Refinement Selection

Dirk Beyer, Stefan Löwe, and Philipp Wendler

University of Passau, Germany

Abstract. Counterexample-guided abstraction refinement (CEGAR) is a commonly-used approach for the automatic construction of an abstract model of a given program. It uses information learned from infeasible error paths to guide the verification process. We address the problem of selecting *which* information to learn from a given infeasible error path. Previously, we presented a method that can extract a set of sliced path prefixes from a given infeasible error path, each of which can be used for refining the abstract model. We showed that the choice which sliced path prefix is used for refinement significantly impacts effectiveness and efficiency of the analysis. In this work, we extend existing work in three directions: (1) we adopt the idea to generate sliced path prefixes to a new domain, namely one based on predicate abstraction, (2) we define and investigate several promising heuristics for selecting an appropriate refinement, and (3) we enable a completely new combination of a value analysis and a predicate analysis that does not only find out *which information* to learn from an infeasible error path, but automatically decides *which analysis* is best to be refined. These contributions allow a more systematic refinement strategy for CEGAR-based analyses. We implemented the new algorithms in the verification framework CPAchecker and make our work publicly available. In a thorough experimental study, we show that refinement selection often avoids state-space explosion in cases where existing approaches diverge, and that it can be even more powerful if applied on a higher level where it dictates which analysis of a combination is best to be refined.

1 Introduction

Abstraction has proven as an indispensable technique to enable the verification of real-world software (cf. [3, 13, 14]) within reasonable time and resource limits. Slam [4], for example, uses predicate abstraction [20] for creating an abstract model of the software. The success of abstraction in software verification is strongly connected with the advent of the counterexample-guided abstraction refinement (CEGAR) [17] framework, which is nowadays incorporated in many successful software verification tools like Slam [4], Blast [6], CPAchecker [8], and others. This automatic technique proposes to iteratively refine an (initially coarse) abstract model using *infeasible* error paths. For the refinement step of the CEGAR framework Craig interpolation [18] is typically used. Interpolation 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 verification, we can use information from interpolation over an infeasible error path to refine the abstract model [21] and iteratively find a level of abstraction that is strong enough to prove the specification.

```
extern int nondet_int();
extern int f(int x);
int main() {
  int b = nondet_int();
  int i = 0;
  if (b) {
    while(i < 1000) {
      f(i++);
    }
  }
  if(i != 0) {
    assert(!b);
  }
}
```

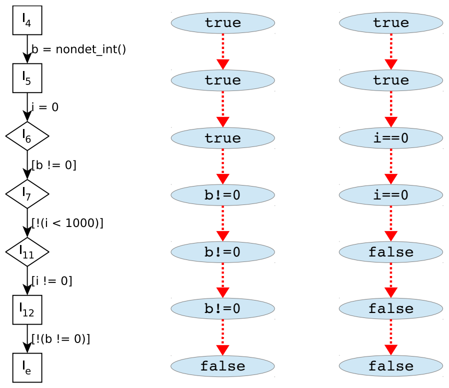

Fig. 1: From left to right, the input program, an infeasible error path, and a "good" and a "bad" interpolant sequence for the infeasible error path

In order to avoid state-space explosion and the divergence of the analysis, care must be taken to keep the precision of the analysis as abstract and concise as possible. This, however, is not ensured by standard interpolation-based refinement strategies, as interpolants are not guaranteed to be minimal. Furthermore, an infeasible error path might contain several reasons of infeasibility, some of which might be easier to track and be more beneficial for the further progress of the analysis than others [12].

Figure 1 shows this via an example: For the given program, an analysis based on CEGAR, with an initially empty precision, will find the shown infeasible error path. The infeasibility of this path can be explained independently by both the valuations of the variables `i` and `b`, respectively, as shown by the two example interpolant sequences. In general, and also in this example, it is advisable to track information about boolean variables, like the variable `b`, rather than loop-counter variables, such as variable `i`, because the latter may have far more valuations, and tracking loop counters would usually lead to expensive loop unrollings. Existing work combines a value analysis and a predicate analysis, with refinements always being tried first for the value analysis and for the predicate analysis only if necessary [10]. Because the value analysis cannot track the information $b \neq 0$, the given error path of the program can —by the value analysis— only be excluded by tracking the loop-counter variable i, which consequently would force unrolling the loop. If instead the predicate analysis could explicitly be chosen for refinement, then it could track the predicate $b \neq 0$ for ruling out this path without unrolling the loop. However, note that the predicate analysis could also start tracking predicates over the loop-counter variable and unroll the loop. Whether this would happen depends solely on the internal heuristics of the used interpolation engine.

Thus, for the error path in this example, we would like the verifier to refine using the predicate analysis, and we would like the interpolation engine to return the interpolant sequence shown on the left, and avoid interpolant sequences such as the one on the right, which references the loop counter. However, interpolation engines cannot distinguish between "good" or "bad" interpolant sequences, because they do not have access to external information such as if a specific variable is a loop counter and should therefore be avoided. Furthermore, the result of an arbitrary interpolation query is not directly controllable from the outside, and thus we might end up with a refinement that leads the analysis to diverge.

It is possible instead to query the interpolation engine multiple times, each time targeted at a different reason of infeasibility, or reformulate the original query in such a way that the result is expected to be "good" for the further course of the verification process. To this end, our previous work introduced the notion of *sliced prefixes* [11, 12] and an approach to extract a set of infeasible sliced prefixes for an infeasible error path. Each of these infeasible sliced prefixes can be used for refining the abstract model, and the choice influences the effectivity and the efficiency of the analysis significantly. We extend these concepts and make in this work the following key contributions:

1. We extend the generation of infeasible sliced prefixes to the domain of predicate abstraction.
2. We introduce and evaluate several heuristics for refinement selection that can significantly improve the effectiveness for both the predicate abstraction and the value domain.
3. We introduce and evaluate a novel combination of program analyses based on a value analysis and a predicate analysis, where refinement selection decides *which* of the two domains to refine.
4. We implement and make publicly available our work in the open-source software-verification framework CPAchecker.

2 Related Work

Interpolant Strength. The strength of interpolants [19] can be controlled by combining proof transformations and labeling functions, so that essentially, from the same proof of infeasibility, different interpolants can be extracted. However, to the best of our knowledge, it is not clear how to exactly exploit the strength of interpolants in order to allow performance improvements in software verification [19, 23]. In contrast to our approach, interpolant strength is not defined for value domains but is restricted to predicate abstraction. Furthermore, this approach requires changes to the implementation of the underlying interpolation engine, and no interpolation engine we know of has this feature implemented.

Exploring Interpolants. Exploring interpolants in interpolant lattices is another technique to systematically extract a set of interpolants for a given interpolation problem [23], with the goal of finding the most abstract interpolant. Similar to our proposed technique, for a single interpolation problem, the input to the interpolation engine is remodeled to obtain not only a single interpolant for a query, but a set of interpolants. This technique also does not require changes to the underlying interpolation engine, but, same as for controlling the interpolant strength, is restricted to the domain of predicate abstraction, and it can make the interpolants only more abstract. Exploring interpolants could be applied together with refinement selection to generate first the most abstract interpolant for each infeasible sliced prefix and then select the most appropriate refinement.

Unsatisfiability Cores. Current satisfiability modulo theories (SMT) solvers can extract unsatisfiability cores [16] from a proof of unsatisfiability, and one might note an analogy between a set of unsatisfiability cores extracted from a (SMT) formula and a set of infeasible sliced prefixes extracted by Alg. 1. Note, however, that the concept of infeasible sliced prefixes is more general, because it is applicable also to domains not based on (SMT) formulae, as, e.g., value domains [12]. Furthermore, while SMT solvers typically strive for small unsatisfiability cores [16], this alone does not guarantee a verifier to be effective, and to the best of our knowledge, we are not aware of any work that extracts during a single refinement several unsatisfiability cores, with the goal of performing some kind of refinement selection, as proposed in this work.

Combination of Value Analysis and Predicate Analysis. A CEGAR-based combination of a value analysis and a predicate analysis, with refinement of the abstract model in one of the two domains for every found infeasible error path, has been proposed before [10]. However, so far there was no *selection* of which domain should be used for refinement, as the strategy was always to refine, if possible, the (supposedly cheaper) value analysis first, and only use the predicate analysis if an infeasible error path is encountered that cannot be ruled out by the value analysis (and thus no refinement for the value analysis is possible at all). While this approach improves the power of the analysis, it may still easily lead to divergence, for example, if an infeasible error path is found that the value analysis can only rule out by tracking a loop-counter variable. The predicate analysis, which maybe could rule out this infeasible error path more efficiently than by unrolling the loop, would not even be considered. With our new approach, we can systematically select the abstract domain that is the most appropriate one for refinement for every single infeasible error path.

3 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 [9, 10, 12].

Programs, Control-Flow Automata, States, Paths, Precisions. We use basic definitions from previous work [12]. 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. 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 $\xrightarrow{g} \subseteq \mathcal{C} \times \{g\} \times \mathcal{C}$. The complete transition relation $\rightarrow$ is the union over all control-flow edges: $\rightarrow = \bigcup_{g \in G} \xrightarrow{g}$. We write $c \xrightarrow{g} c'$ if $(c, g, c') \in \rightarrow$, and $c \rightarrow c'$ if there exists a g with $c \xrightarrow{g} c'$.

An *abstract data state* represents a region of concrete data states. How an abstract data state is represented depends on the abstract domain being used. We write $[\![s]\!]$ for the set of concrete data states represented by an abstract data state s. The abstract data states that represent the empty set and the full set of concrete data states are denoted as $\bot$ and $\top$, respectively ($[\![\bot]\!] = \emptyset$, $[\![\top]\!] = \mathcal{C}$).

The *semantics of an operation $op \in Ops$* is defined by the strongest-post operator $\mathsf{SP}_{op}(\cdot)$: given an abstract data state s, $\mathsf{SP}_{op}(s)$ represents the set of concrete data states that are reachable from the concrete data states in the set $[\![s]\!]$ by executing op. This operator is defined by the abstract domain being used.

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., $\gamma \wedge \gamma' = \langle op_1, \ldots, op_n, op'_1, \ldots, op'_m \rangle$, 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 of the abstract domain being used 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 $(l_n, \mathsf{SP}_{\gamma_\sigma}(\top))$. 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 $(l_n, cd_n) \in Reach$, 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 $\pi : L \rightarrow 2^{\Pi}$, where Π depends on the abstract domain used by the analysis. It assigns to each program location some analysis-dependent information that defines the level of abstraction. For example, if using predicate abstraction, the set Π is a set of predicates over program variables.

Counterexample-Guided Abstraction Refinement. CEGAR [17] is a successful technique used for automatic, iterative refinement of an abstract model and aims at automatically finding a suitable level of abstraction that is precise enough to prove the specification and as abstract as possible to enable an efficient analysis. It is based on the following components: a *state-space exploration algorithm*, which computes the abstract model, a *precision*, which determines the

Algorithm 1 ExtractSlicedPrefixes(σ), taken from [12]

```
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 $\sigma$
Variables: a path $\sigma_f$ that is always feasible
 1: $\Sigma := \{\}$
 2: $\sigma_f := \langle\rangle$
 3: for each $(op, l) \in \sigma$ do
 4:     if $\mathsf{SP}_{\sigma_f \wedge (op,l)}(\top) = \bot$ then
 5:         // add $\sigma_f \wedge (op, l)$ to the set of infeasible sliced prefixes
 6:         $\Sigma := \Sigma \cup \{\sigma_f \wedge (op, l)\}$
 7:         $\sigma_f := \sigma_f \wedge ([true], l)$      // append no-op
 8:     else
 9:         $\sigma_f := \sigma_f \wedge (op, l)$          // append original pair
10: return $\Sigma$
```

current level of abstraction, a *feasibility check*, which decides if an error path is feasible, and a *refinement* procedure to refine the precision of the abstract model.

At first, the *state-space exploration algorithm* computes the abstract state space that is reachable according to the current *precision*, which initially is coarse or even empty. If all program states have been exhaustively checked, and no error was found, then the CEGAR algorithm terminates and reports the verdict TRUE, i.e., the program is safe. Otherwise, i.e., if an error path was found in the abstract state space, this error path is passed to the *feasibility check*, and if the path turns out to be feasible, meaning, there exists a corresponding concrete error path, then this error path represents an actual violation of the specification and the CEGAR algorithm terminates, reporting FALSE as verdict for the program. If, however, the error path is infeasible, i.e., it does not correspond to a concrete program path, then the precision was too coarse and needs to be refined. The *refinement* procedure takes as input the infeasible error path and returns a new precision that is strong enough that the state-space exploration algorithm will be able to exclude the current infeasible error path when it continues to build the abstract state space in the next CEGAR iteration. In practice, the refinement procedure is typically based on interpolation [18], which was first applied to the domain of predicate abstraction [21], and later to value-analysis domains [10].

Infeasible Sliced Prefixes for Refinement Selection. A path $\phi = \langle (op_1, l_1), \ldots, (op_w, l_w) \rangle$ is a *sliced prefix* [12] of a program path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ if $w \leq n$ and for all $1 \leq i \leq w$, $\phi.l_i = \sigma.l_i$ and $(\phi.op_i = \sigma.op_i$ or $(\phi.op_i = [true]$ and $\sigma.op_i$ is an assume operation)) holds, 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. Algorithm 1, extracts from a single infeasible path a set of (more abstract) infeasible sliced prefixes.

CEGAR-based analyses have to refine the abstract model based on infeasible error paths. The ability to extract multiple infeasible sliced prefixes from a single infeasible error path now enables *refinement selection* [12]: Instead of using an infeasible error path directly for refinement, and being stuck with the arbitrary and potentially "bad" interpolants that the heuristics of an interpolation engine

Algorithm 2 Refine$^{+}(\sigma)$, taken from [12]

```
Input: an infeasible error path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$
Output: a precision $\pi \in L \rightarrow 2^{\Pi}$
Variables: a set $\Sigma$ of infeasible sliced prefixes of $\sigma$,
    a mapping $\tau$ from infeasible sliced prefixes to precisions
1: $\Sigma :=$ ExtractSlicedPrefixes($\sigma$)
2: // compute precisions for each infeasible sliced prefix
3: for each $\phi_j \in \Sigma$ do
4:     $\tau(\phi_j) :=$ Refine($\phi_j$)    // c. f. standard Algorithm Refine, e. g., from [12]
5: // select a refinement based on the sliced prefixes and their respective precision
6: return SelectRefinement($\tau$)
```

produces, one can create the infeasible sliced prefixes for the infeasible error path, calculate the refinement based on each of them (using the regular refinement procedure), and then select the refinement that is the most promising, i.e., which will hopefully prevent the analysis from diverging. This is possible because each refinement for an infeasible sliced prefix is also a valid refinement for the original infeasible path [12]. Algorithm 2 is a drop-in replacement for the refinement procedure of CEGAR-based analyses that uses infeasible sliced prefixes and a heuristic for refinement selection to select an appropriate refinement. First, this algorithm calls ExtractSlicedPrefixes (Alg. 1) to extract a set of infeasible sliced prefixes. Second, it computes precisions for each of the infeasible sliced prefixes using a standard refinement procedure and stores them in the mapping τ. Third, for refinement of the abstract model, one of the precisions for the infeasible sliced prefix is selected by a heuristic (in function SelectRefinement). The heuristic can base its decision on information contained in the infeasible sliced prefixes, as well as on details about the precisions, e. g., which variables are referenced in there.

Extracting good precisions from the infeasible error paths is key to the CEGAR technique. Experiments have shown that the heuristic for refinement selection influences significantly the quality of the precision, and thus, the effectiveness of the analysis [12]. We are now interested in studying such heuristics.

4 Sliced Prefixes for the Predicate-Abstraction Domain

While refinement selection is a domain-independent technique, previous work [12] has defined it only for the domain of abstract variable assignments (value analysis). We extend it here to other domains.

In order to support refinement selection using Alg. 1 (ExtractSlicedPrefixes) and Alg. 2 (Refine^{+}), an abstract domain must define a lattice E of abstract data states, which represent sets of concrete states, and states $\top, \bot \in E$, which represent the full and the empty set of concrete states, respectively ($[\![\top]\!] = \mathcal{C}$, $[\![\bot]\!] = \emptyset$). The abstract domain must also define a strongest-post operator $\mathsf{SP}_{op}(\cdot)$ that takes an abstract data state e and returns the strongest abstract data state that represents the result of applying op on every concrete state represented by e: $\mathsf{SP}_{op}(e) = \min\{e' \in E \mid \forall c \in [\![e]\!] : \forall (l, op, l') \in G : \exists c' \in [\![e']\!] : (l, c) \xrightarrow{g} (l', c')\}$. (In case the minimum is not unique, the operator may return any of the minimal abstract states.) Note that the strongest-post operator needs to be as precise as possible within the abstract domain. The set Π of possible precision elements can be defined arbitrarily for the abstract domain. Furthermore, a procedure Refine needs to be defined that takes an infeasible path represented as a constraint sequence and returns a new precision $\pi : L \to 2^{\Pi}$ that allows the analysis to rule out the given infeasible path. These requirements are typically satisfied by any abstract domain that is used with CEGAR and interpolation. In addition to using the extended refinement procedure Refine$^+$ inside the CEGAR loop, no further changes are necessary.

For the domain of predicate abstraction [9, 20], we define the set E of abstract states as the set of all boolean combinations of all predicates over program variables. An abstract data state e represents all concrete states that satisfy e. The abstract data state $\top$ is *true*, and $\bot$ is *false*. Two abstract data states are considered equal if they are equivalent. The strongest post operator returns the strongest boolean combination of all predicates over program variables that holds after executing the given operation. The set Π is the set of all predicates over program variables. The procedure Refine is typically based on standard SMT interpolation after converting the given constraint sequence into formulae, and extracting predicates from the found interpolants.

Note that for a given infeasible path, the set of infeasible sliced prefixes extracted by the procedure ExtractSlicedPrefixes may differ depending on the abstract domain. This results from the fact that some abstract domains are stronger than others. As an example, consider the contradicting constraint sequence $\sigma = \langle (a < 0), (a = 0), (a = 1) \rangle$ over a program variable a. Using the strongest-post operator of the predicate analysis, ExtractSlicedPrefixes returns the set $\{\langle (a < 0), (a = 0) \rangle, \langle (a < 0), \mathit{true}, (a = 1) \rangle\}$ with two infeasible sliced prefixes. However, the value analysis cannot use the fact that a given variable, here a, is less than zero, and thus, can only extract from σ the single constraint sequence $\langle (a < 0), (a = 0), (a = 1) \rangle$, which is equal to the original constraint sequence σ.

Accordingly, a path that is considered infeasible by one analysis might even be considered feasible by a weaker analysis.

5 Heuristics for Refinement Selection

After computing a number of refinements for a given infeasible error path, Alg. 2 delegates to a heuristic for selecting the most promising refinement (procedure SelectRefinement). The heuristic receives as input each infeasible sliced prefix associated with the precision that was computed for the prefix. Heuristics can be as simple as choosing, for example, the shortest or longest infeasible sliced prefix, but they can also try to assess the quality of each possible refinement, for example, by looking at the variables appearing in the new precisions. In the following, we present several heuristics for refinement selection.

Selection by Domain-Type Score of Precision. One possible heuristic would be to look at the types of variables in the resulting precisions and preferring refinements with simpler or smaller types. However, in many cases the type of a variable is quite coarse and distinguishing variables on a more fine-grained level can be beneficial for verification. For example, in C the type `int` is typically used even for variables with a boolean character. To this end, domain types [2] have been proposed, which are orthogonal to the type system of a programming language and allow to classify program variables according to their actual range or usage in a program. With domain types, one can distinguish for example between effectively boolean variables, variables that are used in equality relations, in arithmetic expressions, or in bit-level operations, and variables that share characteristics of a loop counter.

Loop counters are a class of variables that ideally should be abstracted away when constructing the abstract model during the verification of a program. Yet, as loop-counter variables occur in assume operations at the loop exit, they often relate to a reason of infeasibility of a given infeasible error path, and thus can easily end up in the interpolant sequence produced by the interpolation engine and in the following would be tracked by the analysis. So, a promising heuristic is to avoid precisions with loop counters and prefer precisions with only variables of "simpler" (e.g., boolean) types. The rationale behind this heuristic is that variables with only a small number of different valuations have less influence on the growth of the state-space, and therefore are to be preferred. If, however, reasoning about the specification demands unrolling a loop, than the termination of the verification process may be delayed by first refining towards other, irrelevant properties of the program.

Formally, given a precision $\pi : L \mapsto 2^{\Pi}$ and a function $\upsilon : X \mapsto \mathbb{N}$ that maps each variable to its respective domain-type score (which is low for boolean variables, but high for loop-counter variables), we define the domain-type score of a precision as the product of the domain-type scores of every variable appearing in the precision for at least one program location: $\mathsf{DomainTypeScoreOfPrecision}(\pi, \upsilon) = \prod_{x \in \mathcal{X}} \upsilon(x)$ with $\mathcal{X} \subseteq X$ defined as the set of all variables appearing in $\bigcup_{l \in L} \pi(l)$.

Selection by Depth of Pivot Location of Precision. The structure of a refinement, i.e., which parts of the path and the state space are affected, can also be used for refinement selection. For example, refining locally near the error location may have a different effect than selecting a refinement close to the initial program location. We define as *pivot location* the first location in the current infeasible error path where the generated precision is not empty. When using lazy abstraction [22], this is typically the location from which on the reached state space is pruned and re-explored after the refinement. The depth of this pivot location can be used for comparing possible refinements and selecting one of them. Formally, for a precision π produced for a path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$, the depth of the pivot location is defined as $\mathsf{PivotDepthOfPrecision}(\pi, \sigma) = \min \{ i \in [1, \ldots, n] \mid \pi(l_i) \neq \emptyset \}$. Note that the minimum always exists because there is always at least one location with a non-empty precision.

Selecting a refinement with a deep pivot location (near the end of the path) is similar to counterexample minimization [1]. It has the advantage that (if using lazy abstraction) less parts of the state space have to be pruned and re-explored, which can save time. Furthermore, the precision will specify to track only information local to the error location and thus avoid blowing up the state space in other parts of the program. However, preferring a deep pivot location may have negative effects if some information near the entry point of the program is necessary for proving program safety (e.g., initialization of global variables). Refining at the beginning of an error path might also help to rule out a large number of similar error paths with the same prefix, which might be discovered and refined individually with the opposite strategy.

Selection by Width of Precision. Another heuristic that is based on the structure of precisions is to use the number of locations in the infeasible error path for which the generated precision is not empty, which we define as the *width* of a precision. This corresponds to how long the analysis has to track additional information during the state-space exploration for ruling out this infeasible error path, and thus may correlate to how much effort is needed. Similarly to the depth of the pivot location, this heuristic also deals with some form of "locality", but instead of using the locality in relation to the error location, it uses the locality of the information needed to exclude the infeasible error path.

Formally, for a precision π produced for a path $\sigma = \langle (op_1, l_1), \ldots, (op_n, l_n) \rangle$ the width of the precision is defined as $\mathsf{WidthOfPrecision}(\pi, \sigma) = \max \mathbb{I} - \min \mathbb{I} + 1$, where $\mathbb{I} = \{ i \in [1, \ldots, n] \mid \pi(l_i) \neq \emptyset \}$ is the set of indices along the path with a non-empty precision. (Note that if a precision is non-empty for two locations l_i and l_k along a path, it will also be non-empty for all locations between l_i and l_k, else the information tracked at l_i would not be relevant for ruling out the path.)

While it may seem at first glance that a high locality (i.e., a narrow precision) is always preferable because it means tracking additional information in a small part of the state space, note that the locality of loop counters is often high, because in many loops the statements for assigning to the loop counter are close to the loop-exit edges. Thus selecting a narrow precision might often lead to the tracking of loop counters and unrolling of loops.

Selection by Length of Infeasible Sliced Prefix. Two possible heuristics for refinement selection are to simply select the shortest and longest infeasible sliced prefixes, respectively. However, while both these heuristics are legitimate and may work on some benchmarks, we do not regard them as systematical, because they do not attempt to specifically select a refinement that will be beneficial for the progress of the analysis, and any success in using them will solely depend on the internal structure of the analyzed programs.

Further Heuristics. In this section, we presented and motivated several promising heuristics, and of course one can always define and experiment with further heuristics. For example, in the RERS challenge 2014, a heuristic specifically tailored to the characteristics of the event-condition-action systems used in that competition, improved the effectiveness of our tool CPAchecker and allowed it to obtain two gold and one bronze medals, as well as two special achievements [1]. This shows that using domain knowledge in the refinement step of the CEGAR loop is a promising direction, and the definition of an according heuristic for refinement selection is a suitable place to do so.

[1] Results available at `http://www.rers-challenge.org/2014Isola/`

6 Refinement Selection for Combination of Analyses

A combination of two different analyses, such as a value analysis and a predicate analysis, can be beneficial because different facts necessary to prove program safety can be tracked by the analysis that can track a fact most easily [7, 10]. The refinement step is a natural place for choosing which of the analyses should track new information. Thus we extend the idea of refinement selection from an intra-analysis selection to an inter-analysis selection: After an infeasible error path has been found, we can extract the infeasible sliced prefixes according to the semantics of each of the analyses, use the standard refinement procedure of each analysis to compute a new precision for each of the infeasible sliced prefixes, and then give all of them to a combined heuristic for refinement selection. This heuristic can then decide not only which infeasible sliced prefix should be used for refining the abstract model, but also which of the *analyses* should be refined.

This can improve the effectiveness of the analysis, for example, if there is an infeasible error path that forces the analysis to track the information that a certain variable is within some interval. Refining using the value analysis would mean to enumerate all possible values of this variable, whereas the predicate analysis could track this more efficiently using inequality predicates. Previously, combinations of such analyses statically preferred the (supposedly cheaper) refinement of the value analysis [10], which can be suboptimal, as shown.

7 Evaluation

In the following, we present the results of applying refinement selection to several analyses. In order to evaluate the presented heuristics for refinement selection, we have integrated them into the open-source software-verification framework CPAchecker [8] [2]. We also implemented refinement selection for the predicate-abstraction domain [9] in CPAchecker, such that it is now supported for both the value analysis [10] and predicate abstraction.

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 minutes of CPU time, and 15 GB of memory. We measured CPU time and report it rounded to two significant digits. BenchExec [3] was used as benchmarking framework to ensure precise and reproducible results. We used the branch `refinementSelection` in revision 16 781 of CPAchecker for our experiments, and we make the tool, the benchmarks, and the full results, available on our supplementary web page [4].

Benchmarks. For evaluation of our novel approach, we use C programs from SV-COMP'15 [5]. From the 5 803 tasks used in SV-COMP'15, we select those tasks that deal with reachability properties, and exclude tasks from the categories "Arrays", "HeapManipulation", "Concurrency", and "Recursion", because they are not supported by both analyses we evaluate. Furthermore, we present here only

[2] Available under the Apache 2.0 License from `http://cpachecker.sosy-lab.org/`

[3] `https://github.com/dbeyer/benchexec`

[4] `http://www.sosy-lab.org/~dbeyer/cpa-ref-sel/`

results for those tasks where a refinement *selection* is actually possible, i. e., where at least one refinement with more than one infeasible sliced prefix is performed. Thus, the set of all verification tasks in our experiments contains 2 827 and 2 638 verification tasks for the predicate and value analysis, respectively.

Configuration. In order to properly evaluate the effect of the precisions that are chosen by the refinement-selection heuristic, we restart the state-space exploration with the new precision from the initial program location after each refinement. Otherwise, i.e., if we were using lazy abstraction and re-exploring only the necessary part of the state space, not only the new precision but also the amount of re-explored state space would differ depending on the selected refinement, which would have an undesired impact on the performance. For the same reason, we also configure the analysis to interpret the precision globally, i. e., instead of a mapping from program locations to sets of precision elements, the discovered precision elements get used at all program locations (note that this does not change the precision when viewed by the refinement-selection heuristic, but only the precision usage during the state-space exploration).

The predicate analysis is configured to use single-block encoding [9], because for larger blocks there is no single error path per refinement, but instead a sequence of blocks encoding a set of potential error paths, and thus extracting infeasible sliced prefixes from a single path and using them for refinement selection is not applicable anymore. As SMT solver and interpolation engine, the predicate analysis uses SMTInterpol [15].

Refinement-Selection Heuristics. We experiment with implementations of the procedure SelectRefinement in Alg. 2 based on the heuristics from Sec. 5, specifically such that it returns the precision for a (1) short or (2) long infeasible sliced prefix, the precision with a (3) good or (4) bad domain-type score [5], a precision that is (5) narrow or (6) wide, or a precision with a (7) shallow or (8) deep pivot state. For comparison, we report the results of using random choice as heuristic for refinement selection. We also experiment with combinations of heuristics, where at first a primary heuristic is asked, and if this does not lead to a unique selection, a secondary heuristic is used as a tie breaker to select one of those refinements that were ranked best by the primary heuristic. We use the heuristics "good domain-type score" and "narrow precision" for these combinations. In all configurations of refinement selection, if necessary, we use the length of the infeasible sliced prefix as a final tie breaker, and select from equally ranked refinements the one with the shortest infeasible sliced prefix [6].

In the following, we compare the potential of these selection heuristics against each other, as well as against the case where the finding of a refinement is solely left to the interpolation engine, i. e., where no refinement selection is performed.

Refinement Selection for Predicate Analysis. We evaluate the presented heuristics for refinement selection when applied to predicate abstraction. Table 1

[5] We do not expect the precision with the worst domain-type score to be actually useful, we report its results merely for comparison.

[6] Experiments showed no relevant difference between selecting the shortest or the longest infeasible sliced prefix in case of a tie in the primary selection heuristic.

Table 1: Number of solved verification tasks for predicate analysis without and with refinement selection using different heuristics

Heuristic \ Tasks		All Tasks	ControlFlowInt.	DD64	ECA	ProductLines	Seq.
		2 827	35	678	1 140	597	244
— (No Refinement Selection)		1 139	**34**	472	161	325	42
Length of Prefix	Short	1 259	**34**	427	252	**370**	78
	Long	1 301	18	481	311	330	66
Domain-Type Score	Good	1 284	**34**	491	243	342	75
	Bad	1 134	23	399	254	298	64
Width of Precision	Narrow	1 289	29	430	**314**	346	70
	Wide	1 253	27	477	281	312	61
Depth of Precision	Shallow	1 225	25	464	246	340	57
	Deep	1 258	28	418	296	351	64
Random		1 328	**34**	467	295	345	**82**
Combinations	Good&Narrow	**1 363**	30	**494**	**314**	345	81
	Narrow&Good	1 339	28	474	**314**	354	70

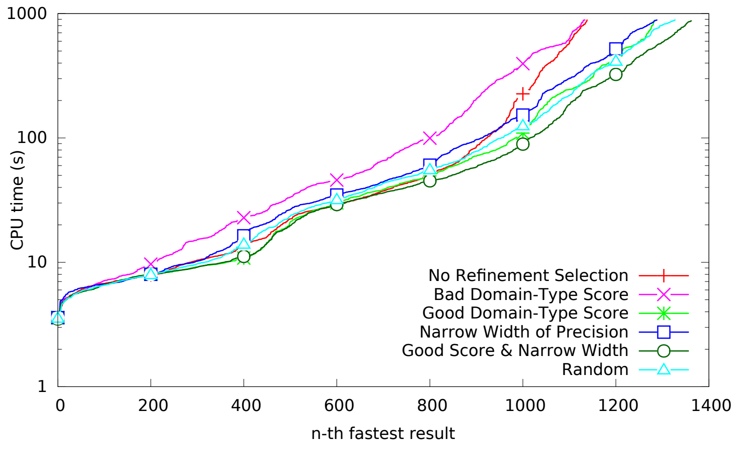

Fig. 2: Quantile plot showing the results for predicate analysis without and with refinement selection using different heuristics

shows the number of verification tasks that the predicate analysis could solve without refinement selection, and with refinement selection using the heuristics and combinations of heuristics listed above. The table lists the results for the full set of 2 827 verification tasks (column "All Tasks") that fit the criterion defined in the previous section, as well as for several subsets corresponding to those categories of SV-COMP'15 ("ControlFlowInteger", "DeviceDrivers64", "ECA", "ProductLines", and "Sequentialized"), where refinement selection has a significant impact. Numbers written in bold digits highlight the best configuration(s) in each column. Figure 2 shows a quantile plot with the results on the full set of tasks for the most interesting refinement-selection heuristics, from which some performance characteristics can be seen.

Refinement Selection Matters. For the full set of tasks, the analysis without refinement selection is worse than all heuristics for refinement selection, except for the intentionally bad heuristic "bad domain-type score". Additionally, the analysis without refinement selection is not the best for any of the shown subsets of tasks, except for "ControlFlowInteger", where it is tied for first with others. This shows that the heuristics of the interpolation engine (with which we are stuck without refinement selection) are not well-suited for verification, and that practically any deviation away from the heuristics of interpolation engine pays off, as witnessed by the relatively good results for the random refinement selection.

Discussion. As Table 1 shows, none of the basic heuristics works best for all classes of programs, but instead in each subset a different heuristic is the best. In the following, we would like to highlight and explain a few interesting results for some subsets of tasks and heuristics. Note that the following discussion is based on the investigation of some program samples and our understanding of the characteristics of the programs in the SV-COMP categories, and we do not claim that our explanations are necessarily true for all programs.

The programs of the subset "DeviceDrivers64" contain many functions and loops, and aspects about the specification are encoded in global boolean variables that are checked right before the error location. Hence, the heuristic "good domain-type score" is effective as it successfully selects precisions with the "easy" and relevant boolean variables. The heuristics "long prefix", "wide precision" and "shallow depth" all happen to work well, too, because those relevant variables are initialized at the beginning and read directly before the error location, meaning that corresponding infeasible sliced prefixes will be long, and resulting precisions containing them will be "shallow" and "wide" (starting to track information close to the program entry, and all the way to the error location). Their opposing heuristics tend to prefer precisions about less relevant local variables.

The subset "ECA" contains artificial programs that represent event-condition-action systems with up to 200 000 lines of code. Most of these programs have only a few variables, and for the larger part of these programs, all the variables have the same domain type, and thus the heuristic using the domain-type score cannot perform a meaningful refinement selection here and degenerates to a heuristic about the number of distinct variables in the precision. Note also, that relying on the internal heuristics for interpolant generation of the SMT solver works particularly bad for these programs.

The programs of the subset "ProductLines" encode state machines and contain a high amount of global variables. In case they contain a specification violation, the bug is often rather shallow, although the full state space is quite complex. This explains why heuristic "short prefix" works especially well here, as this heuristic leads to exploring the state space as close as possible to the initial program location, driving the verification towards shallow bugs.

Combination of Refinement-Selection Heuristics. The above results show that it is worthwhile to experiment with combinations of heuristics in order to find a configuration that works well for a wide range of programs. We used the two heuristics "good domain-type score" and "narrow precision", which are not only

Table 2: Number of solved verification tasks for value analysis without and with refinement selection using different heuristics

Heuristic \ Tasks		All Tasks	DeviceDrivers64	ECA	ProductLines
		2 638	578	1 140	597
— (No Refinement Selection)		1 740	407	**585**	458
Length of Prefix	Short	1 656	424	491	456
	Long	1 630	484	511	361
Domain-Type Score	Good	1 761	**494**	573	408
	Bad	1 524	412	480	359
Width of Precision	Narrow	1 689	422	510	471
	Wide	1 608	482	494	356
Depth of Precision	Shallow	1 663	473	521	383
	Deep	1 729	415	536	**489**
Random		1 626	438	524	379
Combinations	Good&Narrow	**1 774**	**494**	573	421
	Narrow&Good	1 718	**494**	510	427

two of the most successful basic heuristics for the predicate analysis, but are also somewhat complimentary (one has a weak spot where the other is strong, and vice versa). Indeed, regardless of in which order the two heuristics are combined, the combination is the most successful configuration for all tasks. The combination with "good domain-type score" as primary and "narrow precision" as secondary heuristic manages to solve 224 (20 %) more tasks than without refinement selection and is best or close to best in most subsets of tasks.

Refinement Selection for Value Analysis. We now compare the different refinement-selection heuristics when used together with a value analysis. The results are shown in Table 2, which is structured similarly to Table 1, but contains only results for the full set of 2 638 tasks and for the subsets corresponding to the SV-COMP'15 categories "DeviceDrivers64", "ECA", and "ProductLines", because for the remaining categories there is no relevant difference in the results for the value analysis. First it can be seen that the configuration without refinement selection is comparatively good for value analysis, as opposed to predicate analysis, where it is the second worst configuration. This can be explained with the fact that the interpolation engine for the value analysis is implemented in CPAchecker itself and is thus designed and tuned specifically for software verification, whereas the predicate analysis uses an off-the-shelf SMT solver as interpolation engine, which is not designed specifically for software verification. However, for specific subsets of tasks, refinement selection is also effective for the value analysis.

Similarly to the predicate analysis, none of the heuristics is the best for all classes of programs. Again, the basic heuristic that works best on the set of all tasks is "good domain-type score", which is especially well-suited for the subset "DeviceDrivers64" for the same reasons explained above. In fact, note that for the basic heuristics and subsets of tasks presented in Tables 1 and 2, the number of tasks solved by the value analysis often correlates closely to the number of tasks solved by the predicate analysis. On notable exception is for the subset "ECA",

Table 3: Number of solved verification tasks for combinations of analyses without and with refinement selection (PA: predicate analysis; VA: value analysis)

Tasks / Analysis	All Tasks	DD64	ECA	Loops	ProductLines	Seq.
	3 563	1 240	1 140	120	597	261
PA	1 824	1 026	160	**80**	325	42
VA + PA	2 300	993	503	69	**422**	118
VA_{RefSel} + PA_{RefSel}	**2 384**	**1 072**	**521**	68	404	**124**
$(\text{VA}_{RefSel} + \text{PA}_{RefSel})_{RefSel}$	**2 384**	1 065	**521**	79	403	120

where the heuristic "good domain-type score" works well for the value analysis, but not for the predicate analysis. The reason why this is different is that the value analysis anyway solves far more instances than the predicate analysis, and for some of the harder problems —the ones the predicate analysis cannot solve, but the value analysis can— there exist variables with different domain-types, hence, allowing the heuristic "good domain-type score" to be more effective.

Finally, the combination of the refinement-selection heuristics "good domain-type score" and "narrow precision" is again the most effective configuration for the set of all tasks, although the increase over the heuristic "good domain-type score" alone is here not as large as it is for the predicate analysis shown before.

Refinement Selection for Combination of Analyses. We now evaluate the effectiveness of using refinement selection for a combination of analyses. In order to do so, we compare four different analyses: (1) an analysis based on predicate abstraction alone without refinement selection, (2) a combination of a value analysis and a predicate analysis (both without refinement selection), where refinements are always tried first with the value analysis and the predicate analysis is only used if the value analysis cannot rule out an error path, (3) the same combination of a value analysis and a predicate analysis, but now with refinement selection used independently in both domains, and (4) our novel combination presented in Sec. 6 of a value analysis and a predicate analysis, where refinement selection is not only used within each domain but also to decide which domain to prefer in a refinement step. For all configurations with refinement selection here we use the combination of the heuristics "good domain-type score" and "narrow precision". We keep the same setup for the experiment as before, but use a new selection criteria, namely, we only consider verification tasks where an inter-analysis refinement selection is actually possible, i. e., where the analysis based on our novel combination needs to perform at least one refinement.

Results. Table 3 shows the results for this comparison. Confirming previous results [10], even a combination of value analysis and predicate analysis without refinement selection (row "VA + PA") is more effective than the predicate analysis alone (row "PA"). However, this combination also has weak spot, as it fails in "DeviceDrivers64" due to state-space explosion where the predicate analysis alone succeeds. Row "VA_{RefSel} + PA_{RefSel}" shows that using refinement selection is effective not only when applied to individual analyses, but also for combinations of analyses. Finally, the fourth configuration (row "$(\text{VA}_{RefSel} + \text{PA}_{RefSel})_{RefSel}$") takes the idea of refinement selection to the next level. While in the other combinations the value analysis is always refined first, and the predicate analysis only if the value analysis cannot rule out an infeasible error path, our novel combination uses refinement selection to decide whether a refinement for the value or for the predicate analysis is thought to be more effective. On the full set of tasks, this approach is only tied with the previous approach, but the encouraging results in the subset "Loops" show that it works as intended. In this subset the plain predicate analysis is best (row "PA"), and a *naive* combination is less suited for such programs (rows "VA + PA" and "VA_{RefSel} + PA_{RefSel}"). If, however, we apply *inter-analysis* refinement selection to decide which analysis to refine for a given error path, as done by our novel approach, then this does not only clearly out-perform the plain predicate analysis on "All Tasks", but it also matches the effectiveness of the predicate analysis for programs where reasoning about loops is essential.

8 Conclusion

We have shown that refinement selection has a significant impact on the effectiveness and efficiency of CEGAR-based analyses. We have presented a variety of heuristics for utilizing this potential and evaluated them on a wide set of benchmarks, showing that we can improve the performance and the number of solved tasks significantly by selecting an appropriate refinement without any further changes to the analysis. This result is valid for the two abstract domains that are most commonly used with CEGAR (predicate abstraction and value analysis). Furthermore, if using a combination of both a value and a predicate analysis, refinement selection can now be used to systematically select the most appropriate domain for refining the abstract model using an infeasible error path, as opposed to previous work, which always refines a predefined domain first.

As future work, besides more heuristics and combinations of them, an interesting direction would be to have heuristics that use *dynamic* information from the analysis, for example penalizing a variable not if it can be statically detected as a loop counter, but if the number of values that are found for this variable during the analysis exceeds a certain bound, as it is done in dynamic precision adjustment [7]. We intend to extend the support for refinement selection in the predicate-abstraction domain to adjustable-block encoding [9]. Especially for predicate abstraction, one could also devise a heuristic that does not only look at the domain type of variables, but also at how these variables appear in the precision (for example, an equality predicate for a loop counter usually leads to loop unrolling, while an inequality predicate can be more efficient).

References

1. 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.
2. S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. v. Rhein. Domain types: Abstract-domain selection based on variable usage. In *Proc. HVC*, LNCS 8244, pages 262–278. Springer, 2013.

3. T. Ball, B. Cook, V. Levin, and S. Rajamani. SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Proc. IFM*, LNCS 2999, pages 1–20. Springer, 2004.
4. T. Ball and S. K. Rajamani. The SLAM project: Debugging system software via static analysis. In *Proc. POPL*, pages 1–3. ACM, 2002.
5. D. Beyer. Software verification and verifiable witnesses (Report on SV-COMP 2015). In *Proc. TACAS*, LNCS 9035. Springer, 2015.
6. 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.
7. D. Beyer, T. A. Henzinger, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.
8. D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
9. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
10. 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.
11. 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.
12. D. Beyer, S. Löwe, and P. Wendler. Sliced path prefixes: An effective method to enable refinement selection. In *Proc. FORTE*, LNCS 9039, pages 228–243. Springer, 2015. To be published, preprint available at `http://www.sosy-lab.org/~dbeyer/Publications/2015-FORTE.Sliced_Path_Prefixes_An_Effective_Method_to_Enable_Refinement_Selection.pdf`.
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. J. Christ, J. Hoenicke, and A. Nutz. SMTINTERPOL: An interpolating SMT solver. In *Proc. SPIN*, LNCS 7385, pages 248–254. Springer, 2012.
16. A. Cimatti, A. Griggio, and R. Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In J. Marques-Silva and K. A. Sakallah, editors, *Proc. SAT*, volume 4501, pages 334–339. Springer, 2007.
17. 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.
18. W. Craig. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.
19. V. D'Silva, D. Kröning, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proc. VMCAI*, LNCS 5944, pages 129–145. Springer, 2010.
20. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.
21. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
23. P. Rümmer and P. Subotic. Exploring interpolants. In *Proc. FMCAD*, pages 69–76. IEEE, 2013.