Sliced Path Prefixes: An Effective Method to Enable Refinement Selection

Dirk Beyer, Stefan Löwe, and Philipp Wendler

University of Passau, Germany

Abstract. Automatic software verification relies on constructing, for a given program, an abstract model that is (1) abstract enough to avoid state-space explosion and (2) precise enough to reason about the specification. Counterexample-guided abstraction refinement is a standard technique that suggests to extract information from infeasible error paths, in order to refine the abstract model if it is too imprecise. Existing approaches —including our previous work— do not choose the refinement for a given path systematically. We present a method that generates alternative refinements and allows to systematically choose a suited one. The method takes as input one given infeasible error path and applies a slicing technique to obtain a set of new error paths that are more abstract than the original error path but still infeasible, each for a different reason. The (more abstract) constraints of the new paths can be passed to a standard refinement procedure, in order to obtain a set of possible refinements, one for each new path. Our technique is completely independent from the abstract domain that is used in the program analysis, and does not rely on a certain proof technique, such as SMT solving. We implemented the new algorithm in the verification framework CPAchecker and made our extension publicly available. The experimental evaluation of our technique indicates that there is a wide range of possibilities on how to refine the abstract model for a given error path, and we demonstrate that the choice of which refinement to apply to the abstract model has a significant impact on the verification effectiveness and efficiency.

1 Introduction

In the field of automatic software verification, abstraction is a well-understood and widely-used technique, enabling the successful verification of real-world, industrial programs (cf. [4, 13, 14]). Abstraction makes it possible to omit certain aspects of the concrete semantics that are not necessary to prove or disprove the program's correctness. This may lead to a massive reduction of a program's state space, such that verification becomes feasible within reasonable time and resource limits. For example, Slam [5] uses predicate abstraction [18] for creating an abstract model of the software. One of the current research directions is to invent techniques to automatically find suitable abstractions. An ideal model is abstract

A preliminary version of this article appeared as technical report [12].

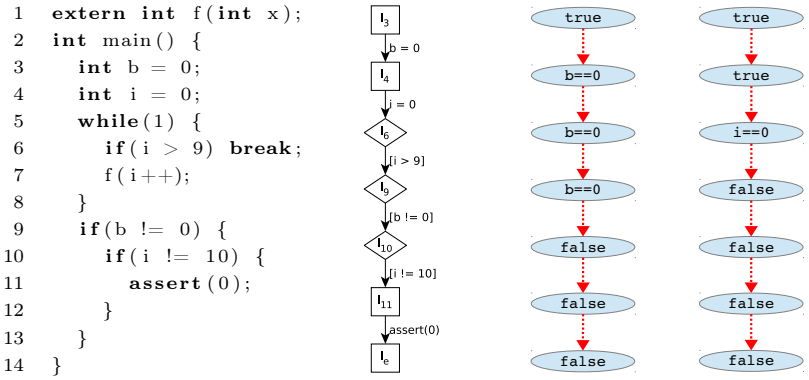


Fig. 1: From left to right, the input program, an infeasible error path, and a "good" and a "bad" interpolant sequence for the infeasible error path

enough to avoid state-space explosion and still contains enough detail to verify the property. Counterexample-guided abstraction refinement (CEGAR) [15] is an automatic technique that starts with a coarse abstraction and iteratively refines an abstract model using *infeasible* error paths. If the analysis does not find an error path in the abstract model, the analysis terminates with the result TRUE. If the analysis finds an error path, the path is checked for feasibility. If this error path is feasible according to the concrete program semantics, then it represents a bug, and the analysis terminates with the result FALSE. However, if the error path is infeasible, then the abstract model was too coarse. In this case, the infeasible error path can be passed to an interpolation engine, which identifies information that is needed to refine the current abstraction, such that the same infeasible error path is excluded in the next CEGAR iterations. CEGAR is successfully used, for example, by the tools SLAM [5], BLAST [7], CPACHECKER [10], and UFO [1].

Craig interpolation [16] is a technique that yields for two contradicting formulas an interpolant formula that contains less information than the first formula, but is still expressive enough to contradict the second formula. In software verification, interpolation was first used for the domain of predicate abstraction [19], and later for value-analysis domains [11]. Independent of the analysis domain, interpolants for path constraints of infeasible error paths can be used to refine abstract models and to eliminate the infeasible error paths in subsequent CEGAR iterations. In this context, it is important to point out that the choice of interpolants is crucial for the performance of the analysis. Figure 1 gives an example: In this program, the analysis will typically find the shown error path, which is infeasible for two different reasons: both the value of `i` and the value of `b` can be used to find a contradiction. In general, it is now beneficial for the verifier to track the value of the boolean variable `b`, and not to track the value of the loop-counter variable `i`, because the latter has many more possible values, and tracking it would usually lead to an expensive unrolling of the loop. Instead, if only variable `b` is tracked, the verifier can conclude the safety of the program without unrolling the loop. Thus, we would like to use for refinement the interpolant sequence shown on the left (with only the boolean variable) and

not the right one (with the loop-counter variable). However, interpolation engines typically do not allow to guide the interpolation process towards "good", or away from "bad", interpolant sequences. The interpolation engines inherently cannot do a better job here: they do not have access to information such as whether a specific variable is a loop counter and should be avoided in the interpolant. Instead, which interpolant is returned depends solely on the internal algorithms of the interpolation engine. This is especially true if the model checker uses an off-the-shelf interpolation engine, which normally cannot be controlled on such a fine-grained level. In this case, the model checker is stuck to what the interpolation engine returns, be it good or bad for the verification process.

Therefore, we present an approach that allows to *guide* the interpolation engine to produce different interpolants, without changing the interpolation engine. To achieve this, we extract from one infeasible error path a set of infeasible sliced paths, each infeasible for a different reason. Each of these sliced paths can be used for interpolation, yielding different interpolant sequences that are all expressive enough to eliminate the original infeasible error path. Our approach fits well into CEGAR (with or without lazy abstraction [20]), because only the refinement component needs customization, and the new approach remains compatible with off-the-shelf interpolation engines.

Contributions. We make the following key contributions: (1) we introduce a domain- and analysis-independent method to extract a set of infeasible sliced paths from infeasible error paths, (2) we prove that interpolants for such a sliced path are also interpolants for the original infeasible error path, (3) we explain that —and how— it is possible to obtain, given a set of infeasible sliced paths, different precisions (interpolants) for the same infeasible error path, and that the choice of the precision makes a significant difference for CEGAR, (4) we implement the presented concepts in the open-source framework for software verification CPAchecker, and (5) we show experimentally that the novel approach to obtain different precisions significantly impacts the effectiveness and efficiency.

While we use interpolation to compute the refined precisions, our method is not bound to interpolation: invariant-generation techniques for refinement such as *path invariants* [8] can equally benefit from the new possibility of choice.

Related Work. The desire to control which interpolants an interpolation engine produces, and trying to make the verification process more efficient by finding good interpolants, is not new. Our goal is to contribute a technique that is independent from the abstract domain that a program analysis uses, and independent from specific properties of interpolation engines.

The first work in this direction suggested to control the *interpolant strength* [17] such that the user can choose between strong and weak interpolants. This approach is unfortunately not implemented in standard interpolation engines. The technique of interpolation abstractions [22], a generalization of term abstraction [2], can be used to guide solvers to pick good interpolants. This is achieved by extending the concrete interpolation problem by so called templates (e.g., terms, formulas, uninterpreted functions with free variables) to obtain a more abstract interpolation problem. An interpolant for the abstract interpolation problem is

also a solution to the concrete interpolation problem. Suitable interpolants can be chosen using a cost function, because these interpolation abstractions form a lattice. In contrast to interpolation abstractions, our approach does not rely on SMT solving and is independent from the interpolation engine and abstract domain, so it is also applicable to, e.g., value and octagon domains.

Path slicing [21] is a technique that was introduced to reduce the burden of the interpolation engine: Before the constraints of the path are given to the interpolation engine, the constraints are weakened by removing facts that are not important for the infeasibility of the error path, i.e., a more abstract error path is constructed. We also make the error path more abstract, but in different directions to obtain different interpolant sequences, from which we can choose one that yields a suitable abstract model. While path slicing is interested in reducing the run time of the *interpolation engine* (by omitting some facts), we are interested in reducing the run time of the *verification engine* (by spending more time on interpolation and selection but creating a better abstract model).

2 Background

Our approach is based on several existing concepts, and in this section we remind the reader of some basic definitions and our previous work in this field [11].

Programs, Control-Flow Automata, States, Paths, Precisions. We restrict the presentation to a simple imperative programming language, where all operations are either assignments or assume operations, and all variables range over integers.[1] A program is represented by a control-flow automaton (CFA). A CFA $A = (L, l_0, G)$ consists of a set L of program locations, which model the program counter, an initial program location $l_0 \in L$, which models the program entry, and a set $G \subseteq L \times Ops \times L$ of control-flow edges, which model the operations that are executed when control flows from one program location to the next. The set of program variables that occur in operations from Ops is denoted by X. A *verification problem* $P = (A, l_e)$ consists of a CFA A, representing the program, and a target program location $l_e \in L$, which represents the specification, i.e., "the program must not reach location l_e".

A *concrete data state* of a program is a variable assignment $cd : X \to \mathbb{Z}$, which assigns to each program variable an integer value; the set of integer values is denoted as $\mathbb{Z}$. A *concrete state* of a program is a pair (l, cd), where $l \in L$ is a program location and cd is a concrete data state. The set of all concrete states of a program is denoted by $\mathcal{C}$, a subset $r \subseteq \mathcal{C}$ is called *region*. Each edge $g \in G$ defines a labeled transition relation $\overset{g}{\to} \subseteq \mathcal{C} \times \{g\} \times \mathcal{C}$. The complete transition relation $\to$ is the union over all control-flow edges: $\to = \bigcup_{g \in G} \overset{g}{\to}$. We write $c \overset{g}{\to} c'$ if $(c, g, c') \in \to$, and $c \to c'$ if there exists a g with $c \overset{g}{\to} c'$.

An *abstract data state* represents a region of concrete data states, formally defined as abstract variable assignment. An *abstract variable assignment* is either a partial function $v : X \rightharpoonup \mathbb{Z}$ mapping variables in its definition range to

[1] Our implementation is based on CPAchecker, which operates on C programs; non-recursive function calls are supported.

integer values, or $\bot$, which represents no variable assignment (i.e., no value is possible, similar to the predicate *false* in logic). The special abstract variable assignment $\top = \{\}$ does not map any variable to a value and is used as initial abstract variable assignment in a program analysis. Variables that do not occur in the definition range of an abstract variable assignment are either omitted by purpose for abstraction in the analysis, or the analysis is not able to determine a concrete value (e.g., resulting from an uninitialized variable declaration or from an external function call). For two partial functions f and f', we write $f(x) = y$ for the predicate $(x, y) \in f$, and $f(x) = f'(x)$ for the predicate $\exists c : (f(x) = c) \wedge (f'(x) = c)$. We denote the *definition range* for a partial function f as $\text{def}(f) = \{x \mid \exists y : f(x) = y\}$, and the *restriction* of a partial function f to a new definition range Y as $f_{|Y} = f \cap (Y \times \mathbb{Z})$. An abstract variable assignment v represents the set $[\![v]\!]$ of all concrete data states cd for which v is valid, formally: $[\![\bot]\!] = \{\}$ and for all $v \neq \bot$, $[\![v]\!] = \{cd \mid \forall x \in \text{def}(v) : v(x) = cd(x)\}$. The abstract variable assignment $\bot$ is called *contradicting*. The *implication* for abstract variable assignments is defined as follows: v implies v' (written $v \Rightarrow v'$) if $v = \bot$, or for all variables $x \in \text{def}(v')$ we have $v(x) = v'(x)$. The *conjunction* for abstract variable assignments v and v' is defined as:

$$v \wedge v' = \begin{cases} \bot & \text{if } v = \bot \text{ or } v' = \bot \text{ or } (\exists x \in \text{def}(v) \cap \text{def}(v') : \neg\, v(x) = v'(x)) \\ v \cup v' & \text{otherwise} \end{cases}$$

The *semantics of an operation* $op \in Ops$ is defined by the strongest-post operator $\mathsf{SP}_{op}(\cdot)$: given an abstract variable assignment v, $\mathsf{SP}_{op}(v)$ represents the set of concrete data states that are reachable from the concrete data states in the set $[\![v]\!]$ by executing op. Formally, given an abstract variable assignment v and an assignment operation $x := exp$, we have $\mathsf{SP}_{x:=exp}(v) = \bot$ if $v = \bot$, or $\mathsf{SP}_{x:=exp}(v) = v_{|X \setminus \{x\}} \wedge v_x$ with

$$v_x = \begin{cases} \{(x, c)\} & \text{if } c \in \mathbb{Z} \text{ is the result of the arith. evaluation of expression } exp_{/v} \\ \{\} & \text{otherwise (if } exp_{/v} \text{ cannot be evaluated)} \end{cases}$$

where $exp_{/v}$ denotes the interpretation of expression exp for the abstract variable assignment v. Given an abstract variable assignment v and an assume operation $[p]$, we have $\mathsf{SP}_{[p]}(v) = \bot$ if $v = \bot$ or the predicate $p_{/v}$ is unsatisfiable, or we have $\mathsf{SP}_{[p]}(v) = v \wedge v_p$, with $v_p = \left\{(x, c) \in (X \setminus \text{def}(v) \times \mathbb{Z}) \;\middle|\; p_{/v} \Rightarrow (x = c)\right\}$ and $p_{/v} = p \wedge \bigwedge_{y \in \text{def}(v)} y = v(y)$.

A *path* σ is a sequence $\langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ of pairs of an operation and a location. The path σ is called *program path* if for every i with $1 \leq i \leq n$ there exists a CFA edge $g = (l_{i-1}, op_i, l_i)$ and l_0 is the initial program location, i.e., the path σ represents a syntactic walk through the CFA. The result of appending the pair (op_n, l_n) to a path $\sigma = \langle (op_1, l_1), \ldots, (op_m, l_m) \rangle$ is defined as $\sigma \wedge (op_n, l_n) = \langle (op_1, l_1), \ldots, (op_m, l_m), (op_n, l_n) \rangle$.

Every path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ defines a *constraint sequence* $\gamma_\sigma = \langle op_1, \ldots, op_n \rangle$. The *conjunction* $\gamma \wedge \gamma'$ of two constraint sequences $\gamma = \langle op_1, \ldots, op_n \rangle$ and $\gamma' = \langle op'_1, \ldots, op'_m \rangle$ is defined as their concatenation, i.e., $\gamma \wedge \gamma' = \langle op_1, \ldots, op_n, op'_1, \ldots, op'_m \rangle$, the *implication* of γ and γ' (denoted by $\gamma \Rightarrow \gamma'$) as the implication of their strongest-post assignments $\mathsf{SP}_\gamma(\top) \Rightarrow \mathsf{SP}_{\gamma'}(\top)$, and γ is *contradicting* if $\mathsf{SP}_\gamma(\top) = \bot$. The *semantics of a*

path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ is defined as the successive application of the strongest-post operator to each operation of the corresponding constraint sequence γ_σ: $\mathsf{SP}_{\gamma_\sigma}(v) = \mathsf{SP}_{op_n}(\ldots \mathsf{SP}_{op_1}(v) \ldots)$. The set of concrete program states that result from running a program path σ is represented by the pair $(l_n, \mathsf{SP}_{\gamma_\sigma}(\top))$, where $\top$ is the initial abstract variable assignment. A path σ is *feasible* if $\mathsf{SP}_{\gamma_\sigma}(\top)$ is not contradicting, i.e., $\mathsf{SP}_{\gamma_\sigma}(\top) \neq \bot$. A concrete state (l_n, cd_n) is *reachable*, denoted by $(l_n, cd_n) \in \mathit{Reach}$, if there exists a feasible program path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ with $cd_n \in [\![\mathsf{SP}_{\gamma_\sigma}(\top)]\!]$. A location l is reachable if there exists a concrete data state cd such that (l, cd) is reachable. A program is *safe* (the specification is satisfied) if l_e is not reachable. A program path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_e) \rangle$, which ends in l_e, is called *error path*.

The *precision* is a function $\pi : L \to 2^{\Pi}$, where Π depends on the abstract domain that is used by the analysis. It assigns to each program location some analysis-dependent information that defines the level of abstraction of the analysis. For example, if using predicate abstraction, the set Π is a set of predicates over program variables. If using a value domain, the set Π is the set X of program variables, and a precision defines which program variables should be tracked by the analysis at which program location.

Counterexample-Guided Abstraction Refinement (CEGAR). CEGAR, a technique for automatic iterative refinement of an abstract model [15], is based on three concepts: (1) a *precision*, which determines the current level of abstraction, (2) a *feasibility check*, which decides if an error path (counterexample) is feasible, and (3) a *refinement* procedure, which takes as input an infeasible error path and extracts a precision to refine the abstract model such that the infeasible error path is eliminated from further exploration. Algorithm 1 shows an instantiation of the CEGAR algorithm. It uses the CPA algorithm [9, 11] for program analysis with dynamic precision adjustment and an abstract domain that is formalized as a configurable program analysis (CPA) with dynamic precision adjustment $\mathbb{D}$. The CPA uses a set E of abstract states and a set $L \to 2^{\Pi}$ of precisions. The analysis algorithm computes the sets reached and waitlist, which represent the current reachable abstract states with precisions and the frontier, respectively. The analysis algorithm is run first with π_0 as coarse initial precision (usually $\pi_0(l) = \{\}$ for all $l \in L$). If all program states have been exhaustively checked, indicated by an empty waitlist, and no error was reached then the CEGAR algorithm terminates and reports TRUE (program is safe). If the CPA algorithm finds an error in the abstract state space, then it stops and returns the yet incomplete sets reached and waitlist. Now the corresponding abstract error path is extracted from the set reached, using the procedure ExtractErrorPath, and passed to the procedure IsFeasible for the *feasibility check*. If the abstract error path is feasible, meaning there exists a corresponding concrete error path, then this error path represents a violation of the specification and the algorithm terminates, reporting FALSE. If the error path is infeasible, i.e., is not corresponding to a concrete program path, then the precision was too coarse and needs to be refined. The *refinement* step is performed by procedure Refine (cf. Alg. 2) which returns a precision π that makes the analysis strong enough to exclude the infeasible

Algorithm 1 CEGAR($\mathbb{D}, e_0, \pi_0$), cf. [11]

Input: a CPA with dynamic precision adjustment $\mathbb{D}$ and
an initial abstract state $e_0 \in E$ with precision $\pi_0 \in (L \to 2^\Pi)$
Output: verification result TRUE (property holds) or FALSE
Variables: a set reached of elements of $E \times (L \to 2^\Pi)$,
a set waitlist of elements of $E \times (L \to 2^\Pi)$, and
an error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$

```
 1: reached := {(e_0, π_0)}; waitlist := {(e_0, π_0)}; π := π_0
 2: while true do
 3:    (reached, waitlist) := CPA(D, reached, waitlist)
 4:    if waitlist = {} then
 5:       return TRUE
 6:    else
 7:       σ := ExtractErrorPath(reached)
 8:       if IsFeasible(σ) then    // error path is feasible: report bug
 9:          return FALSE
10:       else    // error path is infeasible: refine and restart
11:          π(l) := π(l) ∪ Refine(σ)(l), for all program locations l
12:          reached := {(e_0, π)}; waitlist := {(e_0, π)}
```

Algorithm 2 Refine(σ)

Input: an infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$
Output: a precision $\pi \in L \to 2^\Pi$
Variables: a constraint sequence Γ

```
1: Γ := ⟨⟩
2: π(l) := {}, for all program locations l
3: for i := 1 to n − 1 do
4:    γ+ := ⟨op_{i+1}, ..., op_n⟩
5:    Γ := Interpolate(Γ ∧ ⟨op_i⟩, γ+)    // inductive interpolation
6:    π(l_i) := ExtractPrecision(Γ)        // create precision based on Γ
7: return π
```

error path from future state-space explorations. This returned precision is used to extend the current precision of the CPA algorithm, which is started in CEGAR's next iteration and re-computes the sets reached and waitlist based on the new, refined precision. CEGAR is often used with lazy abstraction [20] so that after refining, instead of the whole state space, only some parts of reached and waitlist are removed, and re-explored with the new precision.

Interpolation for Constraint Sequences. An *interpolant* for two constraint sequences γ^- and γ^+, such that $\gamma^- \wedge \gamma^+$ is contradicting, is a constraint sequence Γ for which 1) the implication $\gamma^- \Rightarrow \Gamma$ holds, 2) the conjunction $\Gamma \wedge \gamma^+$ is contradicting, and 3) the interpolant Γ contains in its constraints only variables that occur in both γ^- and γ^+ [11].

Next, we introduce our novel approach, which extracts from one infeasible error path a set of infeasible sliced path prefixes. Sect. 4 then uses this method to extend the procedure Refine to perform precision extraction on a set of infeasible sliced prefixes, offering to select the most suitable precision from several choices.

3 Sliced Prefixes

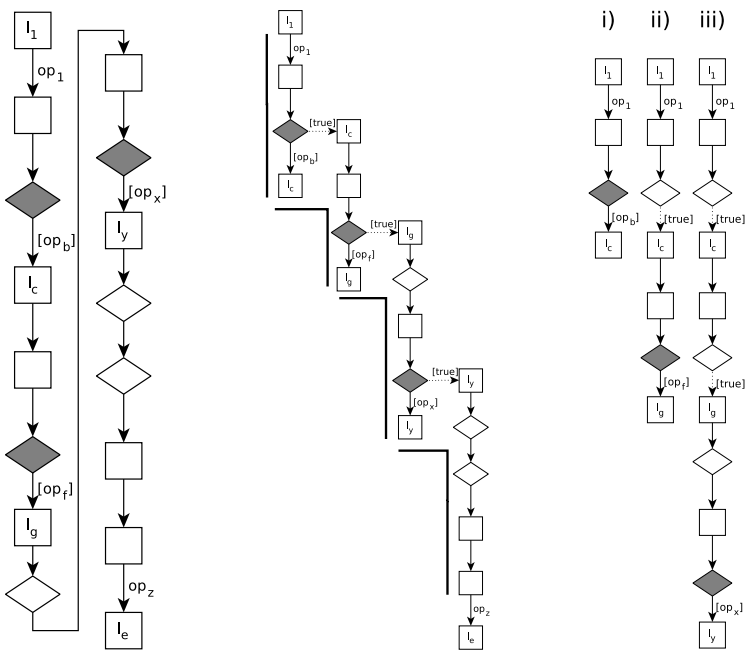
Infeasible Sliced Prefixes. A CEGAR-based analysis encounters an infeasible error path if the precision is too coarse. An infeasible error path contains at least one assume operation for which the reachability algorithm computes a non-contradicting *abstract* successor based on the current precision, but computes a contradicting successor if the *concrete* semantics of the program is used. Every infeasible error path contains at least one such contradicting assume operation, but often, there exist several independently contradicting assume operations in an infeasible error path, which leads to the notion of infeasible sliced prefixes: A path $\phi = \langle (op_1, l_1), \ldots, (op_w, l_w) \rangle$ is a *sliced prefix* for a program path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ if $w \leq n$ and for all $1 \leq i \leq w$, we have $\phi.l_i = \sigma.l_i$ and $(\phi.op_i = \sigma.op_i$ or $(\phi.op_i = [true]$ and $\sigma.op_i$ is assume op)), i.e., a sliced prefix results from a program path by omitting pairs of operations and locations from the end, and possibly replacing some assume operations by no-op operations. If a sliced prefix for σ is infeasible, then σ is infeasible.

Extracting Infeasible Sliced Prefixes from an Infeasible Error Path. Algorithm 3 extracts from an infeasible error path a set of infeasible sliced prefixes. The algorithm iterates through the given infeasible error path σ. It keeps incrementing a feasible sliced prefix σ_f that contains all operations from σ that were seen so far, except contradicting assume operations, which were replaced by no-op operations. Thus, σ_f is always feasible. For every element (op, l) from the original path σ (iterating in order from the first to the last pair), it is checked whether it contradicts σ_f, which is the case if the result of the strongest-post operator for the path $\sigma_f \wedge (op, l)$ is contradicting (denoted by $\bot$). If so, the algorithm has found a new infeasible sliced prefix, which is collected in the set Σ of infeasible sliced prefixes. The feasible sliced prefix σ_f is extended either by a no-op operation (Line 7) or by the current operation (Line 9). When the algorithm terminates, which is guaranteed because σ is finite, the set Σ contains infeasible sliced prefixes of σ, one for each 'reason' of infeasibility. There is always at least one infeasible sliced prefix because σ is infeasible.

Algorithm 3 ExtractSlicedPrefixes(σ)

Input: an infeasible path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$
Output: a non-empty set $\Sigma = \{\sigma_1, \ldots, \sigma_n\}$ of infeasible sliced prefixes of σ
Variables: a path σ_f that is always feasible

```
1: Σ := {}
2: σ_f := ⟨⟩
3: for each (op, l) ∈ σ do
4:    if SP_{σ_f ∧ (op,l)}(⊤) = ⊥ then
5:       // add σ_f ∧ (op, l) to the set of infeasible sliced prefixes
6:       Σ := Σ ∪ {σ_f ∧ (op, l)}
7:       σ_f := σ_f ∧ ([true], l)      // append no-op
8:    else
9:       σ_f := σ_f ∧ (op, l)          // append original pair
10: return Σ
```



(a) Error path (b) Sliced-prefix cascade (c) Sliced prefixes

Fig. 2: From one infeasible error path to a set of infeasible sliced prefixes

The sliced prefixes that Alg. 3 returns have some interesting characteristics: (1) Each sliced prefix ϕ starts with the initial operation op_1, and ends with an assume operation that contradicts the previous operations of ϕ, i.e., $\mathsf{SP}_\phi(\top) = \bot$. (2) The i-th sliced prefix, excluding its (final and only) contradicting assume operation and location, is a prefix of the $(i+1)$-st sliced prefix. (3) All sliced prefixes differ from a prefix of the original infeasible error path σ only in their no-op operations.

The visualizations in Fig. 2 capture the details of this process. Figure 2a shows the original error path. Nodes represent program locations and edges represent operations between these locations (assignments to variables or assume operations over variables, the latter denoted with brackets). To allow easier distinction, program locations that are followed by assume operations are drawn as diamonds, while other program locations are drawn as squares. Program locations before contradicting assume operations are drawn with a filled background. The sequence of operations ends in the error state, denoted by l_e. Figure 2b depicts the cascade-like sliced prefixes that the algorithm encounters during its progress. Figure 2c shows the three infeasible sliced prefixes that Alg. 3 returns for this example.

The refinement procedure can now use any of these infeasible sliced prefixes to create interpolation problems, and is not bound to a single, specific interpolant sequence for the original infeasible error path: a refinement selection from different precisions is now possible. The following proposition states that this is a valid refinement process.

Proposition. Let σ be an infeasible error path and ϕ be the i-th infeasible sliced prefix for σ that is extracted by Alg. 3, then all interpolant sequences for ϕ are also interpolant sequences for σ.

Proof. Let $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ and $\phi = \langle (op_1, l_1), \ldots, (op_w, l_w) \rangle$. Let Γ_{ϕ^j} be the j-th interpolant of an interpolant sequence for ϕ, i.e., for the two constraint sequences $\gamma^-_{\phi^j} = \langle op_1, \ldots, op_j \rangle$ and $\gamma^+_{\phi^j} = \langle op_{j+1}, \ldots, op_w \rangle$, with $1 \leq j < w$. Because ϕ is infeasible, the two constraint sequences $\gamma^-_{\phi^j}$ and $\gamma^+_{\phi^j}$ are contradicting, and therefore, Γ_{ϕ^j} exists [11]. The interpolant Γ_{ϕ^j} is also an interpolant for $\gamma^-_{\sigma^j} = \langle op_1, \ldots, op_j \rangle$ and $\gamma^+_{\sigma^j} = \langle op_{j+1}, \ldots, op_n \rangle$, if (1) the implication $\gamma^-_{\sigma^j} \Rightarrow \Gamma_{\phi^j}$ holds, (2) the conjunction $\Gamma_{\phi^j} \wedge \gamma^+_{\sigma^j}$ is contradicting, and (3) the interpolant Γ_{ϕ^j} contains only variables that occur in both $\gamma^-_{\sigma^j}$ and $\gamma^+_{\sigma^j}$. Consider that $\gamma^-_{\phi^j}$ was created from $\gamma^-_{\sigma^j}$ by replacing some assume operations by no-op operations, and that $\gamma^+_{\phi^j}$ was created from $\gamma^+_{\sigma^j}$ by replacing some assume operations by no-op operations and by removing the operations $\langle op_{w+1}, \ldots, op_n \rangle$ at the end. Thus, both $\gamma^-_{\phi^j}$ and $\gamma^+_{\phi^j}$ do not contain any additional constraints (except for no-op operations) than $\gamma^-_{\sigma^j}$ and $\gamma^+_{\sigma^j}$, respectively.

Because Γ_{ϕ^j} is an interpolant for $\gamma^-_{\phi^j}$ and $\gamma^+_{\phi^j}$, we know that $\gamma^-_{\phi^j} \Rightarrow \Gamma_{\phi^j}$ holds, and because $\gamma^-_{\sigma^j}$ can only be stronger than $\gamma^-_{\phi^j}$, Claim (1) follows. The conjunction $\Gamma_{\phi^j} \wedge \gamma^+_{\phi^j}$ is contradicting, and $\gamma^+_{\sigma^j}$ can only be stronger than $\gamma^+_{\phi^j}$. Thus, Claim (2) holds. Because Γ_{ϕ^j} references only variables that occur in both $\gamma^-_{\phi^j}$ and $\gamma^+_{\phi^j}$, which do not contain more variables than $\gamma^-_{\sigma^j}$ and $\gamma^+_{\sigma^j}$, resp., Claim (3) holds.

4 Slice-Based Refinement Selection

Extracting good precisions from the infeasible error paths is key to the CEGAR technique, and the choice of interpolants influences the quality of the precision, and thus, the effectiveness of the analysis algorithm. By using the results introduced in the previous section, the refinement procedure can now be improved by selecting a precision that is derived via interpolation from a selected infeasible sliced prefix.

Slice-based refinement selection extracts from a given infeasible error path not only one single interpolation problem for obtaining a refined precision, but a set of (more abstract) infeasible sliced prefixes and thus, a set of interpolation problems, from which a refined precision can be extracted. The interpolation problems for the extracted paths can be given, one by one, to the interpolation engine, in order to derive interpolants for each sliced prefix individually. Hence, the abstraction refinement of the analysis is no longer dependent on what the interpolation engine produces, but instead it is free to choose from a set of interpolant sequences the one that it finds most suitable. The move from solving a single interpolation problem to solving multiple interpolation problems, and understanding refinement selection as an optimization problem, is a key insight of our novel approach.

Algorithm 4 shows the algorithm for slice-based refinement selection, which is an extension of Alg. 2 in the CEGAR algorithm, allowing *to choose a suitable precision* during the refinement step. First, this algorithm calls ExtractSlicedPrefixes to extract a set of infeasible sliced prefixes. Second, it computes precisions for

Algorithm 4 $\mathsf{Refine}^{+}(\sigma)$

Input: an infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$
Output: a precision $\pi \in L \to 2^{\Pi}$
Variables: a constraint sequence Γ,
a set Σ of infeasible sliced prefixes of σ,
a mapping τ from infeasible sliced prefixes and program locations to precisions

1: $\Sigma := \mathsf{ExtractSlicedPrefixes}(\sigma)$
2: // compute precisions for each infeasible sliced prefix
3: **for each** $\phi_j \in \Sigma$ **do**
4: $\quad \tau(\phi_j) := \mathsf{Refine}(\phi_j)$ // Alg. 2
5: // select suitable sliced prefix (based on the sliced prefixes and their precisions)
6: $\phi_{selected} := \mathsf{SelectSlicedPrefix}(\tau)$
7: // return precision for CEGAR based on selected sliced prefix
8: **return** $\tau(\phi_{selected})$

the sliced prefixes and stores them in the mapping τ. Third, one sliced prefix is chosen by a heuristic (in function SelectSlicedPrefix), and fourth, the precision of the chosen sliced prefix is selected for refinement of the abstract model. The heuristic can decide based on the information contained in the sliced prefixes as well as in the precisions, e.g., which variables are referenced.

Refinement-Selection Heuristics. We regard the problem of finding and selecting a preferable refinement as an independent direction for further research, and here, we restrict ourselves to presenting some ideas for a few refinement-selection heuristics. There are two obvious options for refinement selection that are independent of the actual interpolants. Using the interpolant sequence derived from the very first, i.e., the shortest, infeasible prefix may rule out many similar infeasible error paths. The downside of this choice is that the analysis may have to track information rather early, possibly blowing up the state-space and making the analysis less efficient. The other straight-forward option (similar to counterexample minimization [2]) is to use the longest infeasible sliced prefix (containing the last contradicting assume operation) for computing an interpolant sequence. This may lead to a precision that is local to the error location and does not require refining large parts of the state space at the beginning of the error path. However, it may also lead to a larger number of refinements if many error paths with a common prefix exist. A more advanced strategy is to analyze the domain types [3] of the variables that are referenced in the extracted precision. Each precision can be assigned a score that depends on the domain types of the variables in the precision such that the score of the infeasible sliced prefix is better if its extracted precision references only 'easy' types of variables, e.g., boolean variables, and no integer variables or even loop counters. This allows to focus on variables that are inexpensive to analyze, avoiding loop unrolling where possible, and keeping the size of the abstract state space as small as possible.

As future work, we plan to systematically investigate many different refinement heuristics; such heuristics can be integrated without changing the overall algorithm, by replacing only the function SelectSlicedPrefix in Alg. 4 accordingly.

5 Experiments

We implemented our approach in the open-source verification framework CPAchecker [10], which is available online [2] under the Apache 2.0 license. CPAchecker already provides several abstract domains that can be used for program analysis with CEGAR. We only extended the refinement process to work according to Alg. 4 (Refine$^+$), and did neither change the abstract domains nor the interpolation engines. Our implementation is available in the source-code repository of CPAchecker. The tool, the benchmark programs, the configuration files, and the complete results are available on the supplementary web page [3].

Setup. For benchmarking, we used machines with two Intel Xeon E5-2650v2 eight-core CPUs with 2.6 GHz and 135 GB of memory. We limited each verification run to two CPU cores, 15 min of CPU time, and 15 GB of memory. We measured CPU time and report it rounded to two significant digits. BenchExec [4] was used as benchmarking framework to ensure precise and reproducible results.

Configurations. Out of the several abstract domains that are supported by CPAchecker, we choose the value analysis with refinement [11] for our experiments. We use CPAchecker, tag `cpachecker-1.4.2-slicedPathPrefixes`.

In order to evaluate the potential of our approach, we compare four different heuristics for refinement selection (function SelectSlicedPrefix in Alg. 4): (1) shortest sliced prefix, (2) longest sliced prefix, (3) sliced prefix with best domain-type score, and (4) sliced prefix with worst domain-type score. The domain-type score of a sliced prefix is computed based on the domain types [3] of the variables that occur in the precisions, i.e., variables with a boolean character are preferred over loop counters and other integer variables.

Benchmarks. To present a thorough evaluation of our approach, we need a large number of verification tasks, and thus, we use the repository of SV-COMP [6] as a source of verification tasks. We select all verification tasks that fulfill the following characteristics, which are necessary for a valid evaluation of our approach: (1) the verification tasks relate to reachability properties, because the analysis that we use does not support other properties; (2) the reachability property of the verification tasks does not rely on concurrency, recursion, dynamic data structures or pointers, because the analysis that we use does not support these features; and (3) there is at least one refinement during the analysis with more than one infeasible sliced prefix, i.e., in at least one refinement iteration, a refinement selection is possible. More restrictions are not necessary because our goal is to show that there *exists* a significant difference in effectiveness and efficiency, depending on the choice of which sliced prefix is used for precision refinement. The scope of our experiments is not to evaluate *which* refinement selection is the best. The set of all verification tasks in our experiments contains a total of 2 696 verification tasks.

[2] `http://cpachecker.sosy-lab.org/`
[3] `http://www.sosy-lab.org/~dbeyer/cpa-ref-sel/`
[4] `https://github.com/dbeyer/benchexec`

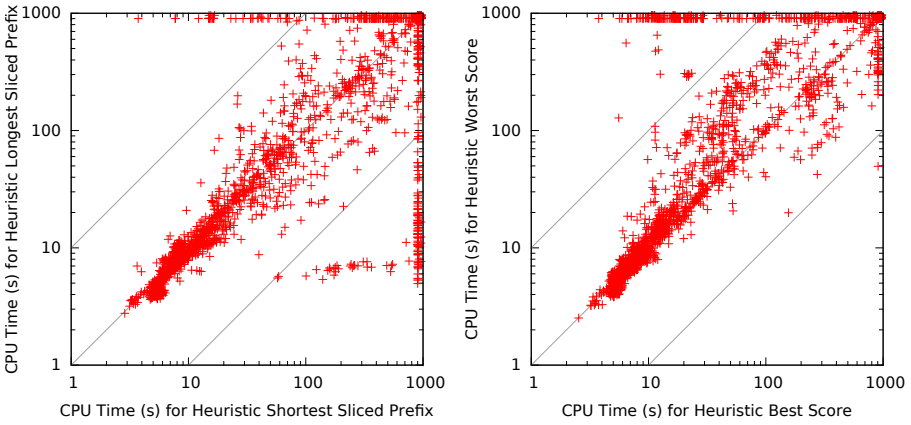
Table 1: Number of solved verification tasks for different heuristics for slice-based refinement selection on different subsets of benchmarks

Heuristic		Sliced-Prefix Length		Score		Oracle		
	# Tasks	Shortest	Longest	Best	Worst	Best	Worst	Diff
DeviceDrivers64	619	326	395	399	319	403	315	88
ECA	1 140	489	512	570	478	611	410	201
ProductLines	597	456	361	402	360	463	353	110
Sequentialized	234	29	22	30	27	30	19	11
All Tasks	2 696	1 369	1 359	1 470	1 252	1 577	1 165	412

Results. Table 1 shows the number of verification tasks that the analysis could solve using refinement selection with one of the four heuristics described above. We also show hypothetical results of a fictional heuristic "Oracle", which, for a given program, always selects the best (or the worst) of the four basic heuristics. In other words, the column "Oracle Best" shows how many tasks could be solved by at least one of the heuristics, and the column "Oracle Worst" shows how many tasks could be solved by all of the heuristics. The difference between these numbers (column "Diff") gives an approximation of the potential of our approach and provides evidence how important refinement selection is. We list the results for the full set of 2 696 verification tasks as well as for several subsets (categories of SV-COMP'15). We consider these categories to be especially interesting because they contain larger programs than the remaining categories and our approach focuses on improving refinements in large programs (with long and complex error paths, and many contradicting assume operations per error path).

The results show that selecting the right refinement can have a significant impact on the effectiveness of an analysis. In our benchmark set there are more than 400 verification tasks for which the choice of the refinement-selection heuristic makes the difference between being able to solve the task and running into a timeout. Without our refinement-selection approach, the choice of the refinement depends solely on the internal algorithm of the interpolation engine, and this potential for improving the analysis would be lost. The results show that none of the presented heuristics is clearly the best. The heuristic that uses the refinement with the best score regarding the domain types of the variables that are contained in the precisions is the best overall (as expected, because it is the only one that systematically tries to select a refinement that hopefully makes it easier for the analysis). However, there are still verification tasks that cannot be solved with this heuristic but with one of the others (as witnessed by the difference between columns "Score Best" and "Oracle Best"). Thus, finding a better refinement-selection heuristics is promising future work.

Figure 3 shows scatter plots for comparing the CPU times of the analysis with two of the four heuristics for slice-based refinement selection. The large number of data points at the top and right borders of the boxes show those results that were solved using one of the heuristics but not by the other. In addition,



(a) Heuristic "Shortest" vs. "Longest" (b) Heuristic "Best Score" vs. "Worst Score"

Fig. 3: Scatter plots comparing the CPU time of the analysis with different heuristics for slice-based refinement selection for all 2 696 verification tasks

one can see that the choice of the refinement-selection heuristic can also have a performance impact of factor more than 10 even for those programs that can be solved by both heuristics (witnessed by the data points in the upper left and lower right corners). This effect also results in a huge performance difference in total: the CPU time for those 1 165 verification tasks that could be solved with all heuristics varies between 110 h (heuristic "Score Worst") and 57 h (heuristic "Score Best"), a potential improvement due to refinement selection of almost 50 %.

6 Conclusion

This paper presents our novel approach of *sliced prefixes* of program paths, which extracts several infeasible sliced prefixes from one single infeasible error path. From any of these infeasible sliced prefixes, an independent interpolation problem can be derived that can be solved by a standard interpolation engine, and the refinement procedure can choose from the resulting interpolant sequences the one that it consideres best for the verification. Our novel approach is independent from the abstract domain (in particular, does not depend on using an SMT solver) and can be combined with any analysis that is based on CEGAR, while previous work on guided interpolation [22] is applicable only to SMT-based approaches. Finally, we demonstrated on a large experimental evaluation on standard verification tasks that the choice, which sliced prefix to take for precision extraction, has a significant impact on the effectiveness and efficiency of the program analysis. In future work, we plan to systematically explore more criteria for ranking sliced prefixes, and then investigate guided techniques for automatically selecting a preferable refinement. Furthermore, we plan to extend our experiments to other abstract domains, such as predicate abstraction and octagons; preliminary results already look promising.

References

1. A. Albarghouthi, Y. Li, A. Gurfinkel, and M. Chechik. Ufo: A framework for abstraction- and interpolation-based software verification. In *Proc. CAV*, LNCS 7358, pages 672–678. Springer, 2012.
2. F. Alberti, R. Bruttomesso, S. Ghilardi, S. Ranise, and N. Sharygina. An extension of lazy abstraction with interpolation for programs with arrays. *Formal Methods in System Design*, 45(1):63–109, 2014.
3. S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. von Rhein. Domain types: Abstract-domain selection based on variable usage. In *Proc. HVC*, LNCS 8244, pages 262–278. Springer, 2013.
4. T. Ball, B. Cook, V. Levin, and S.K. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proc. IFM*, LNCS 2999, pages 1–20. Springer, 2004.
5. T. Ball and S. K. Rajamani. The Slam project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
6. D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Proc. TACAS*, LNCS 9035, pages 401–416. Springer, 2015.
7. D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. The software model checker Blast. *Int. J. Softw. Tools Technol. Transfer*, 9(5-6):505–525, 2007.
8. D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *Proc. PLDI*, pages 300–309. ACM, 2007.
9. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.
10. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
11. D. Beyer and S. Löwe. Explicit-state software model checking based on CEGAR and interpolation. In *Proc. FASE*, LNCS 7793, pages 146–162. Springer, 2013.
12. D. Beyer, S. Löwe, and P. Wendler. Domain-type-guided refinement selection based on sliced path prefixes. Technical Report MIP-1501, University of Passau, January 2015. arXiv:1502.00045.
13. D. Beyer and A. K. Petrenko. Linux driver verification. In *Proc. ISoLA*, LNCS 7610, pages 1–6. Springer, 2012.
14. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proc. PLDI*, pages 196–207. ACM, 2003.
15. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.
16. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.
17. V. D'Silva, D. Kröning, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proc. VMCAI*, LNCS 5944, pages 129–145. Springer, 2010.
18. S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
21. R. Jhala and R. Majumdar. Path slicing. In *Proc. PLDI*, pages 38–47. ACM, 2005.
22. P. Rümmer and P. Subotic. Exploring interpolants. In *Proc. FMCAD*, pages 69–76. IEEE, 2013.