\alpha \models \lambda[\ell]$, and λ labels (ℓ, α) as `true` along with every datapoint in $\texttt{subsets}[(\ell, \alpha)]$. ☐

Proposition 29. After Algorithm 9 executes line 16, for each $(\ell, \alpha) \in \mathbf{D}_{\texttt{toLabel}}$, $\alpha \models (\bigwedge(\texttt{candidates}[\ell]))$. ▪

Proof. There is no $\xi \in \texttt{candidates}[\ell]$, such that $\alpha \not\models \xi$, since it would be added to `toRemove` at line 12. If $\texttt{candidates}[\ell] = \varnothing$, then $\bigwedge(\texttt{candidates}[\ell]) \Leftrightarrow \top$. Otherwise, for every $\xi \in \texttt{candidates}[\ell]$, $\alpha \models \xi$. Therefore, $\alpha \models (\bigwedge(\texttt{candidates}[\ell]))$. ☐

Using these propositions, we can show that the invariant system that Algorithm 9 returns at line 34 classifies the datapoints in $\mathbf{D}$ consistently with $\text{CS}(\mathbf{C})$. It is possible that one of the datapoints is classified `indeterminate`.

Proposition 30. Let λ be the invariant system that Algorithm 9 returns at line 34. If there is a datapoint $D \in \mathbf{D}$ that λ classifies as `true` and a constraint $C \in \text{CS}(\mathbf{C})$ for which $D \subseteq \text{src}(C)$, then λ also classifies $\texttt{positiveDeduction}(C, D)$ as `true`. ▪

Proof. Assume that the algorithm returns an invariant system λ at line 34, and there is a constraint $C \in \text{CS}(\mathbf{C})$ for which $D \subseteq \text{src}(C)$, and λ classifies D as `true`. We will show that λ labels $\texttt{positiveDeduction}(C, D)$ as `true`.

The loop at line 17 always gets executed with the final state of `candidates` before the algorithm returns λ, therefore there is a point when D gets added to $\mathbf{D}_{\texttt{labelled}}$. After that, when we get to line 23, Algorithm 6 assumes that D is forced true and by Proposition 24, there is a $D' \in \mathbf{D}'_{\texttt{toLabel}}$ such that $\texttt{positiveDeduction}(C, D) \subseteq D'$.

If $D' \in \mathbf{D}_{\texttt{labelled}}$, then by Proposition 28, λ classifies D' as `true`.

If $D' \notin \mathbf{D}_{\texttt{labelled}}$, then $D' \in \mathbf{D}_{\texttt{toLabel}}$, and the algorithm does not exit the loop at line 28. By Proposition 29, it gets added to $\mathbf{D}_{\texttt{labelled}}$ in the next iteration, and by Proposition 28, λ classifies D' as `true`.

By Proposition 12, λ also classifies $\texttt{positiveDeduction}(C, D) \subseteq D'$ as `true`. ☐

Algorithm 9 does not always return an invariant system. We will show, however, that it always does when there is an invariant system that adheres to $\text{CS}(\mathbf{C})$ and only uses conjunctions of subsets of $\mathcal{P}$.

Proposition 31. If Algorithm 9 terminates at line 25, there is no invariant system λ such that λ adheres to the constraints and for every structural node $\ell \in N$, $\lambda[\ell] \Leftrightarrow \bigwedge(\mathcal{P}_\ell)$, where $\mathcal{P}_\ell \subseteq \mathcal{P}$. ▪

Proof. The algorithm terminates at line 25 if Algorithm 6 adds a datapoint with the error location to $\mathbf{D}'_{\texttt{toLabel}}$. Similarly to Corollary 4, we can trace back every datapoint in $\mathbf{D}'_{\texttt{toLabel}}$ to a datapoint in $\mathbf{D}_{\texttt{labelled}}$ or a datapoint whose location is ℓ_s.

If the datapoint (ℓ_e, α) at line 24 can be traced back to a datapoint whose location is ℓ_s, then the constraint system is contradictory and there is no possible solution to it.

Otherwise, it can be traced back to a datapoint in $\mathbf{D}_{\texttt{labelled}}$. As shown in the proof of Proposition 15, every classification that classifies the beginning of such a trace `true` must also classify every datapoint in the trace `true`. Therefore, the only way to prevent (ℓ_e, α) being added to $\mathbf{D}'_{\texttt{toLabel}}$ would be to add fewer datapoints to $\mathbf{D}_{\texttt{labelled}}$ which can only be achieved by removing fewer predicates from `candidates`. The algorithm removes a predicate either because it is irrelevant or because it prevents labelling a datapoint `true`.

First we show that we cannot prevent the addition of the datapoint (ℓ_e, α) to $\mathbf{D}'_{\texttt{toLabel}}$ by not removing irrelevant predicates. Since there are no datapoints with ℓ_e in $\mathbf{D}_{\texttt{labelled}}$, any trace leading to (ℓ_e, α) has to begin at a datapoint with a different location ℓ and include at least one reference using a constraint in $\mathrm{CS}(\mathbf{C})$. For a predicate ξ that is irrelevant to ℓ, there is no datapoint $(\ell, \beta) \in \mathbf{D}$ such that $\beta \not\models \xi$ and $(\ell, \beta) \subseteq \mathrm{src}(C)$ for some $C \in \mathrm{CS}(\mathbf{C})$. Assume that there is a trace leading from a datapoint with ℓ in $\mathbf{D}_{\texttt{labelled}}$ to (ℓ_e, α). Let C be the constraint that the first constraint-reference in the trace uses. The trace might start at a superset of $\mathrm{src}(C)$ with a subset-reference, or at a datapoint $D \subseteq \mathrm{src}(C)$. In both cases, $D \in \mathbf{D}_{\texttt{labelled}}$ and it cannot be excluded by irrelevant predicates.

Now we will use induction to prove that we must remove ξ at line 12. In the first iteration of the loop at line 6, $\mathbf{D}_{\texttt{toLabel}} = \mathbf{D}_{\texttt{true}}$, therefore by Proposition 15, every satisfactory invariant system must label every datapoint in $\mathbf{D}_{\texttt{toLabel}}$ as `true`. As long as $\xi \in \texttt{candidates}[\ell]$, $\alpha \not\models \bigwedge(\texttt{candidates}[\ell])$, because $\alpha \not\models \xi$. Therefore, ξ cannot be a part of $\lambda[\ell]$ for any invariant system using conjunctions of subsets of $\mathcal{P}$.

These removals cause other datapoints to be added to $\mathbf{D}_{\texttt{labelled}}$. Since none of the similar invariant systems can use the removed predicates, they all label these datapoints as `true`.

Then, at line 23, the every similar invariant system must also label the datapoints in $\mathbf{D}'_{\texttt{toLabel}}$ as `true`, since they would violate the constraints otherwise. Therefore, in the next iteration, the datapoints in $\mathbf{D}_{\texttt{toLabel}}$ must be labelled as `true`, and our reasoning can be repeated. □

Example 17. *Let* $\mathcal{P} = \{(x = 3), (x < y), (y \equiv 3 \pmod{10})\}$. *Let there be constraints* C_1, C_2 *and* C_3 *such that*

$$\begin{aligned}
\mathrm{src}(C_1) &= D_s = (\ell_s, \{\}),\\
\mathrm{tgt}(C_1) &= D_1 = (\ell_1, \{x \to 5, y \to 5\}),\\
\mathrm{src}(C_2) &= D_2 = (\ell_1, \{x \to 7, y \to 3\}),\\
\mathrm{tgt}(C_2) &= D_3 = (\ell_2, \{x \to 3, y \to 7\}),\\
\mathrm{src}(C_3) &= D_4 = (\ell_2, \{x \to 3, y \to 25\}),\\
\mathrm{tgt}(C_3) &= D_e = (\ell_e, \{\}).
\end{aligned}$$

The set of forced-true datapoints then is $\{D_s, D_1\}$.

Let us run Algorithm 9.

For ℓ_1, the predicates $(x = 3)$ and $(x < y)$ are relevant, because they are not true for $D_2 = \mathrm{src}(C_2)$, but $(y \equiv 3 \pmod{10})$ is not relevant. For ℓ_2, on the other hand, only $(y \equiv 3 \pmod{10})$ is relevant, because it is not true for $D_4 = \mathrm{src}(C_3)$.

Initially, at line 5, $\mathbf{D}_{\texttt{toLabel}} = \{D_1\}$. When the loop at line 7 gets to ℓ_1, the loop at line 9 removes both $(x = 3)$ and $(x < y)$ from `candidates`$[\ell_1]$ *because they are not true for D_1. This causes* `candidates`$[\ell_1]$ *to be empty, and the loop at line 17 to add D_1 and D_2 to $\mathbf{D}_{\texttt{labelled}}$.*

Then at line 23, D_3 is added to $\mathbf{D}'_{\texttt{toLabel}}$ because of C_2. Since $\mathbf{D}'_{\texttt{toLabel}} = \{D_s, D_1, D_2, D_3\}$, the algorithm proceeds to line 27 and $\mathbf{D}_{\texttt{toLabel}} = \{D_3\}$.

When the loop at line 7 gets to ℓ_2, the loop at line 9 removes $(y \equiv 3 \pmod{10})$ from `candidates`$[\ell_1]$ *because it is not true for D_3. This causes* `candidates`$[\ell_2]$ *to also be empty, and the loop at line 17 to add D_4 to $\mathbf{D}_{\texttt{labelled}}$.*

Finally, since at line 23, D_e is added to $\mathbf{D}'_{\texttt{toLabel}}$ because of C_3, the algorithm terminates at line 25.

If, however, we have an additional predicate $(y \neq 25)$, then the algorithm would synthesize the invariant system λ where

$$\begin{aligned}
&\lambda[\ell_s] \Leftrightarrow \top, \\
&\lambda[\ell_1] \Leftrightarrow \top, \\
&\lambda[\ell_2] \Leftrightarrow (y \neq 25), \\
&\lambda[\ell_e] \Leftrightarrow \bot.
\end{aligned}$$

3.4.2.3 Decision tree learner

We created the decision tree learner algorithm based on [3].

Decision trees are a well-known approach to classification problems in machine learning. In our case, the task of finding an invariant system that adheres to the constraints is slightly different from traditional classification problems since the classification of some datapoints is not fixed (they are not forced true or forced false), but they are not free either, labelling them as `true` may force the algorithm to label other datapoints as `true`. Moreover, some decision trees may classify some datapoints as `indeterminate`.

Decision trees have two types of nodes: branches and leafs. A branch contains a decision—in our case a predicate $\xi \in \mathcal{P}$, or a subset of the set of structural nodes $Z \subseteq N$—and they have two children nodes for the two possible outcomes of the decision. A leaf does not have children, and it is labelled with a classification—in our case `true` or `false`. The tree has a single root node, and all other nodes are its descendants (its children, or children of its children, etc.).

Our decision trees classify configurations of the CFA. To get the classification of a configuration (ℓ, α), we start at the root node of the decision tree. If it is a branch, then we check its label. If the label is a structural node set $Z \subseteq N$, then if $\ell \in Z$, we proceed to the left child, otherwise, we proceed to the right child. If the label is a predicate ξ, then if $\alpha \models \xi$, we proceed to the left child, otherwise $\alpha \models \neg\xi$ (by Proposition 2), and we proceed to the right child. We are finished when we reach a leaf: the label of the leaf node is how the decision tree classifies the configuration.

Similarly to previous algorithms, we only consider the classification of a set of datapoints during the building process and whether it violates the constraints. We hope that the resulting invariant system generalizes this information and is satisfactory.

Building the decision tree can be understood as a recursive process. The algorithm gets a set of datapoints, tries to classify them all the same, and checks if that labelling is consistent with the constraint system. If it is consistent, then it puts a leaf in the tree with the appropriate label. However, if it is not consistent, then the algorithm tries to split the datapoints into two groups—preferably one that can be labelled `true` and another that can be labelled `false`—with a decision: either a set of structural nodes or a predicate. It then puts in a branch node with the chosen decision, calculates the set of datapoints that get sent to the two children nodes and proceeds with the same process to create them. In case of a predicate decision ξ, a datapoint (ℓ, α) might have subsets $(\ell, \beta) \subseteq (\ell, \alpha)$ and $(\ell, \gamma) \subseteq (\ell, \alpha)$ such that $\beta \models \xi$ but $\gamma \models \neg\xi$. In such cases, the algorithm sends (ℓ, α) to both children, but notes that the datapoint itself was split: only a subset of it gets routed to either child. During labelling, the algorithm does not know the exact subset of split datapoints that are sent there. It assumes the best: if any subset of the split datapoint can be labelled consistently with the constraints, it accepts the labelling. This may lead to a classification that contradicts the constraints, but that is not catastrophic, since the teacher can give us more specific constraints that highlight that contradiction.

The output of the algorithm is not the decision tree, it is an invariant system. Therefore, it does not need to build the decision tree structure in memory. Instead, it can build a formula in disjunctive normal form (a disjunction of conjunctions) for every structural node. It keeps track of the decisions leading up to each set of datapoints waiting to be processed, and when it successfully labels a set `true`, it adds the conjunction of the ξ-decisions to the disjunction of every structural node that the Z-decisions direct that way. However, when it comes to classifying individual datapoints in Algorithm 12 at line 20, having a decision tree built can be beneficial, since checking if $\alpha \not\models \neg\varphi$ for every set waiting to be processed can be less efficient than traversing the decision tree.

Algorithm 10 is the main algorithm of the decision tree learner. It processes the nodes of the decision tree iteratively.

The set `toProcess` contains 4-tuples describing the decision tree nodes that are waiting to be processed. Every element is of the form $(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)$. The set of datapoints that the decision tree sends to the node are $\mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \subseteq \mathbf{D}$. Every subset of datapoints in $\mathbf{D}_{\texttt{whole}}$ is sent to this node, while datapoints in $\mathbf{D}_{\texttt{split}}$ have subsets that are sent to other nodes. The other two elements, M and φ track the decisions leading up to this node from the root node. The set $M \subseteq N$ is the set of structural nodes that the configurations that get routed to this node may have. It is the intersection of the Z or $\overline{Z}$ (depending on which direction) for Z-decisions leading up to this node. The formula φ is true for exactly the CFA configurations that get routed to this node. It is the conjunction of the ξ or $\neg\xi$ literals on the path leading up to the node.

An iteration of the loop at line 6 takes out an element from `toProcess` and tries to classify all the datapoints as either `true` or `false` using Algorithm 11. If it successfully classifies them as `true`, it updates λ to reflect the new classification. If it classifies them as `false`, λ does not need to be updated. But if the classification is unsuccessful, it creates a branch node with Algorithm 13, and adds the two sets that get forwarded to the two children to `toProcess`.

Algorithm 10: Decision tree learner

Data: $\mathbf{D}$, the set of datapoints
Data: $\mathbf{D}_{\texttt{true}}, \mathbf{D}_{\texttt{false}}$, the set of forced-true and forced-false datapoints
Input: N, ℓ_s, ℓ_e, the set of structural nodes and the start and error locations
Output: λ, an invariant system

1 $\lambda[\ell_s] \leftarrow \top$;
2 **forall** $\ell \in N \cup \{\ell_e\}$ **do**
3 | $\lambda[\ell] \leftarrow \bot$;
4 **end**
/* toProcess is the set of 4-tuples representing the nodes waiting to be processed: $(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)$ where $\mathbf{D}_{\texttt{whole}}$ and $\mathbf{D}_{\texttt{split}}$ are the set of whole/split datapoints to classify at the node, M is the set of structural nodes which the node applies to and φ is the conjunction of the decisions leading up to the node. */
5 toProcess $\leftarrow \{(\{(\ell, \alpha) \in \mathbf{D} \mid \ell \notin \{\ell_s, \ell_e\}\}, \varnothing, N, \top)\}$;
6 **while** toProcess $\neq \varnothing$ **do**
7 | $(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \leftarrow$ one of the elements of toProcess;
8 | toProcess $\leftarrow$ toProcess $\setminus \{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)\}$;
9 | label $\leftarrow$ the result of trying to label the datapoints with Algorithm 11;
10 | **if** label $=$ true **then**
11 | | **forall** $\ell \in M$ **do**
12 | | | $\lambda[\ell] \leftarrow \lambda[\ell] \vee \varphi$;
13 | | **end**
14 | **else if** label $=$ null **then**
15 | | decision $\leftarrow$ the best splitting decision found by Algorithm 13;
16 | | **if** decision *is a set of structural nodes* Z **then**
17 | | | $\varphi_{\texttt{left}} \leftarrow \varphi$;
18 | | | $\varphi_{\texttt{right}} \leftarrow \varphi$;
19 | | | $M_{\texttt{left}} \leftarrow M \cap Z$;
20 | | | $M_{\texttt{right}} \leftarrow M \setminus Z$;
21 | | | $\mathbf{D}_{\texttt{left}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \ell \in Z\}$;
22 | | | $\mathbf{D}_{\texttt{right}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \ell \notin Z\}$;
23 | | **else** // decision is the predicate ξ
24 | | | $\varphi_{\texttt{left}} \leftarrow \varphi \wedge \xi$;
25 | | | $\varphi_{\texttt{right}} \leftarrow \varphi \wedge \neg\xi$;
26 | | | $M_{\texttt{left}} \leftarrow M$;
27 | | | $M_{\texttt{right}} \leftarrow M$;
28 | | | $\mathbf{D}_{\texttt{left}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \alpha \not\models \neg\xi\}$;
29 | | | $\mathbf{D}_{\texttt{right}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \alpha \not\models \xi\}$;
30 | | **end**
31 | | $\mathbf{D}_{\texttt{leftWhole}} \leftarrow (\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{whole}}) \setminus \mathbf{D}_{\texttt{right}}$;
32 | | $\mathbf{D}_{\texttt{leftSplit}} \leftarrow (\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{split}}) \cup (\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{right}})$;
33 | | $\mathbf{D}_{\texttt{rightWhole}} \leftarrow (\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{whole}}) \setminus \mathbf{D}_{\texttt{left}}$;
34 | | $\mathbf{D}_{\texttt{rightSplit}} \leftarrow (\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{split}}) \cup (\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{left}})$;
35 | | toProcess $\leftarrow$ toProcess $\cup \{(\mathbf{D}_{\texttt{leftWhole}}, \mathbf{D}_{\texttt{leftSplit}}, M_{\texttt{left}}, \varphi_{\texttt{left}})\}$;
36 | | toProcess $\leftarrow$ toProcess $\cup \{(\mathbf{D}_{\texttt{rightWhole}}, \mathbf{D}_{\texttt{rightSplit}}, M_{\texttt{right}}, \varphi_{\texttt{right}})\}$;
37 | **end**
38 **end**
39 **return** λ;

Algorithm 11: Trying to label a set of datapoints

Data: $\mathbf{D}_{\texttt{true}}$, the set of forced-true datapoints
Data: $\mathbf{D}_{\texttt{false}}$, the set of forced-false datapoints
Input: $\mathbf{D}_{\texttt{whole}}$, the set of datapoints to label that have not been split by a previous decision
Input: $\mathbf{D}_{\texttt{split}}$, the set of datapoints to label that have been split by a previous decision
Output: The label, or `null` if the labelling is not successful

```
1  D_all ← D_whole ∪ D_split;
2  if D_all ⊆ D_true then
3  |  return true;
4  end
5  if D_all ⊆ D_false then
6  |  return false;
7  end
8  if D_all ∩ D_false = ∅ then
9  |  Run Algorithm 12 with label = true to determine if the datapoints can be
   |    labelled true;
10 |  if they can then
11 |  |  return true;
12 |  end
13 end
14 if D_all ∩ D_true = ∅ then
15 |  Run Algorithm 12 with label = false to determine if the datapoints can be
   |    labelled false;
16 |  if they can then
17 |  |  return false;
18 |  end
19 end
20 return null;
```

Algorithm 12: Check if labelling is consistent with the constraints

Data: $\mathbf{D}_{\texttt{true}}, \mathbf{D}_{\texttt{false}}$, the set of forced-true and forced-false datapoints
Data: `toProcess`, the set of 4-tuples waiting to be processed
Data: λ, the invariant system being built
Input: `label`, the label to try to give the datapoints
Input: $\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}$, the set of whole/split datapoints
Output: Whether the labelling is consistent with the constraints

1 **if** $\texttt{label} = \texttt{true}$ **then**
2 $\quad \mathbf{D}'_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{true}} \cup \mathbf{D}_{\texttt{whole}}$;
3 $\quad \mathbf{D}'_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{false}}$;
4 **else**
5 $\quad \mathbf{D}'_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{true}}$;
6 $\quad \mathbf{D}'_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{false}} \cup \mathbf{D}_{\texttt{whole}}$;
7 **end**
8 **repeat**
9 $\quad \mathbf{D}'_{\texttt{true}} \leftarrow$ the result of Algorithm 6 with $\mathbf{D}_{\texttt{prevTrue}} = \mathbf{D}'_{\texttt{true}}$;
10 $\quad \mathbf{D}'_{\texttt{false}} \leftarrow$ the result of Algorithm 7 with $\mathbf{D}_{\texttt{prevFalse}} = \mathbf{D}'_{\texttt{false}}$;
11 $\quad \mathbf{D}_{\texttt{new}} \leftarrow$ the new datapoints added to $\mathbf{D}$ by the previous two lines;
12 $\quad$ **if** *there is a datapoint* $(\ell_e, \alpha) \in \mathbf{D}'_{\texttt{true}}$ *or* $\mathbf{D}'_{\texttt{true}} \cap \mathbf{D}'_{\texttt{false}} \neq \varnothing$ **then**
13 $\quad\quad$ **return** `false`; `// The labelling contradicts the constraints`
14 $\quad$ **else if** $\texttt{label} = \texttt{true}$ **then**
15 $\quad\quad$ **if** $\mathbf{D}_{\texttt{split}} \cap \mathbf{D}'_{\texttt{false}} \neq \varnothing$ **then return** `false`;
16 $\quad$ **else**
17 $\quad\quad$ **if** $\mathbf{D}_{\texttt{split}} \cap \mathbf{D}'_{\texttt{true}} \neq \varnothing$ **then return** `false`;
18 $\quad$ **end**
19 $\quad$ **foreach** $(\ell, \alpha) \in \mathbf{D}_{\texttt{new}}$ **do**
20 $\quad\quad \texttt{relTP} \leftarrow \left\{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \in \texttt{toProcess} \mid \ell \in M \text{ and } \alpha \not\models \neg\varphi\right\}$;
21 $\quad\quad$ **if** $|\texttt{relTP}| = 1$ **then**
22 $\quad\quad\quad (\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \leftarrow$ the single element in `relTP`;
23 $\quad\quad\quad \texttt{toProcess} \leftarrow \texttt{toProcess} \setminus \left\{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)\right\}$;
24 $\quad\quad\quad \texttt{toProcess} \leftarrow \texttt{toProcess} \cup \left\{(\mathbf{D}_{\texttt{whole}} \cup \{(\ell, \alpha)\}, \mathbf{D}_{\texttt{split}}, M, \varphi)\right\}$;
25 $\quad\quad$ **else if** $|\texttt{relTP}| > 1$ **then**
26 $\quad\quad\quad$ **forall** $(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \in \texttt{relTP}$ **do**
27 $\quad\quad\quad\quad \texttt{toProcess} \leftarrow \texttt{toProcess} \setminus \left\{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)\right\}$;
28 $\quad\quad\quad\quad \texttt{toProcess} \leftarrow \texttt{toProcess} \cup \left\{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}} \cup \{(\ell, \alpha)\}, M, \varphi)\right\}$;
29 $\quad\quad\quad$ **end**
30 $\quad\quad$ **else if** $\alpha \models \lambda[\ell]$ **then**
31 $\quad\quad\quad \mathbf{D}'_{\texttt{true}} \leftarrow \mathbf{D}'_{\texttt{true}} \cup \{(\ell, \alpha)\}$;
32 $\quad\quad$ **else if** $\alpha \models \neg\lambda[\ell]$ **then**
33 $\quad\quad\quad \mathbf{D}'_{\texttt{false}} \leftarrow \mathbf{D}'_{\texttt{false}} \cup \{(\ell, \alpha)\}$;
34 $\quad\quad$ **end**
35 $\quad$ **end**
36 **until** $\mathbf{D}_{\texttt{new}} = \varnothing$;
37 $\mathbf{D}_{\texttt{false}} \leftarrow \mathbf{D}'_{\texttt{false}}$;
38 $\mathbf{D}_{\texttt{true}} \leftarrow \mathbf{D}'_{\texttt{true}}$;
39 **return** `true`;

Algorithm 13: Find splitting decision

Data: $\mathbf{D}_{\texttt{true}}$, the set of forced-true datapoints
Data: $\mathbf{D}_{\texttt{false}}$, the set of forced-false datapoints
Input: $\mathcal{P}$, the set of predicates to try
Input: $\textsc{Impurity} : \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{R}$ a metric to compare datapoint splits by
Input: $\mathbf{D}_{\texttt{toSplit}}$, the set of datapoints to split into two groups
Output: Either a set of structural nodes $Z \subseteq N$ or a predicate $\xi \in \mathcal{P}$

1 **if** *datapoints in* $\mathbf{D}_{\texttt{toSplit}}$ *correspond to multiple structural nodes* **then**
    `/* We decided to essentially build different decision trees for the different structural nodes. Therefore, as long as there are at least two structural nodes among the datapoints, we split by them. */`
2     $Z \leftarrow$ a set of approximately half of the structural nodes among the datapoints
      return Z;
3 **end**
4 $\xi_{\texttt{best}} \leftarrow \texttt{null}$;
5 $\mathsf{e}_{\texttt{best}} \leftarrow \texttt{null}$;
6 **forall** $\xi \in \mathcal{P}$ **do**
7     $\mathbf{D}_{\texttt{left}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{toSplit}} \mid \alpha \models \xi\}$;
8     $\mathbf{D}_{\texttt{right}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{toSplit}} \mid \alpha \models \neg\xi\}$;
9     $\mathsf{e}_{\texttt{left}} \leftarrow \textsc{Impurity}(|\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{true}}|, |\mathbf{D}_{\texttt{left}} \cup \mathbf{D}_{\texttt{false}}|, |\mathbf{D}_{\texttt{left}}|)$;
10     $\mathsf{e}_{\texttt{right}} \leftarrow \textsc{Impurity}(|\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{true}}|, |\mathbf{D}_{\texttt{right}} \cup \mathbf{D}_{\texttt{false}}|, |\mathbf{D}_{\texttt{right}}|)$;
11     $\mathsf{e} \leftarrow \mathsf{e}_{\texttt{left}} + \mathsf{e}_{\texttt{right}}$;
12   **if** $(\mathsf{e}_{\texttt{best}} = \texttt{null}) \vee (\mathsf{e} < \mathsf{e}_{\texttt{best}})$ **then**
13         $\xi_{\texttt{best}} \leftarrow \xi$;
14         $\mathsf{e}_{\texttt{best}} \leftarrow \mathsf{e}$;
15   **end**
16 **end**
17 **return** $\xi_{\texttt{best}}$;

Algorithm 11 tries to classify every datapoint the same. If every datapoint is either forced true or forced false, their classification is straightforward and consistent with the constraints. Otherwise, if none of the datapoints are forced false, it checks using Algorithm 12 if classifying them as `true` is consistent with the constraints. Similarly, if none of them are forced true, it checks if it can classify them as `false`.

Algorithm 12 determines if classifying a set of datapoints either `false` or `true` is consistent with the constraints. It creates a temporary set of forced-true and forced-false datapoints: $\mathbf{D}'_{\texttt{true}}$ and $\mathbf{D}'_{\texttt{false}}$. It adds the datapoints in $\mathbf{D}_{\texttt{whole}}$ to the one corresponding to the desired label. We do not calculate the exact subset of datapoints in $\mathbf{D}_{\texttt{split}}$ that get routed to a specific node in the decision tree. Adding the whole datapoint would imply that every one of its subsets get routed to this datapoint and might yield false inconsistencies. As stated before, a false consistency is preferable here, because the teacher can give more concrete constraints in the next round.

Algorithm 12 then uses Algorithm 6 and Algorithm 7 to find a contradiction, similarly to how Algorithm 9 does. As these algorithms make deductions, they sometimes discover new datapoints. Since some of the decision tree might already be built, these datapoints might already have classifications. Therefore, the algorithm checks their classification and if it contradicts with their constraint-enforced classifications.

Finally, when the datapoints cannot be classified the same, Algorithm 13 chooses a decision to split them into two groups. As its input, it receives Impurity, a function that estimates the *impurity* of the datapoints at a node of the decision tree. Impurity has three parameters: the number of forced-true datapoints, the number of forced-false datapoints and the total number of datapoints at the decision tree node. We say that a set of datapoints is pure, if every member can be classified the same. We expect Impurity to return a low number for such a set. Conversely, in an impure set of datapoints, about half of the datapoints have to be classified as `true` and the other half as `false`. We expect Impurity to return a high number for such a set.

The algorithm only compares Impurity values for different sets to each other. It ranks the possible decisions by the sum of the Impurity values for the two children. It chooses the decision with the lowest sum, which hopefully achieves two relatively pure sets of datapoints.

While other functions may yield better results, we use the following:

$$\textsc{Impurity}(n_{\texttt{true}}, n_{\texttt{false}}, n) = \min(n_{\texttt{true}}, n_{\texttt{false}}) + \frac{n - n_{\texttt{true}} - n_{\texttt{false}}}{2}.$$

The function is intended to estimate the number of misclassified datapoints: the number of datapoints that would be incorrectly classified if we classified every datapoint the same. The first part of the sum is the minimum number of misclassified datapoints. The second part estimates that about half of the datapoints that do not have forced classifications are misclassified. The second part also penalizes decisions that split datapoints, since the total number of datapoints is higher then.

Having the algorithm consider every possible Z-decision might make the decision tree smaller by merging paths for structural nodes that have similar decisions, but it would not make the invariant system simpler. Therefore, we decided to essentially build different decision trees for the different structural nodes by putting in decisions that send half the structural nodes one way and the other half the other way until there is only one.

Example 18. *Let* $\mathcal{P} = \{(x = 3), (x + y = 10)\}$*. Let there be constraints* C_1, C_2 *and* C_3 *such that*

$$\begin{aligned}
\mathrm{src}(C_1) &= D_s = (\ell_s, \{\}),\\
\mathrm{tgt}(C_1) &= D_1 = (\ell_1, \{x \to 5, y \to 13\}),\\
\mathrm{src}(C_2) &= D_2 = (\ell_1, \{x \to 7, y \to 3\}),\\
\mathrm{tgt}(C_2) &= D_3 = (\ell_2, \{x \to 3, y \to 7\}),\\
\mathrm{src}(C_3) &= D_4 = (\ell_2, \{x \to 3, y \to 25\}),\\
\mathrm{tgt}(C_3) &= D_e = (\ell_e, \{\}).
\end{aligned}$$

The set of forced-true datapoints is $\mathbf{D}_{\texttt{true}} = \{D_s, D_1\}$*, and the set of forced-false datapoints is* $\mathbf{D}_{\texttt{false}} = \{D_e, D_4\}$

Let us run Algorithm 10.

Initially at line 6,

$$\begin{aligned}
\texttt{toProcess} &= \{(\{D_1, D_2, D_3, D_4\}, \varnothing, \{\ell_1, \ell_2\}, \top)\},\\
\lambda[\ell_s] &\Leftrightarrow \top,\\
\lambda[\ell_1] \Leftrightarrow \lambda[\ell_2] \Leftrightarrow \lambda[\ell_e] &\Leftrightarrow \bot.
\end{aligned}$$

The first iteration of the loop calls Algorithm 11 which tries to label all the datapoints, but since $D_1 \in \mathbf{D}_{\texttt{true}}$ *and* $D_4 \in \mathbf{D}_{\texttt{false}}$*, it is unsuccessful and returns* `null`.

Then the main algorithm calls Algorithm 13 to find a splitting decision. It returns a structural node decision: $\{\ell_1\}$*. This results in*

$$\begin{aligned}
\texttt{toProcess} = \{&(\{D_1, D_2\}, \varnothing, \{\ell_1\}, \top),\\
&(\{D_3, D_4\}, \varnothing, \{\ell_2\}, \top)\}.
\end{aligned}$$

In the second iteration, the algorithm picks $(\{D_1, D_2\}, \varnothing, \{\ell_1\}, \top)$ *from* `toProcess`*. It tries to label it with Algorithm 11, which calls Algorithm 12 at line 9.*

In Algorithm 12, the first iteration results in

$$\mathbf{D}'_{\texttt{true}} = \{D_s, D_1, D_2, D_3\}, \mathbf{D}'_{\texttt{false}} = \{D_4, D_e\}, \mathbf{D}_{\texttt{new}} = \varnothing.$$

Since $\mathbf{D}_{\texttt{new}} = \varnothing$*, the labelling is successful and there are no more iterations.*

Algorithm 10 accepts the labelling and sets $\lambda[\ell_1] \Leftrightarrow \top$*.*

In the second iteration, the algorithm picks $(\{D_3, D_4\}, \varnothing, \{\ell_2\}, \top)$ *from* `toProcess`*. Since* $D_3 \in \mathbf{D}_{\texttt{true}}$ *and* $D_4 \in \mathbf{D}_{\texttt{false}}$*, the labelling is not successful. Algorithm 13 can choose from the predicates* $x = 3$*—which is true for both datapoints—and* $x + y = 10$*—which is true for* D_3 *and false for* D_4*. Naturally, it chooses* $x + y = 10$*. This results in*

$$\begin{aligned}
\texttt{toProcess} = \{&(\{D_3\}, \varnothing, \{\ell_2\}, x + y = 10),\\
&(\{D_4\}, \varnothing, \{\ell_2\}, x + y \neq 10)\}.
\end{aligned}$$

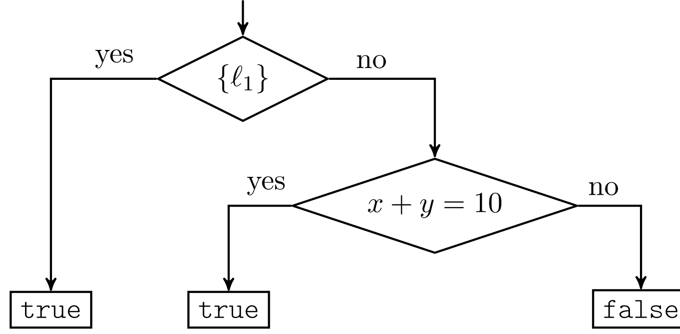

Figure 3.2: Decision tree for Example 18

In the fourth iteration, Algorithm 10 picks $(\{D_3\}, \varnothing, \{\ell_2\}, x + y = 10)$ *from* `toProcess`. *Algorithm 11 succeeds in labelling* D_3 *as* `true` *since* $D_3 \in \mathbf{D}_{\texttt{true}}$. *Therefore,* $\lambda[\ell_2] \Leftrightarrow (x + y = 10)$.

In the fifth and final iteration, Algorithm 10 picks $(\{D_4\}, \varnothing, \{\ell_2\}, x + y \neq 10)$ *from* `toProcess`. *Algorithm 11 succeeds in labelling* D_4 *as* `false` *since* $D_4 \in \mathbf{D}_{\texttt{false}}$.

The resulting decision tree can be seen in Figure 3.2. For the resulting invariant system λ

$$\begin{aligned}
\lambda[\ell_s] &\Leftrightarrow \top, \\
\lambda[\ell_1] &\Leftrightarrow \top, \\
\lambda[\ell_2] &\Leftrightarrow (x + y = 10), \\
\lambda[\ell_e] &\Leftrightarrow \bot.
\end{aligned}$$

Algorithm 10 can synthesize invariant systems as long as there are predicates in $\mathcal{P}$ that can separate the datapoints. We recommend using it with a $\mathcal{P}$ set based on the set of datapoints, that includes at least one predicate for every separable pair of datapoints that separates them.

Chapter 4

Implementation

We implemented the discussed algorithms in the Kotlin programming language as part of the THETA[11] framework. The framework provides an internal representation for variables, logical formulae, expressions, statements, control flow automata etc., as well as utilities for e.g. parsing CFA or making requests to an SMT solver.

4.1 Main modules

We implemented a TEACHER module based on Algorithm 3 and LEARNER modules based on Algorithm 8 (SIMPLELEARNER), Algorithm 9 (SORCARLEARNER) and Algorithm 10 (DECISIONTREELEARNER).

The data structure consisting of $\mathbf{D}$, `subsets`, $\mathbf{D}_{\texttt{true}}$ and $\mathbf{D}_{\texttt{false}}$ is represented by the CONSTRAINTSYSTEM module. We chose to make CONSTRAINTSYSTEM objects immutable, and we implemented the CONSTRAINTSYSTEMBUILDER module which uses Algorithm 5, Algorithm 6 and Algorithm 7 to create CONSTRAINTSYSTEM instances. Additionally, CONSTRAINTSYSTEMBUILDER checks if there is a datapoint with the error location in $\mathbf{D}_{\texttt{true}}$.

To coordinate the interaction of TEACHER and LEARNER modules, we implemented two different COORDINATOR modules. A COORDINATOR has the responsibility of giving the invariant system that the LEARNER suggests to a TEACHER to check, using CONSTRAINTSYSTEMBUILDER to create a CONSTRAINTSYSTEM instance from the constraints that the TEACHER returns, and giving it to a LEARNER to get a new invariant system. It also notices when an invariant system is satisfactory or a constraint system is contradictory and returns the appropriate result.

The SIMPLECOORDINATOR implementation coordinates the interaction of one TEACHER and one LEARNER module in a single-threaded way.

The MULTITHREADEDCOORDINATOR, however, utilizes multi-core processors by running multiple LEARNER and multiple TEACHER modules as separate threads. The teachers have a common input buffer, and everything put in the input buffer is checked by exactly one of the teachers. The teachers also put their output in a common buffer, which the main thread reads and processes. The main thread accumulates every constraint ever returned by any of the teachers. The learners have separate input buffers, each holding at most one (the latest) CONSTRAINTSYSTEM, and they put their results into the common input buffer of the teachers. If the constraints get so complicated that a learner is no longer able to synthesize an invariant system that adheres to them, they stop their thread.

When the main thread notices that one of the teachers either finds an invariant system satisfactory or gives contradictory constraints, it interrupts every thread, and returns the appropriate result.

4.2 Learner combinations

In addition to the standalone LEARNER modules, we implemented two combination modules. These modules are single-threaded, but try to combine the results of multiple learners.

FALLBACKLEARNER has a list of LEARNER modules. It forwards the CONSTRAINTSYSTEM to the first learner in the list until the constraints become too complicated for that learner to synthesize an invariant system that adheres to them. Then it removes the learner from the list, and proceeds using the previous second, now first element of the list. When the list becomes empty, the FALLBACKLEARNER signals that it is no longer able to synthesize invariant systems that adhere to the constraints.

ROUNDROBINLEARNER also has a list of LEARNER modules. It rotates the list at every request: it forwards the first CONSTRAINTSYSTEM to the first LEARNER in the list, the second CONSTRAINTSYSTEM to the second LEARNER, etc., and when it gets to the end of the list, it starts again at the beginning. If one of the LEARNER modules is unable to synthesize an invariant system that adheres to the constraints, the ROUNDROBINLEARNER removes it from the list. When the list becomes empty, the ROUNDROBINLEARNER signals that it is no longer able to synthesize invariant systems that adhere to the constraints.

4.3 Predicate patterns

SORCARLEARNER and DECISIONTREELEARNER use a set of predicates $\mathcal{P}$. We implemented four PREDICATEPATTERN modules that can generate predicates.

INTLEQ creates predicates of the form $x \leq a$ where $x \in X$ is a variable and $a \in \mathbb{Z}$ is an integer. From the infinite set of such predicates, it only creates the ones that are relevant to the current set of datapoints. It adds the predicate $x \leq a$ if and only if there is a datapoint whose valuation assigns a to x. These types of predicates can also be ranked more efficiently for the decision tree: sorting the datapoints by the value they assign to a variable, then going through the values, we can simply count the forced-true and forced-false datapoints that get from the right child to the left child when we increase the value.

ATOM extracts atoms from the CFA and uses them as predicates. For example, if there are statements $[x < y + 2 \cdot z]$ and $(z \coloneqq x + y)$, it adds $(x < y + x \cdot z)$ and $(z = x + y)$ to $\mathcal{P}$.

INTBUILDER extracts the integer expressions from the CFA and uses them to build predicates of the form $a = b$, $a < b$ and $a > b$ where a and b are integer expressions. For example if there are statements $[x < y + 2 \cdot z]$ and $y \coloneqq x - 3 \cdot y$, it uses the expressions

$\{x, y + 2 \cdot z, y, x - 3 \cdot y\}$, which results in the following predicates:

$$\begin{gathered}(x = y + 2 \cdot z), (x < y + 2 \cdot z), (x > y + 2 \cdot z),\\ (x = y), (x < y), (x > y),\\ (x = x - 3 \cdot y), (x < x - 3 \cdot y), (x > x - 3 \cdot y),\\ (y + 2 \cdot z = y), (y + 2 \cdot z < y), (y + 2 \cdot z > y),\\ (y + 2 \cdot z = x - 3 \cdot y), (y + 2 \cdot z < x - 3 \cdot y), (y + 2 \cdot z > x - 3 \cdot y),\\ (y = x - 3 \cdot y), (y < x - 3 \cdot y), (y > x - 3 \cdot y).\end{gathered}$$

MODULUS extracts the integer expressions from the CFA similarly to INTBUILDER, but it uses them to create predicates of the form $a \equiv b \pmod{c}$ where a, b and c are all integer expressions.

4.4 Configurability

Our implementation can run in several different configurations. The user can choose which LEARNER implementation or implementations to use, how to combine them, they can choose PREDICATEPATTERN implementations for each LEARNER, and they can choose whether to use the SIMPLECOORDINATOR or MULTITHREADEDCOORDINATOR.

We determined that the range of options is too complicated to be convenient to configure with just command line arguments. Therefore, we decided to also allow configuration using YAML [1].

With command line options, one can configure a system with a SIMPLECOORDINATOR, one teacher and one combination of a number of learners. One can also choose the PREDICATEPATTERN implementations, and the IMPURITY function, but every learner uses the same set.

With the YAML file, one can configure any system. The file has a hierarchic structure of YAML mappings and sequences. The root object is a mapping with the following keys:

- `coordinator`: either MULTITHREADED or SIMPLE (default: SIMPLE),
- `teachers`: the number of TEACHER objects (must be 1 for SIMPLECOORDINATOR, default: 1),
- `learners`: sequence of learner objects (must have only one element for SIMPLECOORDINATOR, required).

A learner object is a mapping with the following keys:

- `type`: SIMPLE, SORCAR, DECISIONTREE, FALLBACK or ROUNDROBIN (default: DECISIONTREE)
- `name`: the name of the learner, used for logging purposes (optional)
- `predicatePatterns`: sequence of ATOMS, INTLEQ, INTBUILDER or MODULUS (default: [ATOMS, INTLEQ])
- `children`: a sequence of learner objects that FALLBACKLEARNER or ROUNDROBINLEARNER combine (required for FALLBACKLEARNER and ROUNDROBINLEARNER).

Chapter 5

Evaluation

We ran measurements to evaluate the performance of the prototype we developed. In every instance, the prototype either gave the correct solution, ran out of time or ran out of memory.

5.1 Methodology

We used a benchmark suite of 569 models, most of them from the International Competition on Software Verification (SV-COMP) [8]. Of the used models, 103 were unsafe and 466 were safe. Every model was converted to the CFA format used by the Theta framework prior to the measurement.

For the measurement, we used a virtual machine in the university cloud with 4 CPU cores and 8 GiB of memory. Swapping was disabled. We stopped the benchmarks after either 60 seconds of wall time has elapsed or the prototype used more than 60 seconds of CPU time, where CPU time is the sum of the time each individual CPU core spends executing the process. For configurations using the MultiThreadedCoordinator, the elapsed CPU time can be up to 4 times as large as the wall time. By also stopping the benchmarks based on CPU time, we ensure that the multithreaded configurations do not get an advantage of computational resources, only the benefit of being able to pursue multiple directions at the same time. Single-threaded combinations of learners make other learners wait while one learner is running, thereby allocating more time to learners that run for a long time. Multithreaded combinations, on the other hand, are more fair in their allocation of CPU time among the learners.

5.2 Tested configurations

We ran the prototype with the following 10 configurations.

C1: Simple-Simple
- SimpleCoordinator
- SimpleLearner

C2: Simple-DecTree-Few
- SimpleCoordinator
- DecisionTreeLearner with Atoms, IntLEQ

C3: Simple-DecTree-All
- SimpleCoordinator
- DecisionTreeLearner with Atoms, IntLEQ, Modulus, IntBuilder

C4: Simple-Fallback-Few
- SimpleCoordinator
- FallbackLearner
  - SorcarLearner with Atoms
  - DecisionTreeLearner with Atoms, IntLEQ

C5: Simple-Fallback-All
- SimpleCoordinator
- FallbackLearner
  - SorcarLearner with Atoms
  - DecisionTreeLearner with Atoms, IntLEQ, Modulus, IntBuilder

C6: Simple-RoundRobin-Few
- SimpleCoordinator
- RoundRobinLearner
  - SimpleLearner with every predicate pattern
  - FallbackLearner
    * SorcarLearner with Atoms
    * DecisionTreeLearner with Atoms, IntLEQ

C7: Simple-RoundRobin-All
- SimpleCoordinator
- RoundRobinLearner
  - SimpleLearner
  - FallbackLearner
    * SorcarLearner with Atoms
    * DecisionTreeLearner with Atoms, IntLEQ, Modulus, IntBuilder

C8: Multi-Fallback-Few
- MultiThreadedCoordinator
- 2 teachers
- SimpleLearner with Atoms
- FallbackLearner
  - SorcarLearner with Atoms
  - DecisionTreeLearner with Atoms, IntLEQ

C9: Multi-Fallback-All
- MultiThreadedCoordinator
- 2 teachers
- SimpleLearner with Atoms
- FallbackLearner
  - SorcarLearner with Atoms
  - DecisionTreeLearner with Atoms, IntLEQ, Modulus, IntBuilder

Configuration	Total	Safe	Unsafe
C1 (Simple-Simple)	65	25	40
C2 (Simple-DecTree-Few)	237	205	32
C3 (Simple-DecTree-All)	181	153	28
C4 (Simple-Fallback-Few)	237	207	30
C5 (Simple-Fallback-All)	203	175	28
C6 (Simple-RoundRobin-Few)	241	204	37
C7 (Simple-RoundRobin-All)	211	176	35
C8 (Multi-Fallback-Few)	213	175	38
C9 (Multi-Fallback-All)	182	148	34
C10 (Multi-DecTree-Sep)	163	142	21

Table 5.1: Number of models solved by each configuration

C10: Multi-DecTree-Sep
- MultiThreadedCoordinator
- 4 teachers
- DecisionTreeLearner with Atoms predicate pattern
- DecisionTreeLearner with IntLEQ predicate pattern
- DecisionTreeLearner with Modulus predicate pattern
- DecisionTreeLearner with IntBuilder predicate pattern

5.3 Results

264 models were solved by at least one of the configurations, i.e., approximately 46.4% of all models. Among those, 44 were unsafe and 220 were safe. I.e., the prototype was able to solve approximately 42.7% of the unsafe and 47.2% of the safe models.

Table 5.1 shows the number of solved models by configuration.

Unsurprisingly, Simple-Simple was the worst overall. However, only considering the unsafe models, it was the best. It uses SimpleLearner, which is the simplest and computationally least expensive of the learners. This allows it to quickly get new constraints, and the larger number of constraints allows it to find longer error paths faster. On the other hand, due to their lack of generality, the invariant systems it synthesizes are rarely satisfactory. In a suite that has more unsafe models, we expect this configuration to rank higher overall. Adding SimpleLearner to configurations combining multiple learners seems to improve their performance on unsafe models. This is a trade-off in our test setup, because adding it takes away CPU time from the other learners, lowering their chance to generate a satisfactory invariant system for safe models.

Among the safe models, Simple-Fallback-Few performed the best. It solved 207 safe models. The SorcarLearner synthesized the invariant system for 155 of those, and the DecisionTreeLearner synthesized the invariant system for an additional 52 models that were too complicated for SorcarLearner. Our theory is that SorcarLearner is very efficient in synthesizing invariants for relatively simple models, and this configuration allows the apparently less efficient but more capable DecisionTreeLearner to step in for some of the more complicated cases. Another seeming advantage of the SorcarLearner used in this configuration is that it only tries the predicates extracted from the checked program, which are the most relevant to it, therefore the most likely to appear in a satisfactory invariant system. Using other predicate patterns slows the process of invariant generation.

C2	C4	C6	Total	Safe	Unsafe
			317	252	65
✓	✓	✓	225	195	30
		✓	10	4	6
✓	✓		7	7	0
	✓	✓	4	4	0
✓			3	2	1
✓		✓	2	1	1
	✓		1	1	0

Table 5.2: Comparison of Simple-DecTree-Few, Simple-Fallback-Few and Simple-RoundRobin-Few

C6	C8	Total	Safe	Unsafe
		324	261	63
✓	✓	209	174	35
✓		32	30	2
	✓	4	1	3

Table 5.3: Comparison of Simple-RoundRobin-Few and Multi-Fallback-Few

Simple-RoundRobin-Few ended up being the best configuration overall. It combines the best configuration for unsafe models and the best configuration for safe models. It solved 241 models. 204 of those were safe, the SimpleLearner synthesized the invariant system for 8 of them, the SorcarLearner for 149 of them and the DecisionTreeLearner for an additional 47.

Table 5.2 compares the single-threaded configurations using only the Atoms and IntLEQ predicate patterns. We can see that both combining DecisionTreeLearner with SorcarLearner and also combining them with SimpleLearner has some drawbacks. There were some models that the combined learners could solve individually, but not together, since they had to share computational resources. Overall, the advantages outweighed the disadvantages, and the combinations performed better.

In our current setup, multithreaded configuration did not perform well. Using multiple CPU cores simultaneously caused the allocated CPU time to run out faster, and the overhead caused by organizing multiple threads seemingly outweighed the advantage of the learners not having to wait for each other. The best configuration among both the safe and unsafe models was Multi-Fallback-Few. It combines the same learners as Simple-RoundRobin-Few, but instead of using RoundRobinLearner, it uses MultiThreadedCoordinator. Table 5.3 shows that there are only 4 configurations that the multithreaded combination can solve, but the single threaded cannot.

Adding the predicate patterns that offer numerous predicates (IntBuilder and Modulus) deteriorated performance in our test setup. The best configuration using these predicate patterns was Simple-RoundRobin-All. It only differs from Simple-RoundRobin-Few in giving IntBuilder and Modulus to the DecisionTreeLearner. Of the 176 safe models it solved, the SimpleLearner synthesized the invariant system for 8, the SorcarLearner for 149 of them, and the DecisionTreeLearner with all of the predicate patterns synthesized only 19 satisfactory invariant systems. Table 5.4 compares the performance of the best configuration using all available predicate patterns and the best configuration overall. There were only 3 models that only Simple-RoundRobin-All

C6	C7	Total	Safe	Unsafe
		325	259	66
✓	✓	208	173	35
✓		33	31	2
	✓	3	3	0

Table 5.4: Comparison of SIMPLE-ROUNDROBIN-FEW and SIMPLE-ROUNDROBIN-ALL

Predicate Pattern	Synthesized invariant systems
INTLEQ	60
ATOMS	48
MODULUS	18
INTBUILDER	16

Table 5.5: Comparison of the different learners in MULTI-DECTREE-SEP

could solve. The time and memory we allocated for the prototype was not enough to successfully utilize the large number of predicates these patterns can generate.

The configuration MULTI-DECTREE-SEP allows us to further compare the different predicate patterns, because it ran four instances of DECISIONTREELEARNER, each using of the patterns. Table 5.5 shows the number of models each learner generated an invariant system for. Note that for each safe model that the configuration solved only one learner was able to generate the invariant system. Our implementation of the INTLEQ pattern (which ranks the available predicates more efficiently than other patterns) seems to suite the DECISIONTREELEARNER well. As we saw earlier, the MODULUS and INTBUILDER patterns were only useful a few times in this test suite. Allocating more time and memory to the prototype might make these patterns useful for models that the prototype did not solve in the current setup.

Table 5.6 provides a breakdown of the models by the set of configurations that solved them. The first ten columns correspond to the configurations and the rows give the number of models that were solved by exactly the configurations whose column has a ✓ symbol in the row.

Our earlier implementation of the predicate patterns INTBUILDER and MODULUS created every predicate in memory, which caused the prototype to run out of memory for larger models. We tweaked the implementation to only generate the predicates lazily. This is a trade-off, because generating the predicates requires CPU time, but not storing them decreases memory usage. Figure 5.1 and Figure 5.2 show that while running out of memory still occurs in our current implementation, CPU time tends to be the low resource and the configurations using the INTBUILDER and MODULUS patterns do not tend to us significantly more memory than their counterparts using only ATOMS and INTBUILDER.

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	Total	Safe	Unsafe
										305	246	59
	✓	✓	✓	✓	✓	✓	✓	✓	✓	98	97	1
✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	33	14	19
	✓		✓		✓		✓			19	19	0
	✓		✓	✓	✓	✓	✓	✓		14	14	0
	✓	✓	✓	✓	✓	✓	✓		✓	12	12	0
	✓		✓		✓					9	9	0
	✓	✓	✓	✓	✓	✓			✓	8	8	0
	✓		✓							6	6	0
	✓	✓	✓	✓	✓	✓		✓	✓	6	6	0
✓	✓		✓	✓	✓	✓	✓	✓		5	5	0
✓										5	1	4
✓	✓	✓	✓	✓	✓	✓	✓	✓		5	0	5
✓					✓	✓	✓	✓		5	1	4
	✓	✓	✓	✓	✓	✓	✓	✓		3	3	0
	✓									2	2	0
		✓		✓		✓				2	2	0
					✓		✓			2	2	0
	✓	✓	✓	✓	✓	✓		✓		2	2	0
							✓			2	1	1
✓	✓	✓	✓	✓	✓	✓		✓	✓	2	2	0
✓	✓		✓		✓	✓	✓	✓		1	0	1
✓	✓	✓	✓	✓	✓	✓	✓			1	0	1
					✓		✓	✓		1	0	1
			✓		✓		✓			1	1	0
			✓	✓	✓	✓	✓			1	1	0
			✓	✓	✓	✓	✓	✓		1	1	0
		✓		✓						1	1	0
✓	✓	✓	✓	✓	✓	✓			✓	1	0	1
		✓	✓	✓		✓			✓	1	1	0
		✓	✓	✓	✓	✓	✓		✓	1	1	0
	✓	✓	✓	✓	✓	✓				1	1	0
	✓						✓	✓		1	0	1
	✓	✓	✓	✓	✓	✓	✓			1	1	0
✓	✓		✓		✓		✓			1	0	1
✓	✓				✓	✓	✓	✓		1	0	1
	✓		✓	✓	✓	✓		✓		1	1	0
	✓		✓	✓	✓	✓	✓			1	1	0
✓					✓	✓	✓			1	0	1
	✓	✓			✓	✓	✓			1	1	0
	✓	✓	✓	✓					✓	1	1	0
✓					✓	✓		✓		1	1	0
✓							✓	✓		1	0	1
✓								✓		1	1	0
✓	✓	✓	✓	✓	✓	✓				1	0	1

Table 5.6: Number of models by which configurations solved them

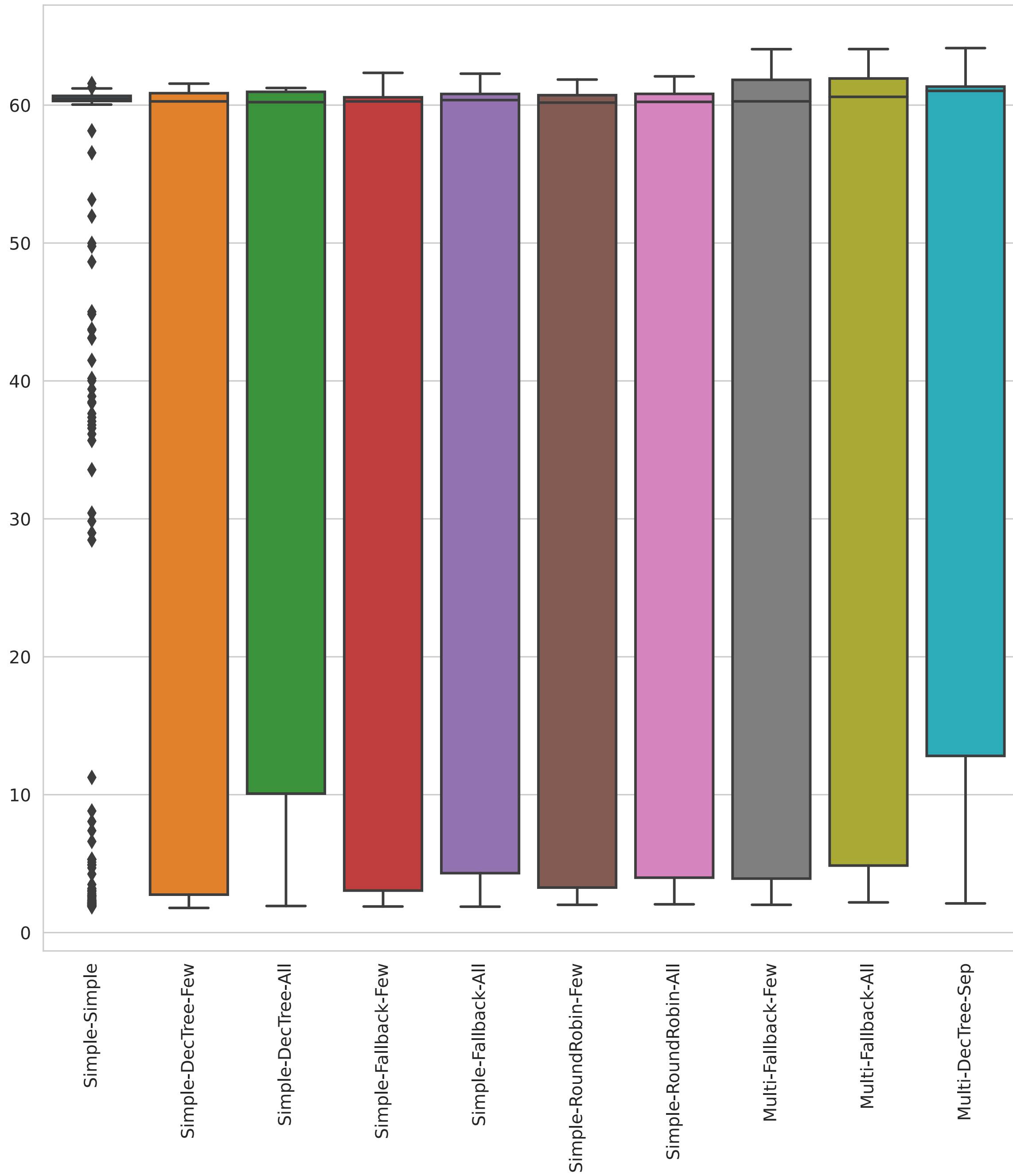

Figure 5.1: Box plot of CPU time in seconds used by the configurations

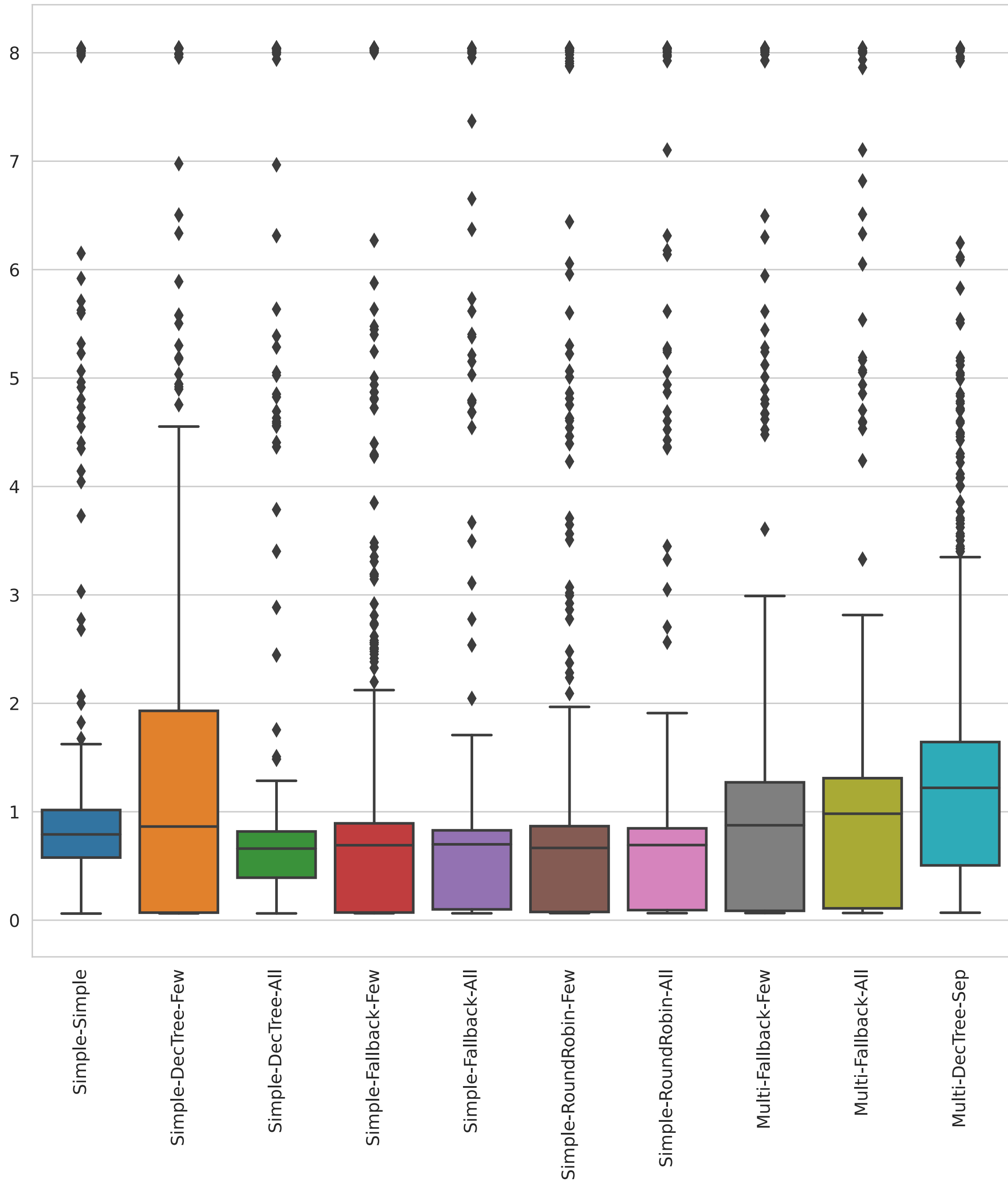

Figure 5.2: Box plot of memory in gigabytes used by the configurations

Chapter 6

Conclusion

In this thesis, we have discussed the formal verification of software through invariant synthesis. We have introduced the theoretical background to invariant synthesis for control flow automata. We have presented a family of algorithms utilizing machine learning techniques to synthesize invariant systems. We have implemented the algorithms as a configurable and extendable prototype that allows the user to combine them in multiple ways. Finally, we have evaluated multiple configurations of the prototype with measurements.

Future work The framework we developed can be extended with other types of learner algorithms. As we have shown, invariant synthesis can be treated as a classification problem, which the field of machine learning offers many solutions to. While other solutions may not be as straightforward to translate to logical formulae as e.g. decision trees, it may be possible to integrate them into the framework.

Adding predicate patterns can increase the applicability of the toolkit by allowing it to synthesize more complicated invariants. In our implementation, however, it also has a drawback, as the predicates that are not useful to the current task waste resources. A process could be developed to choose predicate patterns based on the program code.

The approach also lends itself to interactive verification. A tool could be developed that asks the user for invariants and gives feedback in the form of constraints. It may increase the usability of verification techniques for complicated cases when the automatic tools fail, but engineers have knowledge of the system.

Bibliography

[1] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. Yaml ain't markup language. https://yaml.org/spec/1.2/spec.html, 2009. Retrieved: 2020-12-08.

[2] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. Horn clause solvers for program verification. In Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte, editors, *Fields of Logic and Computation II*, volume 9300 of *Lecture Notes in Computer Science*, pages 24–51. Springer, 2015. DOI: 10.1007/978-3-319-23534-9_2.

[3] P. Ezudheen, Daniel Neider, Deepak D'Souza, Pranav Garg, and P. Madhusudan. Horn-ICE learning for synthesizing invariants and contracts. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA), oct 2018. DOI: 10.1145/3276501.

[4] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Solving constrained horn clauses using syntax and data. In Nikolaj Bjørner and Arie Gurfinkel, editors, *Proceedings of the 18th Conference on Formal Methods in Computer-Aided Design*, pages 170–178, 2018. ISBN 978-0-9835678-8-2. DOI: 10.23919/FMCAD.2018.8603011.

[5] Miklós Ferenczi. *Matematikai logika*. Műszaki Kiadó, 2014. ISBN 9789631628708.

[6] Daniel Neider, Shambwaditya Saha, Pranav Garg, and P. Madhusudan. SORCAR: Property-driven algorithms for learning conjunctive invariants. In Bor-Yuh Evan Chang, editor, *Static Analysis*, volume 11822 of *Lecture Notes in Computer Science*, page 323–346. Springer, 2019. DOI: 10.1007/978-3-030-32304-2_16.

[7] Daniel Neider, P. Madhusudan, Shambwaditya Saha, Pranav Garg, and Daejun Park. A learning-based approach to synthesizing invariants for incomplete verification engines. *Journal of Automated Reasoning*, 64(7):1523–1552, jul 2020. DOI: 10.1007/s10817-020-09570-z.

[8] LMU München Software and Computational Systems Lab. Collection of verification tasks. https://github.com/sosy-lab/sv-benchmarks. Retrieved: 2020-12-10.

[9] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, jun 1972. DOI: 10.1137/0201010.

[10] Tamás Tegzes. Applying incremental, inductive model checking to software. Bachelor's thesis, Budapest University of Technology and Economics, 2018.

[11] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.