MŰEGYETEM 1782

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

MŰEGYETEM 1782

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 11, 2020

Contents

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 11.

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 támogatják a szoftverhibák kiszűrését 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) can usually only reduce the probability of bugs being present, one can never be completely sure that the software under test works correctly in the cases they did not try. Formal methods, on the other hand, 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 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 which can be proven by induction. The lemmas together 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 that 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 examples 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. [2] gives an overview of Horn clauses and existing techniques to solving them. 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.

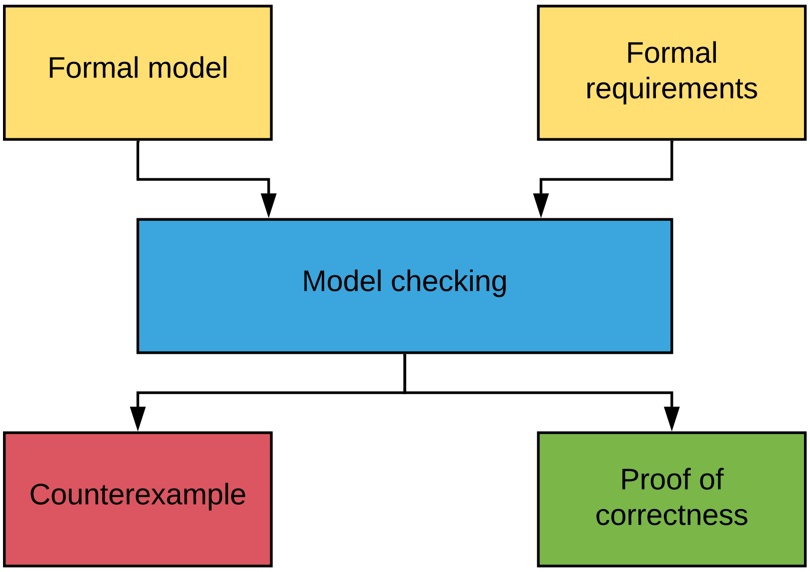

Figure 1.1: Overview of model checking

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. Finally, in Chapter 5 we evaluate the proposed solutions and draw conclusions.

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 F,

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

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 $\alpha = \{x \to 2, y \to (-2)\}$. $\alpha \models (x > y) \vee (z = 0)$, 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, $\alpha \models \phi$, 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 $i \in \mathbb{N}$, $x^{(i)}$ is 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

$u \leftarrow x$;
$z \leftarrow y$;
while $u \neq y$ **do**
  if $x < y$ **then**
    $z \leftarrow z + 1$;
    $u \leftarrow u + 1$;
  else
    $z \leftarrow z - 1$;
    $u \leftarrow u - 1$;
  end
end
`assert`$(x + z = 2 \cdot y)$; `/* Requirement */`

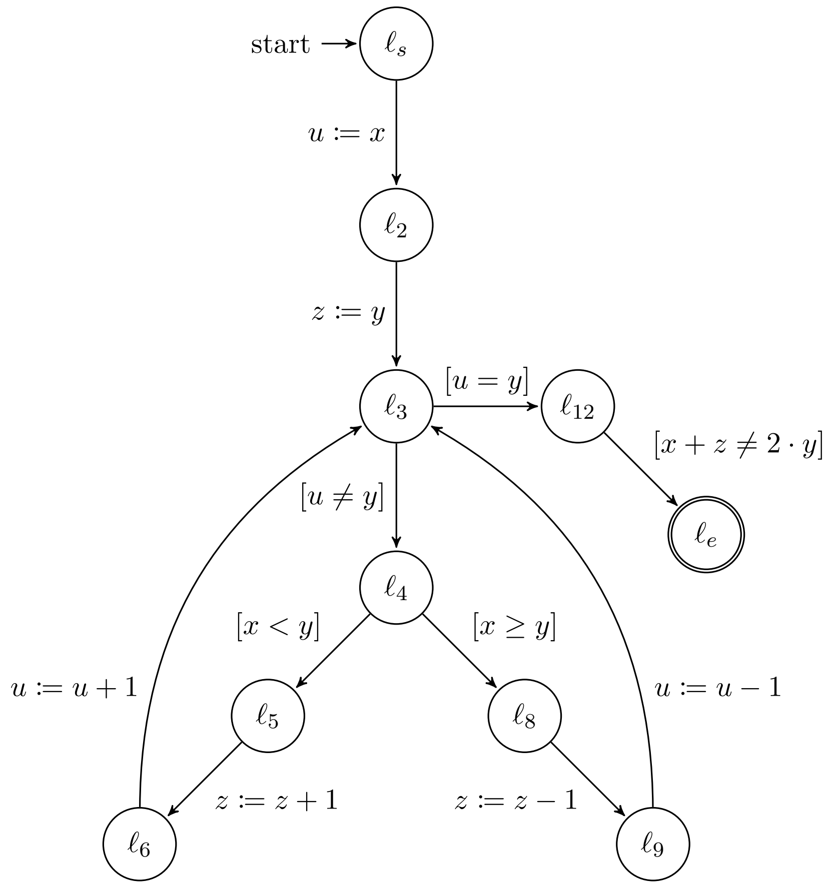

Figure 2.1: A graphical representation of the CFA created from Algorithm 1

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

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

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 $x \coloneqq t$ and statements of the form `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 `havoc` x, the new value of x can be any integer chosen nondeterministically. In the case of $x \coloneqq t$, 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 $\mathrm{same}(X) \mathrel{\hat=} \bigwedge_{x \in X} x' = x$.

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

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.

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

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, 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 $\delta^{(i)}_{s_{i+1}}$ 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, $[a..b] \triangleq \{a, a+1, a+2, \ldots, b\}$

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, for every loop $(\ell_0, s_1, \ell_1)\dots(\ell_{k-1}, s_k, \ell_0)$ in the CFA, there is an $i \in [0..k-1]$ 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

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

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, among the internal locations of a structural edge, every location can only appear at most once and structural edges can be at most $|L| + 1$ long. Since the number of edges is finite, the number of $|L| + 1$ long edge sequences is also finite. Thus, $\mathrm{SE}(\mathcal{A}, N)$ 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 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. During a pass through the loop, both sides either increase or decrease by one, therefore they remain equal.

Starting from a configuration (ℓ_3, α) where $\alpha \models \lambda[\ell_3]$, the error state is unreachable 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 Φ_j ($j \in [1..m]$) such that Φ_j leads from $\ell_{n_{j-1}}$

to ℓ_{n_j}. Let

$$\Phi_j = \left(\ell_{n_{j-1}}, s_{n_{j-1}+1}, \ell_{n_{j-1}+1}\right) \dots \left(\ell_{n_j-1}, s_{n_j}, \ell_{n_j}\right),$$
$$X_j = \left(\bigcup_{i=n_{j-1}}^{n_j} X^{(i)}\right).$$

For all Φ_j, the formula

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

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 $\alpha_i \coloneqq \alpha\left[X^{(i)}\right]$. Since $\alpha_{n_0} \models \lambda[\ell_{n_0}]$ 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:

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

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

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, $L_{\ell_s} \triangleq \top$ and $L_{\ell_e} \triangleq \bot$.

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 λ_ℓ 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:*

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

[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.

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,
- $\beta : \left(\overline{\Delta_\Phi}\right) \nrightarrow \mathbb{Z}$ 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 $\alpha_1, \alpha_2, \ldots, \alpha_{k-1} : X \to \mathbb{Z}$ such that

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

▪

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*

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

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 in a CFA, then every satisfactory invariant system λ must have the following property: If $(\alpha_0 \oplus \beta) \models \lambda[\mathrm{src}(\Phi)]$, then $(\alpha_k \oplus \beta) \models \lambda[\mathrm{tgt}(\Phi)]$. ▪

***Proof*.** Assume indirectly that in a CFA there is a constraint $(\alpha_0, \Phi, \alpha_k, \beta)$ and a satisfactory invariant system λ for which $(\alpha_0 \oplus \beta) \models \lambda[\mathrm{src}(\Phi)]$, 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

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

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

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

Let Φ be the concatenation of $\Phi_1, \Phi_2, \dots, \Phi_n$. 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 $k_i = \mathrm{len}(\Phi_i)$ 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 choose them in reverse order: ζ_n first, then ζ_{n-1}, then 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] $\alpha_{i_1}, \alpha_{i_2}, \dots, \alpha_{i_{k_i-1}} : X \to \mathbb{Z}$, 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 strongly connected component. Every strongly connected component has a loop in it. ▪

The structure of a CFA is a directed graph where the vertices are the 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 of the control flow graph. 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.

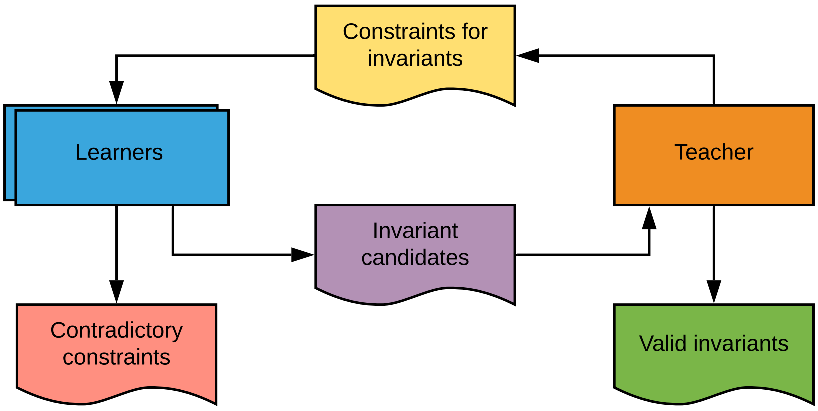

Figure 3.1: Overview of the algorithm architecture

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

$N \leftarrow \varnothing$;
`sccs` $\leftarrow$ the SCCs Tarjan's algorithm finds in the CFG as sets of vertices;
repeat
    `scc` $\leftarrow$ an element of `sccs`;
    `sccs` $\leftarrow$ `sccs` $\setminus$ `scc`;
    $\ell \leftarrow$ a vertex (location) in `scc`;
    if *there is an edge leading from ℓ to itself or another vertex in* `scc` **then**
        $N \leftarrow N \cup \{\ell\}$;
        `newSCCs` $\leftarrow$ the SCCs Tarjan's algorithm finds in the subgraph induced by `scc` $\setminus \{\ell\}$;
        `sccs` $\leftarrow$ `sccs` $\cup$ `newSCCs`;
    else
        `// scc =` $\{\ell\}$ `and` ℓ `does not have a loop edge`
    end
until `sccs` $= \varnothing$;
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 `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, and so on, 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, our algorithm 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 the set of variables X and X'. We introduce x' for each $x \in X$, and add $\mathrm{same}\left(\overline{\Delta_s}\right)$ to make sure that the new value is the same as the old for the variables that s does not refer to. The solver, on the other hand, 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.

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 assign value to only one of z and z', or if it assigns value to neither, then there is a $\hat{\alpha} \subseteq \alpha$ that assigns to them different values. Then $\hat{\alpha} \models \neg\delta_s$, which means $\alpha \not\models \delta_s$. Therefore, α must assign a value to both z and z', even though what 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 connected to the value of x'. It is useful to know when we can choose arbitrary values, and when we cannot and 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', 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 giving requirements for its two functions, $\mathrm{SAT}(\Phi, \lambda)$ and $\mathrm{VAL}(\Phi, \lambda)$. For an invariant system λ and a structural edge

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

(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) $\mathrm{VAL}(\Phi, \lambda) : \left(X \cup X' \cup \dots \cup X^{(\mathrm{len}(\Phi))}\right) \nrightarrow \mathbb{Z}$ 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) $\mathrm{VAL}(\Phi, \lambda) \models \left(\mathrm{sames}(\Phi) \to \left(\lambda[\mathrm{src}(\Phi)] \wedge \delta_\Phi \wedge (\lambda[\mathrm{tgt}(\Phi)])^{(\mathrm{len}(\Phi))}\right)\right)$,

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 these 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 **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. ▪

Proof. The algorithm returns **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 5, γ satisfies R2.1 and R2.2.

Let $k = \mathrm{len}(\Phi)$. Let $\gamma_0, \gamma_1, \dots, \gamma_k : \Delta_\Phi \nrightarrow \mathbb{Z}$ and $\zeta : \overline{\Delta_\Phi} \nrightarrow \mathbb{Z}$ be valuations such that

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

This division is possible because of R2.1.

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

$\mathbf{C} \leftarrow \varnothing$;

forall $\Phi \in \mathrm{SE}(\mathcal{A}, N)$ **do**

  $k \leftarrow \mathrm{len}(\Phi)$;

  if $\mathrm{SAT}(\Phi, \lambda)$ **then**

    $\gamma \leftarrow \mathrm{VAL}(\Phi, \lambda)$; `//` $\gamma : \left(X \cup X' \cup \cdots \cup X^{(k)}\right) \nrightarrow \mathbb{Z}$

    `/* For a constraint we only need the values` γ `assigns to variables in` X `and in` $X^{(k)}$`. */`

    $\alpha_0 \leftarrow \gamma[\Delta_\Phi]$; `//` $\alpha_0 : \Delta_\Phi \nrightarrow \mathbb{Z}$

    $\dot{\alpha}_k \leftarrow \gamma\left[(\Delta_\Phi)^{(k)}\right]$; `//` $\dot{\alpha}_k : (\Delta_\Phi)^{(k)} \nrightarrow \mathbb{Z}$

    Let α_k be such that $(\mathcal{X}_{\alpha_k})^{(k)} = \mathcal{X}_{\dot{\alpha}_k}$ and $\forall x \in \mathcal{X}_{\alpha_k}\ \alpha_k(x) = \dot{\alpha}_k\left(x^{(k)}\right)$;

    `//` $\alpha_k : (\Delta_\Phi) \nrightarrow \mathbb{Z}$

    $\beta \leftarrow \gamma\left[\overline{\Delta_\Phi}\right]$; `//` $\beta : \overline{\Delta_\Phi} \nrightarrow \mathbb{Z}$

    `// By R2.1,` $\forall x \in \mathcal{X}_\beta\ \beta(x) = \gamma(x) = \gamma(x') = \cdots = \gamma\left(x^{(k)}\right)$

    $\mathbf{C} \leftarrow \mathbf{C} \cup (\alpha_0, \Phi, \beta, \alpha_k)$;

  end

end

return $\mathbf{C}$; `// Empty if` λ `is satisfactory.`

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

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

To get $\hat{\alpha}_0$, $\hat{\alpha}_k$ and $\hat{\beta}$, we add value assignments to α_0, α_k and β. We will propagate the same value assignments to some of $\gamma_1, \gamma_2, \ldots, \gamma_{k-1}$ as R2.1 requires. For all $i \in [1..k-i]$, let $\hat{\gamma}_i : \Delta_\Phi \to \mathbb{Z}$ be a full valuation such that $\hat{\gamma}_i \subseteq \gamma_i$, and such that the valuation

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

adheres to R2.1. We can achieve that by propagating the new assignments in $\hat{\alpha}_0$ and $\hat{\alpha}_k$ as the same $\left(\overline{\Delta_{s_i}}\right)$ atoms require it. 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.

Then $\hat{\gamma}$ is a full valuation that adheres to R2.1. By Corollary 3, $\hat{\gamma} \models \delta_\Phi$, since $\theta \subseteq \hat{\gamma}$ implies that $\theta = \hat{\gamma}$. □

3.4 Learners

Learners synthesize invariant systems that adhere to a set of constraints they receive from the teacher. They also have to learn more general properties about the CFA from the constraints in order to eventually synthesize a satisfactory invariant system.

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 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 that adheres to the constraints in the way it classifies the *datapoints* we take from those constraints.

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 (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.

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 $D_1 = (\ell, \{x \to 2, y \to 3\})$, $D_2 = (\ell, \{y \to 5\})$ and $D_3 = (\ell, \{x \to 4\})$ be datapoints, 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. We will now introduce the translation between these concepts.

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 $\mathrm{tgt}(C) = (\mathrm{tgt}(\Phi), \alpha_k \oplus \beta)$. ▪

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 to the variables that cannot change upon executing the structural edge in both datapoints.

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 $\texttt{positiveDeduction}(C, D) = \left(\mathrm{tgt}(\Phi), \alpha_k \oplus \gamma\left[\overline{\Delta_\Phi}\right]\right)$ 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 $\texttt{negativeDeduction}(C, D) = \left(\mathrm{tgt}(\Phi), \alpha_0 \oplus \gamma\left[\overline{\Delta_\Phi}\right]\right)$ 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 $\texttt{positiveDeduction}(C, D_s)$ 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 $\texttt{positiveDeduction}(C, D_1)$ 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 $\texttt{negativeDeduction}(C, D_t)$ 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 $\texttt{negativeDeduction}(C, D_t)$ 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

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

3.4.2 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.

Our algorithm keeps track of which datapoints are *forced true*, and which are *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.

If D is forced true because of Condition 1, then by Definition 17, $\lambda[\ell] \Leftrightarrow \top$, and λ classifies D as `true`.

If D is forced true because of Condition 2, then by Proposition 12, λ classifies D as `true`.

If D is forced true because of Condition 3, then by Proposition 13, λ classifies D as `true`.

There is no other way for D to be forced true. □

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

$$(\hat{\alpha}_0, \Phi_1, \alpha_1, \beta_1), (\hat{\alpha}_1, \Phi_2, \alpha_2, \beta_2), (\hat{\alpha}_2, \Phi_3, \alpha_3, \beta_3), \ldots, (\hat{\alpha}_{n-1}, \Phi_n, \alpha_n, \beta_n) = C_1, C_2, C_3, \ldots, C_n$$

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

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

Thus, $\text{tgt}(C_n)$, 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 $\mathcal{C}$. (ℓ_e, ζ) cannot be forced true because of Condition 1, since $\ell_e \neq \ell_s$. By Corollary 4, it can thus 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 include the constraint from references that use Condition 3: 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 $\text{tgt}(C_1) \subseteq \text{src}(C_2)$, 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.

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

Proof. At line 5, $\text{src}(C)$ is forced true because of Condition 1, and $\text{tgt}(C)$ 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 $\text{CS}(\mathbf{C})$ 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 $\text{CS}(\mathbf{C})$, therefore by Proposition 16, $\text{CS}(\mathbf{C})$ is contradictory.

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

```
D_found ← ∅;
forall (α_0, Φ, α_k, β) ∈ C do
    C ← (α_0, Φ, α_k, β);
    if src(Φ) = ℓ_s then
        D_found ← D_found ∪ {src(C), tgt(C)};   // Condition 1 and Condition 3.
    end
end
D_true ← D_found;
while D_found ≠ ∅ do
    D_found ← ∅;
    forall C ∈ C do
        D_0 ← src(C);
        D_k ← tgt(C);
        if D_k ∉ D_true then
            if D_0 ∈ D_true then
                D_found ← D_found ∪ {D_k};
            else
                forall D ∈ D_true such that D and D_0 are not disjoint do
                    D̂_0 ← D ∩ D_0;                            // Condition 2
                    D̂_k ← positiveDeduction(C, D̂_0);           // Condition 3
                    if D̂_k ∉ D_true then
                        D_found ← D_found ∪ {D̂_k};
                    end
                end
            end
        end
    end
    D_true ← D_true ∪ D_found;
end
if for some valuation γ, (ℓ_e, γ) ∈ D_true then
    return false;                    // Contradictory by Proposition 16.
else
    return true;                     // Consistent.
end
```

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

We will prove by induction that for all $i \in [1..n]$, a datapoint D_i for which $\mathrm{tgt}(C_i) \subseteq D_i$ 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, $\mathrm{tgt}(C'_1)$ is added to $\mathbf{D}_{\texttt{found}}$ at line 5 and then to $\mathbf{D}_{\texttt{true}}$ at line 8. Trivially, $\mathrm{tgt}(C_i) \subseteq \mathrm{tgt}(C'_i)$.

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 $\mathrm{src}(C'_{i+1}) = \mathrm{src}(C_{i+1}) = D_i$, then the algorithm adds $D_{i+1} = \mathrm{tgt}(C'_{i+1}) = \mathrm{tgt}(C_{i+1})$ to $\mathbf{D}_{\texttt{found}}$ at line 16. Otherwise, since $\mathrm{src}(C_{i+1}) \subseteq \mathrm{tgt}(C_i) \subseteq D_i$, the loop at line 18 does get to D_i and then

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

is added to $\mathbf{D}_{\texttt{found}}$ at line 22. Since $\mathrm{src}(C_{i+1}) \subseteq (\mathrm{src}(C'_{i+1}) \cap D_i)$, $\mathrm{tgt}(C_{i+1}) \subseteq D_{i+1}$. 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`. □

3.4.3 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 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.

If D is forced false because of Condition 1, then by Definition 17, $\lambda[\ell] \Leftrightarrow \bot$, and λ classifies D as `false`.

If D is forced false because of Condition 2, then by Proposition 12, λ classifies D as `false`.

If D is forced false because of Condition 3, then by Proposition 14, λ classifies D as `false`.

There is no other way for D to be forced false. □

We will keep track of the set of forced-true and forced-false datapoints for the current constraint system.

Moreover, when one of our learner algorithms searches for an invariant system, it does not consider the classification of every configuration of the CFA, it only considers the classification of a set of datapoints, and only rejects an invariant system if its classification of those datapoints contradicts the constraint system. If a datapoint is classified as `indeterminate`, and we do not check the classification of its subsets, we might miss a contradiction.

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*

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

classifies both $\mathrm{src}(C)$ *and* $\mathrm{tgt}(C)$ *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*

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

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

In case of complex formulae, however, it can be relatively expensive to check if such datapoints exist. We did not find an efficient way to eliminate the problem completely, since we would have to know the exact set of program configurations that an invariant system classifies true and false, and in order to learn that, we would have to make calls to the solver. It is not catastrophic, however, if a learner suggests an invariant system that contradicts the constraints in a non-obvious way. The solver checks the invariant system and will give more specific valuations which will turn into more specific constraints and more specific datapoints for our algorithm to notice the inconsistent classification of. To mitigate this problem, we create a large set of datapoints by combining those available in the given set of constraints.

Moreover, at line 18, iterating over every forced true datapoint may be too expensive.

To solve these problems, we keep track of a set of datapoints such that if D_1 and D_2 are in the set, and they are not disjoint, then $D_1 \cup D_2$ is also in the set. Additionally, for every datapoint D_1, we keep a list of its subsets, datapoints D_2 for which $D_2 \subseteq D_1$.

In Algorithm 5, we present how we keep track of datapoints. This procedure is called whenever a new datapoint emerges either from a new constraint or from a deduction.

Proposition 20. If $\mathbf{D}$ has the property $(D_a \in \mathbf{D} \wedge D_b \in \mathbf{D}) \Rightarrow (D_a \cap D_b) \in \mathbf{D}$ when Algorithm 5 starts, then when the algorithm terminates, $\mathbf{D}$ still has that property and 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 $\mathbf{D}$.

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

while $\mathbf{D}_{\texttt{new}} \neq \varnothing$ **do**
 $D \leftarrow$ some element of $\mathbf{D}_{\texttt{new}}$;
 $\mathbf{D}_{\texttt{new}} \leftarrow \mathbf{D}_{\texttt{new}} \setminus \{D\}$;
 if $D \notin \mathbf{D}$ **then**
  $\texttt{subsets}[D] \leftarrow \{D\}$;
  for $D' \in \mathbf{D}$ **do**
   if $D' \subseteq D$ **then**
    $\texttt{subsets}[D] \leftarrow \texttt{subsets}[D] \cup \{D'\}$;
   else if $D \subseteq D'$ **then**
    $\texttt{subsets}[D'] \leftarrow \texttt{subsets}[D'] \cup \{D\}$;
    if $D' \in \mathbf{D}_{\texttt{true}}$ **then** $\mathbf{D}_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{true}} \cup \{D\}$; `// Condition 2`
    if $D' \in \mathbf{D}_{\texttt{false}}$ **then** $\mathbf{D}_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{false}} \cup \{D\}$; `// Condition 2`
   else if D *and* D' *are not disjoint* **then**
    $\mathbf{D}_{\texttt{new}} \leftarrow \mathbf{D}_{\texttt{new}} \cup \{D \cap D'\}$;
   end
  end
  $\mathbf{D} \leftarrow \mathbf{D} \cup D$;
 end
end

If $D_a \in \mathbf{D} \setminus \mathbf{D}_{\texttt{new}}$ and $D_b \in \mathbf{D} \setminus \mathbf{D}_{\texttt{new}}$, then the precondition ensures that $(D_a \cap D_b) \in \mathbf{D}$.

If $D_a \in \mathbf{D}_{\texttt{new}} \setminus \mathbf{D}$ and $D_b \in \mathbf{D} \setminus \mathbf{D}_{\texttt{new}}$, then when $D = D_a$ at line 2, three cases are possible. (1) 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. (2) If $D_b \subseteq D_a$, then $D_a \cap D_b = D_b$, and D_b is already in $\mathbf{D}$. (3) 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 $\mathbf{D}$.

If $D_a \in \mathbf{D}_{\texttt{new}} \setminus \mathbf{D}$ and $D_b \in \mathbf{D}_{\texttt{new}} \setminus \mathbf{D}$, then whichever gets selected first at line 2 gets added to $\mathbf{D}$ first. Then the previous case 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 some $D \in \mathbf{D}$, $\texttt{subsets}[D] \neq \{D' \in \mathbf{D} \mid D' \subseteq D\}$ 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 5. 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.

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

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

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-false datapoints

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

```
D_found ← ∅;
forall (α_0, Φ, α_k, β) ∈ C do
    C ← (α_0, Φ, α_k, β);
    if src(Φ) = ℓ_s then
        D_found ← D_found ∪ subsets[src(C)] ∪ subsets[tgt(C)];
    end
end
D_true ← D_found ∪ D_prevTrue;
while D_found ≠ ∅ do
    D_found ← ∅;
    forall C ∈ C do
        D_0 ← src(C);
        D_k ← tgt(C);
        if D_k ∉ D_true then
            if D_0 ∈ D_true then
                D_found ← D_found ∪ subsets[D_k];
            else
                forall D̂_0 ∈ subsets[D_0] ∩ D_true do
                    D̂_k ← positiveDeduction(C, D̂_0);
                    if D̂_k ∉ D then
                        Call Algorithm 5 with D_new = {D̂_k};
                    end
                    if D̂_k ∉ D_true then
                        D_found ← D_found ∪ subsets[D̂_k];
                    end
                end
            end
        end
    end
    D_true ← D_true ∪ D_found;
end
return D_true;
```

$\mathrm{src}(\Phi_1) = \ell_s$, $\mathrm{tgt}(\Phi_1) = \mathrm{src}(\Phi_2)$, $\mathrm{tgt}(\Phi_2) = \mathrm{src}(\Phi_1)$ 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, deduce

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

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, 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, then 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. $\square$

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

Proof. Assume that there is a forced-true datapoint $D \in \mathbf{D} \setminus \mathbf{D}_{\texttt{true}}$. If it is forced true because of Condition 1, the algorithm adds it at line 5.

If it is forced true because $D \subseteq D'$ and D' is forced true (Condition 2), then $D \in \texttt{subsets}[D']$, and the algorithm also adds D when it adds D'.

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

Therefore, we have a contradiction. $\square$

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 eventually D is selected at line 18, and $\texttt{positiveDeduction}\left(\hat{C}, D\right)$ is added at line line 24. The datapoint $\texttt{positiveDeduction}\left(\hat{C}, D\right)$ suffices as $\tilde{D}$, because $\texttt{positiveDeduction}(C, D) \subseteq \texttt{positiveDeduction}\left(\hat{C}, D\right)$. $\square$

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 $\mathrm{CS}(\mathbf{C})$ ▪

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

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, 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, then 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. $\square$

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
Input: $\mathbf{D}_{\texttt{prevFalse}}$, a set of previously calculated forced-false datapoints
Output: $\mathbf{D}_{\texttt{false}}$, the set of forced-false datapoints

$\mathbf{D}_{\texttt{found}} \leftarrow \varnothing$;
forall $(\alpha_0, \Phi, \alpha_k, \beta) \in \mathbf{C}$ **do**
  $C \leftarrow (\alpha_0, \Phi, \alpha_k, \beta)$;
  if $\mathrm{tgt}(\Phi) = \ell_e$ **then**
    $\mathbf{D}_{\texttt{found}} \leftarrow \mathbf{D}_{\texttt{found}} \cup \texttt{subsets}[\mathrm{src}(C)] \cup \texttt{subsets}[\mathrm{tgt}(C)]$;
  end
end
$\mathbf{D}_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{found}} \cup \mathbf{D}_{\texttt{prevFalse}}$;
while $\mathbf{D}_{\texttt{found}} \neq \varnothing$ **do**
  $\mathbf{D}_{\texttt{found}} \leftarrow \varnothing$;
  forall $C \in \mathbf{C}$ **do**
    $D_0 \leftarrow \mathrm{src}(C)$;
    $D_k \leftarrow \mathrm{tgt}(C)$;
    if $D_0 \notin \mathbf{D}_{\texttt{false}}$ **then**
      if $D_k \in \mathbf{D}_{\texttt{false}}$ **then**
        $\mathbf{D}_{\texttt{found}} \leftarrow \mathbf{D}_{\texttt{found}} \cup \texttt{subsets}[D_0]$;
      else
        forall $\hat{D}_k \in \texttt{subsets}[D_k] \cap \mathbf{D}_{\texttt{false}}$ **do**
          $\hat{D}_0 \leftarrow \texttt{negativeDeduction}\left(C, \hat{D}_k\right)$;
          if $\hat{D}_0 \notin \mathbf{D}$ **then**
            Call Algorithm 5 with $\mathbf{D}_{\texttt{new}} = \left\{\hat{D}_0\right\}$;
          end
          if $\hat{D}_0 \notin \mathbf{D}_{\texttt{false}}$ **then**
            $\mathbf{D}_{\texttt{found}} \leftarrow \mathbf{D}_{\texttt{found}} \cup \texttt{subsets}\left[\hat{D}_0\right]$;
          end
        end
      end
    end
  end
  $\mathbf{D}_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{false}} \cup \mathbf{D}_{\texttt{found}}$;
end
return $\mathbf{D}_{\texttt{false}}$

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

Proof. Assume that there is a forced-false datapoint $D \in \mathbf{D} \setminus \mathbf{D}_{\texttt{false}}$. If it is forced false because of Condition 1, the algorithm adds it at line 5.

If it is forced false because $D \subseteq D'$ and D' is forced false (Condition 2), then $D \in \texttt{subsets}[D']$, and the algorithm also adds D when it adds D'.

If it is forced false because there is a constraint C and a forced-false datapoint D' such that $D = \texttt{negativeDeduction}(C, D')$ (Condition 3), then in the first iteration of the loop at line 9 after the algorithm adds D', 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 suggests new candidates.

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.

3.4.4 Simple learner

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

When given a set of constraints $\mathbf{C}$, the simple learner algorithm runs Algorithm 5 for $\mathbf{D}_{\texttt{new}} = \{\mathrm{src}(C) \mid C \in \mathbf{C}\} \cup \{\mathrm{tgt}(C) \mid C \in \mathbf{C}\}$. Then it runs Algorithm 6 to find the set of forced-true datapoints, $\mathbf{D}_{\texttt{true}}$. Finally, it returns the invariant system λ, where

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

The invariant system λ 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 $\mathbf{D}_{\texttt{true}}$. Configurations, on the other hand, are never classified as `indeterminate`.

We describe the process in Algorithm 8.

Corollary 6. 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 6, 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 classified $\tilde{D}$ as well as its subsets as `true`. □

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

if *there is a datapoint* $(\ell_e, \alpha) \in \mathbf{D}_{\texttt{true}}$ **then**
  terminate; `// The constraint system is contradictory`
end
$\lambda[\ell_s] \leftarrow \top$;
$\lambda[\ell_e] \leftarrow \bot$;
for $\ell \in N$ **do**
  $\lambda[\ell] \leftarrow \bigvee_{(\ell,\alpha)\in\mathbf{D}_{\texttt{true}}} (\bigwedge_{x\in\mathcal{X}_\alpha} x = \alpha(x))$; `// Equation 3.2`
end
return λ;

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

$$\begin{aligned}
D_1 =&(\ell_1, \{x \to 2\}),\\
D_2 =&(\ell_2, \{y \to 3, z \to 4\}),\\
D_3 =&(\ell_3, \{x \to 2, z \to 4\}),\\
D_4 =&(\ell_4, \{y \to 3, z \to 16\}),
\end{aligned}$$

Algorithm 8 returns the invariant system λ*, where*

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

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 learn generic properties of the CFA from the constraints.

3.4.5 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 its 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. We use Algo-

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, the set of structural nodes

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

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

for $\ell \in N$ **do**
    $\texttt{candidates}[\ell] \leftarrow \{\xi \in \mathcal{P} \mid \xi$ is relevant to $\ell\}$;
end

$\mathbf{D}_{\texttt{labelled}} \leftarrow \varnothing$; // The set of datapoints labelled true
$\mathbf{D}_{\texttt{toLabel}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{true}} \mid \ell \neq \ell_s\}$; // Datapoints yet to be labelled true
repeat
    for $\ell \in N$ **do**
        $\texttt{toRemove} \leftarrow \varnothing$;
        for $\xi \in \texttt{candidates}[\ell]$ **do**
            for $(\ell, \alpha) \in \mathbf{D}_{\texttt{toLabel}}$ **do**
                if $\alpha \not\models \xi$ **then** we must remove ξ to classify (ℓ, α) as $\texttt{true}$
                    $\texttt{toRemove} \leftarrow \texttt{toRemove} \cup \{\xi\}$;
                end
            end
        end
        $\texttt{candidates}[\ell] \leftarrow \texttt{candidates}[\ell] \setminus \texttt{toRemove}$;
        for $(\ell, \alpha) \in \mathbf{D} \setminus \mathbf{D}_{\texttt{labelled}}$ **do**
            if $\alpha \models \bigwedge(\texttt{candidates}[\ell])$ **then** this algorithm must always classify (ℓ, α) as $\texttt{true}$
                $\mathbf{D}_{\texttt{labelled}} \leftarrow \mathbf{D}_{\texttt{labelled}} \cup \texttt{subsets}[(\ell, \alpha)]$;
            end
        end
    end
    // Which datapoints get forced true by datapoints in $\mathbf{D}_{\texttt{labelled}}$?
    $\mathbf{D}'_{\texttt{toLabel}} \leftarrow$ the result of Algorithm 6 with $\mathbf{D}_{\texttt{prevTrue}} = \mathbf{D}_{\texttt{labelled}}$;
    if *there is a datapoint* $(\ell_e, \alpha) \in \mathbf{D}'_{\texttt{toLabel}}$ **then**
        terminate; // The learner cannot express an invariant
    end
    $\mathbf{D}_{\texttt{toLabel}} = \{(\ell, \alpha) \in \mathbf{D}'_{\texttt{toLabel}} \setminus \mathbf{D}_{\texttt{labelled}} \mid \ell \neq \ell_s\}$;
until $\mathbf{D}_{\texttt{toLabel}} = \varnothing$;
$\lambda[\ell_s] \leftarrow \top$;
$\lambda[\ell_e] \leftarrow \bot$;
for $\ell \in N$ **do**
    $\lambda[\ell] \leftarrow \bigwedge(\texttt{candidates}[\ell])$
end
return λ;

rithm 6 to determine the set of datapoints that would be forced true, if the datapoints we labelled `true` were forced true. Since we are not necessarily using only forced-true datapoints as its input, the output may contain datapoints that are not forced true by Definition 30. However, Algorithm 9 must classify these datapoints as `true`. If there is a datapoint with the final location in the output, 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.

We consider a predicate ξ relevant to a structural node ℓ if there is a constraint C such that $\mathrm{src}(C) = (\ell, \alpha)$ and $\alpha \not\models \xi$. This means that by adding ξ to the conjunction, we can rule out at least one datapoint ($D \subseteq \mathrm{src}(C)$) 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 $(\ell, \alpha) \subseteq \mathrm{src}(C)$ and $\alpha \not\models \xi$. ▪

The invariant system that Algorithm 9 returns at line 34 classifies the datapoints in $\mathbf{D}$ such that it adheres to $\mathrm{CS}(\mathbf{C})$.

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, $\alpha \models (\bigwedge(\texttt{candidates}[\ell]))$. ☐

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 $C \in \mathrm{CS}(\mathbf{C})$ for which $D \subseteq \mathrm{src}(C)$, 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 $C \in \mathrm{CS}(\mathbf{C})$ for which $D \subseteq \mathrm{src}(C)$, 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 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 `positiveDeduction`$(C, D) \subseteq D'$ as `true`. □

Algorithm 9 does not always return an invariant system. We will show that that only happens when there is no invariant system that adheres to $\mathrm{CS}(\mathbf{C})$ and only uses conjunctions of subsets of $\mathcal{P}$. We assume that the constraint system is not contradictory, since the input $\mathbf{D}_{\texttt{true}}$ is acquired using Algorithm 6, and if $\ell_e \in \mathbf{D}_{\texttt{true}}$ there is no point invoking Algorithm 9.

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

Proof. Similarly to Corollary 4, we can trace back every datapoint in $\mathbf{D}'_{\texttt{toLabel}}$ at line 23 to a datapoint in $\mathbf{D}_{\texttt{labelled}}$ or a datapoint whose location is ℓ_s. As shown in Proposition 15, the classification must follow the reasoning behind the traces. 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 $\mathbf{D}_{\texttt{labelled}}$, and we must show that there are no datapoints unnecessarily in $\mathbf{D}_{\texttt{labelled}}$.

Datapoints are added to $\mathbf{D}_{\texttt{labelled}}$ because the algorithm removes predicates from `candidates`$[\ell]$ for some structural node ℓ. There are two reasons why the algorithm removes predicates: because they are irrelevant to the structural node or because they contradict with a datapoint that the algorithm must label as `true`.

It is not possible that the datapoint (ℓ_e, α) at line 24 can be traced back to a datapoint $(\ell, \beta) \in \mathbf{D}_{\texttt{labelled}}$ that could be excluded using an irrelevant predicate, because since $\ell \neq \ell_e$, there would be a constraint in the trace, which would make any predicate ξ for which $\beta \not\models \xi$ relevant.

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, the algorithm must label every datapoint in $\mathbf{D}_{\texttt{toLabel}}$ as `true`. If $\xi \in$ `candidates`$[\ell]$, then $\alpha \not\models \bigwedge($`candidates`$[\ell])$, because $\alpha \not\models \xi$. Therefore, the algorithm must remove ξ—it cannot be a part of $\lambda[\ell]$ for any solution where it is a conjunction.

These removals cause other datapoints to be added to $\mathbf{D}_{\texttt{labelled}}$. Since the removals were necessary, these datapoints are also necessarily labelled as `true`.

Then at line 23, the algorithm must also label the datapoints in $\mathbf{D}'_{\texttt{toLabel}}$ as `true`, since it would violate the constraints otherwise. Therefore, in the next iteration, the algorithm has to label the datapoints in $\mathbf{D}_{\texttt{toLabel}}$ as `true`, and our reasoning can be repeated.

We have shown that whenever the algorithm removes a predicate, leaving it in would not be useful, or it would violate the constraints. □

Example 17. *Let $\mathcal{P} = \{(x = 3), (x < y), (y \equiv 3 \pmod{10})\}$. Let there be structural edges Φ_1 leading from ℓ_s to ℓ_1, Φ_2 leading from ℓ_1 to ℓ_2 and Φ_3 leading from ℓ_2 to ℓ_e, and* *constraints C_1, C_2, 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

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

3.4.6 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 of datapoints is not fixed (they are not forced true or forced false), and 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 (ℓ, α).

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 generalized 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 and 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 $(\ell, \beta), (\ell, \gamma) \subseteq (\ell, \alpha)$ 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 concrete constraints that highlight that specific problem.

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 sets 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 $(\mathbf{D}_{\mathtt{whole}}, \mathbf{D}_{\mathtt{split}}, M, \varphi)$. The set of datapoints that the decision tree sends to the node are $\mathbf{D}_{\mathtt{whole}} \cup \mathbf{D}_{\mathtt{split}} \subseteq \mathbf{D}$. Every subset of datapoints in $\mathbf{D}_{\mathtt{whole}}$ is sent to this node, while datapoints in $\mathbf{D}_{\mathtt{split}}$ 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

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, the set of structural nodes
Output: λ, an invariant system

$\lambda[\ell_s] \leftarrow \top$;
for $\ell \in N \cup \{\ell_e\}$ **do**
  $\lambda[\ell] \leftarrow \bot$;
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. */
`toProcess` $\leftarrow \{(\mathbf{D}, \varnothing, N, \top)\}$;
while `toProcess` $\neq \varnothing$ **do**
  $(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \leftarrow$ one of the elements of `toProcess`;
  `toProcess` $\leftarrow$ `toProcess` $\setminus \{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)\}$;
  `label` $\leftarrow$ the result of trying to label the datapoints with Algorithm 11;
  if `label` $=$ `true` **then**
    for $\ell \in M$ **do**
      $\lambda[\ell] \leftarrow \lambda[\ell] \vee \varphi$;
    end
  else if `label` $=$ `null` **then**
    `decision` $\leftarrow$ the best splitting decision found by Algorithm 13;
    if `decision` *is a set of structural nodes* Z **then**
      $\varphi_{\texttt{left}} \leftarrow \varphi$;
      $\varphi_{\texttt{right}} \leftarrow \varphi$;
      $M_{\texttt{left}} \leftarrow M \cap Z$;
      $M_{\texttt{right}} \leftarrow M \setminus Z$;
      $\mathbf{D}_{\texttt{left}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \ell \in Z\}$;
      $\mathbf{D}_{\texttt{right}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \ell \notin Z\}$;
    else // `decision is the predicate` ξ
      $\varphi_{\texttt{left}} \leftarrow \varphi \wedge \xi$;
      $\varphi_{\texttt{right}} \leftarrow \varphi \wedge \neg\xi$;
      $M_{\texttt{left}} \leftarrow M$;
      $M_{\texttt{right}} \leftarrow M$;
      $\mathbf{D}_{\texttt{left}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \alpha \not\models \neg\xi\}$;
      $\mathbf{D}_{\texttt{right}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}} \mid \alpha \not\models \xi\}$;
    end
    $\mathbf{D}_{\texttt{leftWhole}} \leftarrow (\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{whole}}) \setminus \mathbf{D}_{\texttt{right}}$;
    $\mathbf{D}_{\texttt{leftSplit}} \leftarrow (\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{split}}) \cup (\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{right}})$;
    $\mathbf{D}_{\texttt{rightWhole}} \leftarrow (\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{whole}}) \setminus \mathbf{D}_{\texttt{left}}$;
    $\mathbf{D}_{\texttt{rightSplit}} \leftarrow (\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{split}}) \cup (\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{left}})$;
    `toProcess` $\leftarrow$ `toProcess` $\cup \{(\mathbf{D}_{\texttt{leftWhole}}, \mathbf{D}_{\texttt{leftSplit}}, M_{\texttt{left}}, \varphi_{\texttt{left}})\}$;
    `toProcess` $\leftarrow$ `toProcess` $\cup \{(\mathbf{D}_{\texttt{rightWhole}}, \mathbf{D}_{\texttt{rightSplit}}, M_{\texttt{right}}, \varphi_{\texttt{right}})\}$;
  end
end
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

$\mathbf{D}_{\texttt{all}} \leftarrow \mathbf{D}_{\texttt{whole}} \cup \mathbf{D}_{\texttt{split}}$;
if $\mathbf{D}_{\texttt{all}} \subseteq \mathbf{D}_{\texttt{true}}$ **then**
    return `true`;
end
if $\mathbf{D}_{\texttt{all}} \subseteq \mathbf{D}_{\texttt{false}}$ **then**
    return `false`;
end
if $\mathbf{D}_{\texttt{all}} \cap \mathbf{D}_{\texttt{false}} = \varnothing$ **then**
    Run Algorithm 12 with `label` = `true` to determine if the datapoints can be labelled `true`;
    if *they can* **then**
        return `true`;
    end
end
if $\mathbf{D}_{\texttt{all}} \cap \mathbf{D}_{\texttt{true}} = \varnothing$ **then**
    Run Algorithm 12 with `label` = `false` to determine if the datapoints can be labelled `false`;
    if *they can* **then**
        return `false`;
    end
end
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

```
if $\texttt{label} = \texttt{true}$ then
    $\mathbf{D}'_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{true}} \cup \mathbf{D}_{\texttt{whole}}$;
    $\mathbf{D}'_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{false}}$;
else
    $\mathbf{D}'_{\texttt{true}} \leftarrow \mathbf{D}_{\texttt{true}}$;
    $\mathbf{D}'_{\texttt{false}} \leftarrow \mathbf{D}_{\texttt{false}} \cup \mathbf{D}_{\texttt{whole}}$;
end
repeat
    $\mathbf{D}'_{\texttt{true}} \leftarrow$ the result of Algorithm 6 with $\mathbf{D}_{\texttt{prevTrue}} = \mathbf{D}'_{\texttt{true}}$;
    $\mathbf{D}'_{\texttt{false}} \leftarrow$ the result of Algorithm 7 with $\mathbf{D}_{\texttt{prevFalse}} = \mathbf{D}'_{\texttt{false}}$;
    $\mathbf{D}_{\texttt{new}} \leftarrow$ the new datapoints added to $\mathbf{D}$ by the previous two lines;
    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
        return false;        // The labelling contradicts the constraints
    else if $\texttt{label} = \texttt{true}$ then
        if $\mathbf{D}_{\texttt{split}} \cap \mathbf{D}'_{\texttt{false}} \neq \varnothing$ then return false;
    else
        if $\mathbf{D}_{\texttt{split}} \cap \mathbf{D}'_{\texttt{true}} \neq \varnothing$ then return false;
    end
    foreach $(\ell, \alpha) \in \mathbf{D}_{\texttt{new}}$ do
        relTP $\leftarrow \{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \in \texttt{toProcess} \mid \ell \in M$ and $\alpha \not\models \neg\varphi\}$;
        if $|\texttt{relTP}| = 1$ then
            $(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \leftarrow$ the single element in relTP;
            toProcess $\leftarrow$ toProcess $\setminus \{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)\}$;
            toProcess $\leftarrow$ toProcess $\cup \{(\mathbf{D}_{\texttt{whole}} \cup \{(\ell, \alpha)\}, \mathbf{D}_{\texttt{split}}, M, \varphi)\}$;
        else if $|\texttt{relTP}| > 1$ then
            for $(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi) \in \texttt{relTP}$ do
                toProcess $\leftarrow$ toProcess $\setminus \{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}}, M, \varphi)\}$;
                toProcess $\leftarrow$ toProcess $\cup \{(\mathbf{D}_{\texttt{whole}}, \mathbf{D}_{\texttt{split}} \cup \{(\ell, \alpha)\}, M, \varphi)\}$;
            end
        else if $\alpha \models \lambda[\ell]$ then
            $\mathbf{D}'_{\texttt{true}} \leftarrow \mathbf{D}'_{\texttt{true}} \cup \{(\ell, \alpha)\}$;
        else if $\alpha \models \neg\lambda[\ell]$ then
            $\mathbf{D}'_{\texttt{false}} \leftarrow \mathbf{D}'_{\texttt{false}} \cup \{(\ell, \alpha)\}$;
        end
    end
until $\mathbf{D}_{\texttt{new}} = \varnothing$;
$\mathbf{D}_{\texttt{false}} \leftarrow \mathbf{D}'_{\texttt{false}}$;
$\mathbf{D}_{\texttt{true}} \leftarrow \mathbf{D}'_{\texttt{true}}$;
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 $\xi \in \mathcal{P}$

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. */`
  $Z \leftarrow$ a set of approximately half of the structural nodes among the datapoints
  return Z;
end
$\xi_{\texttt{best}} \leftarrow \texttt{null}$;
$\mathsf{e}_{\texttt{best}} \leftarrow \texttt{null}$;
for $\xi \in \mathcal{P}$ **do**
  $\mathbf{D}_{\texttt{left}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{toSplit}} \mid \alpha \models \xi\}$;
  $\mathbf{D}_{\texttt{right}} \leftarrow \{(\ell, \alpha) \in \mathbf{D}_{\texttt{toSplit}} \mid \alpha \models \neg\xi\}$;
  $\mathsf{e}_{\texttt{left}} \leftarrow \textsc{Impurity}(|\mathbf{D}_{\texttt{left}} \cap \mathbf{D}_{\texttt{true}}|, |\mathbf{D}_{\texttt{left}} \cup \mathbf{D}_{\texttt{false}}|, |\mathbf{D}_{\texttt{left}}|)$;
  $\mathsf{e}_{\texttt{right}} \leftarrow \textsc{Impurity}(|\mathbf{D}_{\texttt{right}} \cap \mathbf{D}_{\texttt{true}}|, |\mathbf{D}_{\texttt{right}} \cup \mathbf{D}_{\texttt{false}}|, |\mathbf{D}_{\texttt{right}}|)$;
  $\mathsf{e} \leftarrow \mathsf{e}_{\texttt{left}} + \mathsf{e}_{\texttt{right}}$;
  if $(\mathsf{e}_{\texttt{best}} = \texttt{null}) \lor (\mathsf{e} < \mathsf{e}_{\texttt{best}})$ **then**
    $\xi_{\texttt{best}} \leftarrow \xi$;
    $\mathsf{e}_{\texttt{best}} \leftarrow \mathsf{e}$;
  end
end
return $\xi_{\texttt{best}}$;

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`, 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. But if the classification is unsuccessful, it creates a branch node with Algorithm 13, and adds its two children to `toProcess`.

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 value 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.

Chapter 4

Implementation

We implemented the discussed algorithms as part of the THETA[11] framework in the Kotlin programming language. 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 found an invariant system satisfactory or gave 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.

LEQPATTERN 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.

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

INTBUILDERPATTERN 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:

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

ModulusPattern extracts the integer expressions from the CFA similarly to IntBuilderPattern, 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 548 models, most of them from the International Competition on Software Verification (SV-COMP) [8]. Of the used models, 95 were unsafe and 453 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 computer in the university cloud with 4 CPU cores and 8 GiB of memory. Swapping was disabled. We stopped the benchmarks after either 2 minutes of wall time has elapsed or the prototype used more than 2 minutes of CPU time, where CPU time is the sum of time each 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.

5.2 Tried configurations

We tried 7 different configurations of the prototype.

C1: SIMPLE-SIMPLE

- SIMPLECOORDINATOR
- SIMPLELEARNER

C2: SIMPLE-DECTREE

- SIMPLECOORDINATOR
- DECISIONTREELEARNER with every predicate pattern (ATOMS, INTLEQ, MODULUS, INTBUILDER)

C3: Simple-Fallback

- SimpleCoordinator
- FallbackLearner
  - SorcarLearner with Atoms predicate pattern
  - DecisionTreeLearner with Atoms and IntLEQ predicate patterns

C4: Simple-RoundRobin

- SimpleCoordinator
- RoundRobinLearner
  - SorcarLearner with every predicate pattern
  - DecisionTreeLearner with every predicate pattern

C5: Multi-FewPreds

- MultiThreadedCoordinator
- 3 teachers
- SimpleLearner
- DecisionTreeLearner with Atoms and IntLEQ predicate patterns
- FallbackLearner
  - SorcarLearner with Atoms predicate pattern
  - DecisionTreeLearner with Atoms and IntLEQ predicate patterns

C6: Multi-DecTree

- MultiThreadedCoordinator
- 4 teachers
- DecisionTreeLearner with Atoms predicate pattern
- DecisionTreeLearner with IntLEQ predicate pattern
- DecisionTreeLearner with Modulus predicate pattern
- DecisionTreeLearner with IntBuilder predicate pattern

C7: Multi-Triple

- MultiThreadedCoordinator
- 3 teachers
- SimpleLearner
- DecisionTreeLearner with every predicate pattern
- FallbackLearner
  - SorcarLearner with every predicate pattern
  - DecisionTreeLearner with every predicate pattern

5.3 Results

260 models were solved by at least one of the configurations. Among those 46 were unsafe and 214 were safe.

Table 5.1 shows the number of solved models by configuration. Simple-Fallback ended up being the best configuration overall. It solved 230 problems. 204 of those were safe models, the SorcarLearner synthesized the invariant system for 185 of them and the DecisionTreeLearner synthesized the invariant system for an additional 59 that the

Configuration	Total	Safe	Unsafe
C1 (SIMPLE-SIMPLE)	65	22	43
C2 (SIMPLE-DECTREE)	181	156	25
C3 (SIMPLE-FALLBACK)	230	204	26
C4 (SIMPLE-ROUNDROBIN)	102	78	24
C5 (MULTI-FEWPREDS)	170	137	33
C6 (MULTI-DECTREE)	180	160	20
C7 (MULTI-TRIPLE)	134	104	30

Table 5.1: Number of models solved by each configuration

C1	C2	C3	C4	C5	C6	C7	Total	Safe	Unsafe
							288	239	49
	✓	✓		✓	✓	✓	52	52	0
		✓					32	32	0
	✓	✓	✓	✓	✓	✓	32	31	1
✓	✓	✓	✓	✓	✓	✓	30	13	17
	✓	✓		✓	✓		22	22	0
	✓	✓	✓		✓		13	13	0
		✓			✓		13	13	0
✓							12	0	12
	✓	✓	✓	✓	✓		9	9	0
✓	✓	✓	✓	✓		✓	5	0	5
✓				✓		✓	5	2	3
	✓	✓					3	3	0
	✓	✓	✓				3	3	0
	✓	✓			✓		2	2	0
			✓				2	2	0
✓				✓			2	0	2
✓						✓	2	2	0
	✓		✓				2	2	0
	✓	✓	✓	✓			2	2	0
✓	✓	✓		✓	✓	✓	2	0	2
✓		✓			✓		2	2	0
				✓			2	0	2
		✓		✓			1	1	0
	✓	✓	✓	✓		✓	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	1	0
✓		✓		✓		✓	1	0	1

Table 5.2: Number of models by which configurations solved them

SorcarLearner was unable to synthesize an invariant for. 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.

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. Our theory is that this allowed it to quickly get new constraints and the larger number of constraints allowed it to find longer error paths faster. 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 multithreaded configurations 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 multithreaded configurations, Multi-DecTree solved the most problems. Of the 160 safe models it solved, the final satisfactory invariant system was synthesized by the learner with the Atoms predicate pattern in 78 cases, by the learner with the IntLEQ pattern in 67 cases, by the learner with the Modulus pattern in 26 cases and the learner with the IntBuilder pattern in 18 cases. However, the 44 cases when either the Modulus or the IntBuilder pattern was used for the invariant synthesis, the synthesized invariant system was trivial, i.e., it assigned the constant $\top$ or $\bot$ formula to every structural node. We can conclude that using these predicate patterns by themselves is not useful for this test suite.

Table 5.2 provides a different breakdown of the results. The first seven columns correspond to the configurations and the rows give the number of models that was solved by exactly the configurations with a ✓ symbol in the row.

We can see that there of the 30 models that the Simple-Fallback configuration did not solve, 10 were safe and 20 were unsafe. Of the 20 unsafe, Simple-Simple solved 18. Combining these two configurations may result in an even better configuration.

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`.