Refinement Selection

Dirk Beyer, Stefan Löwe, and Philipp Wendler

University of Passau, Germany

Abstract. Counterexample-guided abstraction refinement (CEGAR) is a property-directed approach for the automatic construction of an abstract model for a given system. The approach learns information from infeasible error paths in order to refine the abstract model. We address the problem of selecting *which* information to learn from a given infeasible error path. In previous work, we presented a method that *enables* refinement selection by extracting a set of sliced prefixes from a given infeasible error path, each of which represents a different reason for infeasibility of the error path and thus, a possible way to refine the abstract model. In this work, we (1) define and investigate several promising heuristics for selecting an appropriate precision for refinement, and (2) propose a 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* should be preferred for a refinement. These contributions allow a more systematic refinement strategy for CEGAR-based analyses. We evaluated the idea on software verification. We provide an implementation of the new concepts in the verification framework CPAchecker and make it publicly available. In a thorough experimental study, we show that refinement selection often avoids state-space explosion where existing approaches diverge, and that it can be even more powerful if applied on a higher level, where it decides which analysis of a combination should be favored for a refinement.

1 Introduction

Abstraction is a key concept to enable the verification of real-world software (cf. [3, 4, 14, 25]) within reasonable time and resource limits. Slam [5], for example, uses predicate abstraction [21] for creating an abstract model of the software. The abstract model is often constructed using counterexample-guided abstraction refinement (CEGAR) [17], which iteratively refines an (initially coarse) abstract model using *infeasible* error paths (property-directed refinement). This technique is integrated in many successful software-verification tools, e.g., Slam [5], Blast [7], and CPAchecker [9]. In the refinement step of the CEGAR framework, Craig interpolation [18, 26] is often used to extract the information that needs to be tracked by the analysis [11, 22]. Formula 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 interpolants over an infeasible error path to refine the abstract model, and iteratively augment the abstraction until it is strong enough so that the specification can be proven.

```
1  extern int nondet_int();
2  extern int f(int x);
3  int main() {
4    int b = nondet_int();
5    int i = 0;
6    if (b) {
7      while (i < 1000) {
8        f(i++);
9      }
10   }
11   if (i != 0) {
12     if (!b) {
13       assert(0);
14     }
15   }
16 }
```

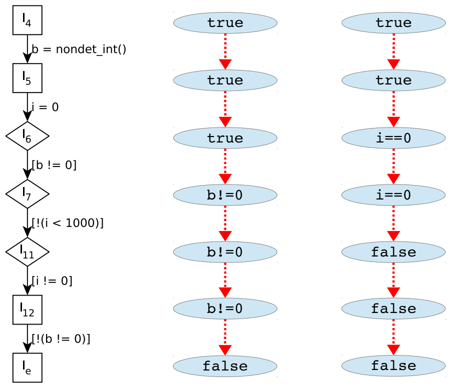

Fig. 1: From left to right, the input program, an infeasible error path, and a "good" and a "bad" interpolant sequence for the infeasible error path

In order to avoid state-space explosion and divergence during program analysis, we need to keep the precision of the analysis as abstract and concise as possible. Existing approaches that use interpolation to extract precision information from infeasible error paths assign a lot of choice to the interpolation engine, i.e., an infeasible error path might contain several reasons for its infeasibility, some of which might be easier to track than others and thus might be more beneficial for the further progress of the analysis [13]. Our work addresses the choice between different precisions — a concept that we refer to as *refinement selection*.

Figure 1 shows this via an example: For the given program, an analysis based on CEGAR, with an initially empty precision, may find the shown infeasible error path. The infeasibility of this path can be explained independently by the valuations of the variables `i` and `b`, as shown by the two example interpolant sequences. In general, and also in this example, it is advisable to track information about boolean variables [1], like the variable `b`, rather than loop-counter variables, such as variable `i`, because the latter may have far more different valuations, and tracking loop counters would usually lead to expensive loop unrollings. The given error path of the program can be eliminated from further exploration by tracking the loop-counter variable i, which might force unrolling the loop in further iterations of CEGAR. If we instead choose to track the boolean variable b, then the path can be eliminated and it is guaranteed that the loop is not unrolled. In the next CEGAR iteration, variable i will be added to the precision in order to stop the exploration at line 11. After that, the program is proved correct. In existing work, the decision which variable to track depends solely on the interpolation engine.

For the error path in this example, we would like the verifier to refine using the interpolant sequence shown on the left, and avoid interpolant sequences such

[1] In the programming language C, a boolean variable is modeled by an integer variable b for which `b==0` represents the value *false* and `b!=0` represents the value *true* (cf. [2] for a discussion of more fine-grained types for C).

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 to divergence of the analysis.

It is possible instead to send several queries to the interpolation engine, each targeted at a different reason of infeasibility, and then choose the result that is expected to be "good" for the further construction of the abstract model. Our previous work introduced the notion of *sliced prefixes* [13] together with an approach to extract a set of such infeasible sliced paths for one given infeasible error path. Each of these infeasible sliced paths can be used for refining the abstract model, and the choice influences the effectivity and the efficiency of the analysis significantly. This work investigates *refinement selection*, yielding the following contributions:

- We present several heuristics for intra-analysis refinement selection, for which we conduct a thorough evaluation that reveals significant effectiveness improvements for both a predicate analysis and a value analysis.
- We define a novel combination of analyses, where inter-analysis refinement selection decides *which* analysis in the combination of analyses is refined.
- We provide an implementation that is publicly available in the open-source software-verification framework CPAchecker.

Related Work. We categorize the related approaches into approaches that manipulate error paths, interpolation approaches to be implemented inside the interpolation engine, or outside the interpolation engine, approaches based on unsat cores, and combination approaches.

Extraction of Paths. The most related approaches to refinement selection are works that manipulate infeasible error paths. Path slicing [24] is a technique that weakens the path constraints before interpolation by removing facts that are not important for the infeasibility of the error path. This technique produces one infeasible sliced path for one infeasible error path. We need several infeasible sliced paths in order to create a space of choice for refinement selection. Sliced path prefixes [13] is a method that produces a set of infeasible sliced paths, i.e., a set of infeasible sliced prefixes of the original infeasible error path. One of our heuristics (deep pivot location) is similar to counterexample minimization [1].

Interpolant Strength. The strength of interpolants [20] 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, it is not yet clear from the literature how to exactly exploit the strength of interpolants in order to improve the performance of software verification [20, 27]. In contrast to our approach, interpolant strength is restricted to predicate analysis, requires changes to the implementation of the underlying interpolation engine, and no available interpolation engine provides this feature.

Exploring Interpolants. Exploring interpolants [27] in interpolant lattices is a technique to systematically extract a set of interpolants for a given interpolation problem, 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 is restricted to predicate analysis. Yet, this technique could be applied together with refinement selection to generate first the most abstract interpolant for each infeasible sliced path and then select the most appropriate refinement.

Unsatisfiability Cores. Satisfiability modulo theory (SMT) solvers can extract unsatisfiability cores [16] from a proof of unsatisfiability, and there is an analogy between a set of unsatisfiability cores extracted from a formula and a set of infeasible sliced paths [13]. The concept of infeasible sliced paths is more general, because it is applicable also to domains not based on SMT formulas, such as value domains [13]. While SMT solvers typically strive for small unsatisfiability cores [16], this alone does not guarantee a verifier to be effective. It would be interesting to extract several unsatisfiability cores during a single refinement, with the goal of performing 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 [11]. However, so far there was no path-based *selection* which domain should be refined: the strategy was to refine first, if possible, the (supposedly cheaper) value analysis, and only refine the predicate analysis if the value analysis could not eliminate the infeasible error path. This analysis may diverge, if the value analysis needs to track a loop-counter variable, for example. The predicate analysis, which might have eliminated the infeasible error path without unrolling the loop, would have not even been considered. With our new approach, we can systematically select the abstract domain that is the most appropriate for refinement, for every single infeasible error path.

2 Preliminaries

Programs, Control-Flow Automata, States. We use basic definitions from previous work [13]. 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 a value from the set $\mathbb{Z}$ of integer values. A region ϕ is a formula that represents a set of concrete data states. For a region ϕ and a CFA edge (l, op, l'), we write $\mathsf{SP}_{op}(\phi)$ to denote the strongest postcondition. Each program analysis comes with an own implementation of SP, each with possibly different expressive power. For example, a program analysis that is restricted to the theory of linear arithmetics will provide a strongest postcondition that uses formulas in the theory of linear arithmetic.

Paths, Sliced Paths, Precisions. 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 *semantics of a path* $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$ and an initial region ϕ is defined as the successive application of the strongest postcondition to each operation of the path: $\mathsf{SP}_\sigma(\phi) = \mathsf{SP}_{op_n}(\ldots \mathsf{SP}_{op_1}(\phi) \ldots)$. A path σ is *feasible* if $\mathsf{SP}_\sigma(\mathit{true})$ is not contradicting. A program path $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_e)\rangle$, which ends in l_e, is called *error path*, and a program is considered *safe* (the specification is satisfied) if there is no feasible error path.

A *sliced path* is a path that results from a path by omitting pairs of operations and locations from the beginning or from the end, and possibly replacing some assume operations by no-op operations. Formally, a path $\phi = \langle(op'_j, l'_j), \ldots, (op'_w, l'_w)\rangle$ is called a *sliced path* of a path $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$ if $j \geq 1$, $w \leq n$, and for all $j \leq i \leq w$, $\phi.l'_i = \sigma.l_i$ and $(\phi.op'_i = \sigma.op_i$ or $(\phi.op'_i = [\mathit{true}]$ and $\sigma.op_i$ is an assume operation$))$ holds.

The definition of sliced paths is inspired by path slicing [24] and sliced prefixes [13]. To ensure that any standard interpolation-based refinement procedure can be used, the following proposition is necessary: Let σ be an infeasible path and ϕ be an infeasible sliced path of σ, then all interpolant sequences for ϕ are also interpolant sequences for σ. The proof for this proposition follows directly from the respective proof for infeasible sliced prefixes [13]. This property allows to replace a refinement procedure that uses only the original infeasible path, by a procedure that uses a set of infeasible sliced paths.

Previously, we introduced one possible approach to extract a set of infeasible sliced paths from one infeasible path: generating infeasible sliced prefixes [13]. It was only defined for a value analysis, however, it can be extended to any analysis that provides a representation of sets of concrete data states and an operator SP for computing strongest postconditions. The predicate analysis fulfills these requirements, allowing us to implement Alg. ExtractSlicedPrefixes [13] and Alg. 1 (Refine^+) for the predicate analysis. Other approaches for generating infeasible sliced paths from an infeasible path are equally applicable for refinement selection.

A *precision* is a function $\pi : L \to 2^{\Pi}$, 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.

Algorithm 1 $\mathsf{Refine}^{+}(\sigma)$, adopted from [13]

```
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 set $\Sigma$ of infeasible sliced paths of $\sigma$,
    a set $\tau$ of pairs of an infeasible sliced path and a precision
1: $\Sigma := \mathsf{ExtractSlicedPaths}(\sigma)$
2: // compute precisions for each infeasible sliced path
3: for each $\phi_j \in \Sigma$ do
4:     $\tau := \tau \cup (\phi_j, \mathsf{Refine}(\phi_j))$    // cf. standard Alg. Refine, e. g., from [13]
5: // select a refinement based on original path, infeasible sliced paths, and resp. precisions
6: return $\mathsf{SelectRefinement}(\sigma, \tau)$
```

Counterexample-Guided Abstraction Refinement. CEGAR [17] is a 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 while being 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 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.

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 correct. 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 is feasible, then this error path represents an actual violation of the specification and the CEGAR algorithm terminates with verdict FALSE. If, however, the error path is infeasible, i.e., does not correspond to a concrete program execution, 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 not explore that infeasible error path again in the next CEGAR iterations. The refinement procedure is often based on interpolation [18], which was first applied to the predicate domain [22], and later to the value-analysis domain [11].

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 [13]. Here, we are interested in studying such heuristics.

3 Refinement Selection using Heuristics

CEGAR needs a module Refine that takes as input an infeasible program path and yields as output a precision that is used for refinement of the abstract model. Instead of using an infeasible program path directly for a standard interpolation-based refinement, and being stuck with the arbitrary and potentially "bad" interpolants that the internal heuristics of an interpolation engine produce, we use a new module Refine$^+$. Algorithm 1 can be substituted for the refinement procedure of CEGAR-based analyses. This new module first extracts a set of infeasible sliced paths by calling method ExtractSlicedPaths, which are more abstract than the original program path. (ExtractSlicedPrefixes [13] is one possible implementation of method ExtractSlicedPaths.) Second, Alg. Refine$^+$ calculates the precision for each infeasible sliced path (using a regular refinement procedure) and stores the pairs in set τ. Third, the algorithm selects the precision that is the most promising from τ, i.e., which will hopefully prevent the analysis from diverging. The selection is implemented in a method SelectRefinement and uses details from the precisions, e.g., which variables are referenced in the precision. Each implementation of SelectRefinement, i.e., each heuristic, receives as input the original infeasible path as well as the set of all pairs of infeasible sliced paths and respective precisions. The remainder of this section presents some possible heuristics that can be used to implement SelectRefinement.

Selection by Domain-Type Score of Precision. Our first heuristic inspects the types of variables in the resulting precisions and prefers refinements with simpler or smaller types. In C, the type of a variable is quite coarse and distinguishing variables on a more fine-grained level can be beneficial for verification. For example, the C type `int` is typically used even for variables with a boolean character. For this purpose, domain types [2] have been proposed, which refine 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 between variables that are used as booleans, variables that are used in equality relations only, in arithmetic expressions, or in bit-level operations, and variables that share characteristics of a loop counter [19, 28, 29].

Loop counters are a class of variables that a program analysis should ideally omit in many cases from the abstract model of a program. Because 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. Thus, those variables are often included in the interpolant sequence that a standard interpolation engine might produce, forcing the program analysis to track them. Therefore, a promising heuristic is to avoid precisions that contain 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 values to grow the state-space, and therefore are to be preferred. If, however, reasoning about the specification demands unrolling a loop, then the termination of the verification process may be delayed by first refining towards other, irrelevant properties of the program.

In order to compute the domain-type score for a precision $\pi : L \mapsto 2^{\Pi}$, we first define a function $\delta : X \mapsto \mathbb{N} \setminus \{0\}$ that assigns to each program variable its domain-type score. The domain type for all program variables can be inferred by an efficient data-flow analysis [2], and we use low score values for variables with small ranges (e.g., boolean variables), and a specifically high value for loop counters. Thus, we define the domain-type score of a precision as the product of the domain-type scores of every variable that is referenced in the precision: $\mathsf{DomainTypeScoreOfPrecision}(\pi, \delta) = \prod_{x \text{ referenced in } \pi} \delta(x)$.

This function, as well as the design of function δ, are mere proposals for assessing the quality of a precision. However, we experimented with several different implementations for both functions, and come to the conclusion that the most important requirement to be fulfilled is that precisions with only boolean variables should be associated with a low score, and precisions referencing loop-counter variables should be penalized with a high score.

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 close to the error location may have a different effect than refining close to the program entry. We define the *pivot location* of an infeasible error path σ as the first location in σ where the precision is not empty. If using lazy abstraction [23], 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 for a path $\sigma = \langle(op_1, l_1), \ldots, (op_n, l_n)\rangle$, the depth of the pivot location is defined as $\mathsf{PivotDepthOfPrecision}(\pi, \sigma) = \min \{i \mid \pi(l_i) \neq \emptyset\}$. (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 (close to 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 be more efficient. Furthermore, the precision will specify to track preferably information local to the error location and thus avoid unfolding the state space in other parts of the program. However, preferring a deep pivot location may have negative effects if some information close to the program entry 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 precision, which might otherwise be discovered and refined individually.

Selection by Width of Precision. Another heuristic that is based on the structure of a refinement is to use the number of locations in the infeasible error path for which the precision is not empty, which we define as the *width* of a precision. This corresponds to how long on a path the analysis has to track additional information during the state-space exploration, and thus correlates to how long the precision contributes to the state-space unfolding. 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 depth, it uses the locality in relation to the width. 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 $\mathsf{WidthOfPrecision}(\pi, \sigma) = 1 + \max I - \min I$, where $I = \{i \mid \pi(l_i) \neq \emptyset\}$ is the set of indices along the path with a non-empty precision.

It may seem that narrow precisions are in general preferable, because it means tracking additional information only in a small part of the state space. However, narrow precisions favor loop counters because in many loops the statements for assigning to the loop counter are close to the loop-exit edges. Thus, selecting a narrow precision often leads to loop unrollings.

Selection by Length of Infeasible Sliced Path. Selecting the shortest or longest infeasible sliced path, respectively, are two simple heuristics for refinement selection as well.

Further Heuristics. We presented and motivated several promising heuristics, but other heuristics are possible as well. For example, in the RERS challenge 2014, a heuristic specifically tailored to the characteristics of the event-condition-action systems in that competition, improved the effectiveness of CPAchecker and allowed it to obtain good results [2]. This shows that using *domain knowledge* in the refinement step of CEGAR is a promising direction, and a specific heuristic for refinement selection is a suitable place to define this.

4 Refinement Selection for Combination of Analyses

A combination of different analyses, such as a value analysis and a predicate analysis, can be beneficial because different facts necessary to prove program correctness can be handled by the analysis that can track a fact most efficiently [8, 11]. 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.

This concept, which can be broken down into four phases, is depicted in Figure 2, which shows an example combination of a value analysis (VA) and a predicate analysis (PA). The first phase is the standard exploration phase of CEGAR. The composite analysis performs the state-space exploration, constructing the abstract model using the initial, empty precision for all component analyses. In the figure, we refer to the precisions as π^{VA} and π^{PA} for the value analysis and the predicate analysis, respectively.

If the outcome of the state-space exploration is the verdict TRUE (the model fulfills the specification) or the verdict FALSE (the model contains a concrete error path) then the analysis terminates. If the model contains an infeasible error path σ, then the model is inconclusive and, according to the CEGAR algorithm, a refinement is initiated.

With the refinement step, the second phase begins, which also marks the starting point of our novel approach for inter-analysis refinement selection. There, for all component analyses, we extract infeasible sliced paths stemming from the infeasible error path σ. Each program analysis provides its own strongest-postcondition SP, with each having different expressive power, and therefore, the set of infeasible sliced paths might differ for each analysis. For example, with

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

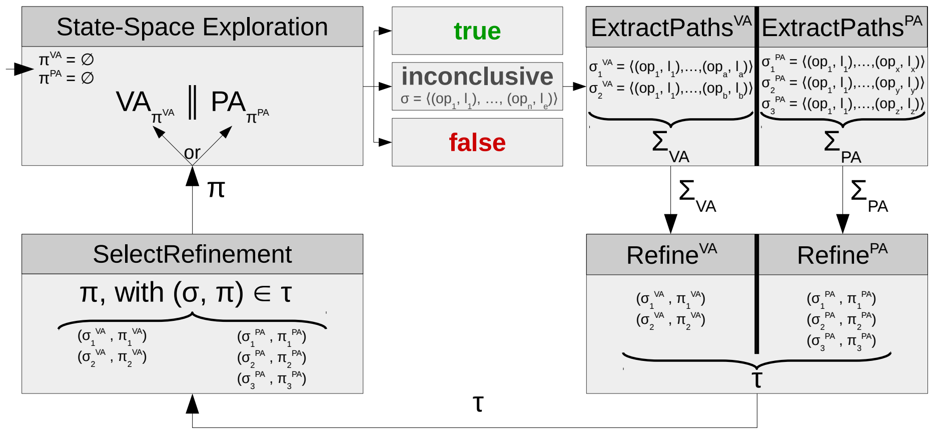

Fig. 2: Refinement selection for combination of analyses, here, consisting of a value analysis and a predicate analysis

$\mathsf{SP}^{\mathsf{VA}}$ we can extract paths that are infeasible due to non-linear arithmetic, while with $\mathsf{SP}^{\mathsf{PA}}$ we get paths that are infeasible due to contradicting range predicates.

In the third phase, for each infeasible sliced path from the previous phase, a precision (i.e., a possible refinement) is computed by delegating to the default refinement routine Refine of the respective analysis. At the end of the third phase, the set τ contains the available refinements (as pairs of infeasible sliced paths and precisions) for all of the component analyses.

In the fourth phase, *one* suitable precision π (in the example, either π^{VA} or π^{PA}) is selected from the set τ, which is added to the respective precision of the component analysis for state-space exploration, finishing one iteration of CEGAR. A proper strategy for refinement selection can be crucial for the effectiveness of the composite analysis, because there is no analysis superior to all other analysis for any given program, but one analysis may be a good fit for one class of programs, but less suitable for another class, while it can be the other way around for a second analysis. Suppose, for example, an infeasible error path that can only by excluded by tracking that a certain variable is within some interval. Refining the value analysis would mean to enumerate all possible values of this variable, whereas the predicate analysis could track this efficiently using inequality predicates. The following evaluation provides evidence that inter-analysis refinement selection can be superior to statically preferring the refinement of one analysis, which is an improvement over our previous work [11].

5 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 [9][3]. We also implemented refinement selection for the predicate analysis [10] in CPAchecker, such that it is now supported for both the value analysis [11] and the predicate analysis.

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. BenchExec [12] was used as benchmarking framework to ensure accurate, reproducible results. We used the tag `cpachecker-1.4.6-spin15` of CPAchecker, and provide the tool, the benchmarks, and the full results on our supplementary web page [4].

Benchmarks. For evaluating the refinement-selection heuristics and our novel combination of analyses, we use a subset of the 5 803 C programs from SV-COMP'15 [6]. We select those tasks that deal with reachability properties, and exclude the categories "Arrays", "HeapManipulation", "Concurrency", and "Recursion", because they are not supported by both analyses we evaluate. Furthermore, we present here only results for those tasks where a refinement *selection* is actually possible, i. e., where at least one refinement with more than one infeasible sliced path is performed. Thus, the set of all verification tasks in our experiments contains 2 828 and 2 638 tasks for the predicate and value analysis, respectively.

Configuration. We use the approach of extracting infeasible sliced prefixes [13] for generating infeasible sliced paths during refinement (method ExtractSlicedPaths in Alg. 1). In order to properly evaluate the effect of the precisions that are chosen by the refinement-selection heuristic, we 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 as seen by the refinement-selection heuristic, but only the precision that is given to the state-space exploration. For the same reason, we also restart the state-space exploration with the refined precision from the initial program location after each refinement. Otherwise, i.e., if we used lazy abstraction and re-explored 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 influence on the performance.

The predicate analysis is configured to use single-block encoding [10], because for larger blocks there is no single error path per refinement, but instead a sequence of blocks which encode a set of potential error paths. As we do not yet have an efficient technique to extract infeasible sliced paths from a sequence of blocks, using refinement selection is not applicable in an ABE configuration. The predicate analysis uses SMTInterpol [15] as satisfiability modulo theories (SMT) solver and interpolation engine.

Refinement-Selection Heuristics. We experiment with implementations of the procedure SelectRefinement in Alg. 1 based on the heuristics from Sec. 3, specifi-

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

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

cally such that it returns the precision for a (1) short or (2) long infeasible sliced path, 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 location. 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 path as a final tie breaker, and select from equally ranked refinements the one with the shortest infeasible sliced path [6].

In the following, we compare the potential of these selection heuristics against each other, as well as against the case where the choice of refinement is solely left to the interpolation engine, i. e., where no refinement selection is performed and the precision extraction is based on the complete, original infeasible error path.

Refinement Selection for Predicate Analysis. We evaluate the presented heuristics for refinement selection when applied to the predicate analysis. Table 1 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 2 828 verification tasks (column "All Tasks") that fit the criterion defined above, 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 3 shows a plot with the quantile functions for the most interesting refinement-selection heuristics on the full set of tasks. In this figure, for each configuration the right end of the graph marks the number of tasks that the configuration could solve, and the area below the graph indicates the sum of the runtime for all solved verification tasks. Thus, in general a graph that is lower and stretches further to the right indicates a better configuration.

Refinement Selection Matters. For the full set of tasks, the analysis without refinement selection performs worse than all other refinement selection heuristics, even worse than the intentionally bad heuristic "bad domain-type score". Figure 3 shows that the analysis without refinement selection scales badly. While it is competitive for easier tasks (below 60 s of CPU time), it solves only a relatively small number of tasks with a runtime between 60 s and 900 s. Additionally, this configuration is not the best for any of the shown subsets, except for "ControlFlowInteger", where it is tied for first with others. This shows that the

[5] We do not expect the precision with a bad 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 path 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 2 828	ControlFlowInt. 35	DD64 679	ECA 1 140	ProductLines 597	Seq. 244
— (No Refinement Selection)		1 142	**34**	473	162	325	43
Length of Sliced Path	Short	1 278	**34**	429	261	**375**	78
	Long	1 325	18	484	322	330	73
Domain-Type Score	Good	1 291	**34**	493	247	339	76
	Bad	1 161	23	404	259	298	79
Width of Precision	Narrow	1 302	28	431	329	347	64
	Wide	1 297	27	480	309	309	76
Depth of Precision	Shallow	1 237	25	466	251	341	57
	Deep	1 260	28	421	313	352	45
Random		1 352	**34**	473	303	350	**86**
Combinations	Good&Narrow	**1 368**	30	**494**	329	338	75
	Narrow&Good	1 354	28	474	**330**	355	65

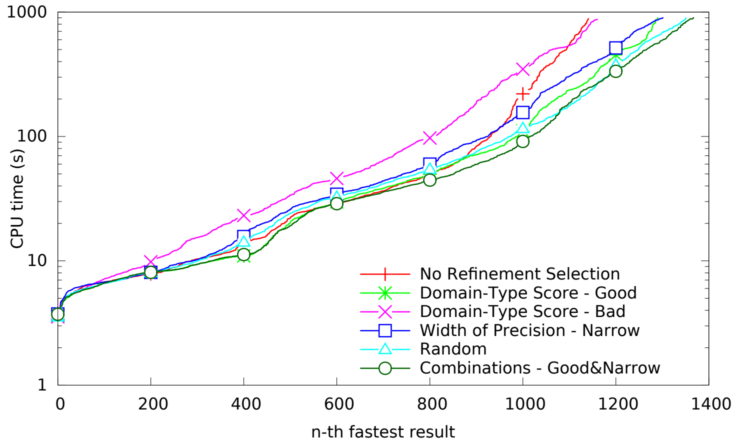

Fig. 3: Quantile plot showing the results for predicate analysis without and with refinement selection using different heuristics

heuristics of the interpolation engine (with which we are stuck without diligent 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 other heuristics.

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 applicable to 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 because it successfully selects precisions with the "easy" and relevant boolean variables. The heuristics "long sliced path", "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 paths 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 in the majority of programs all variables have the same domain type, and thus the heuristic using the domain-type score cannot perform a meaningful selection here and degenerates to a heuristic about the number of distinct variables in the precision. Note also that relying on the interpolation heuristics 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 the heuristic "short sliced path" works especially well here, because 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 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 the order in which the two heuristics are combined, the combination is more successful than the basic configurations if applied to the category of all tasks. The combination with "good domain-type score" as primary and "narrow precision" as secondary heuristic manages to solve 226 (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 if used together with a value analysis. The results are shown in Table 2, which is structured similarly to Table 1, but contains results only 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 the value analysis, as opposed to the predicate analysis,

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)		1 726	408	**575**	453
Length of Sliced Path	Short	1 644	422	488	450
	Long	1 627	484	508	361
Domain-Type Score	Good	1 760	**494**	572	408
	Bad	1 518	410	474	359
Width of Precision	Narrow	1 685	422	507	470
	Wide	1 605	483	491	355
Depth of Precision	Shallow	1 658	471	518	383
	Deep	1 725	414	534	**488**
Random		1 622	433	527	378
Combinations	Good&Narrow	**1 767**	**494**	569	418
	Narrow&Good	1 714	492	507	428

where it is the worst configuration. This can be explained by 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. One notable exception is the subset "ECA", for which the heuristic "good domain-type score" works well for the value analysis, but not for the predicate analysis. The reason for this difference is that the value analysis solves far more instances than the predicate analysis, and for some of the harder "ECA" problems, which the predicate analysis cannot solve, but the value analysis can, there exist variables with different domain-types. Hence, the heuristic "good domain-type score" is 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 not as large as for the predicate analysis.

Refinement Selection for Combination of Analyses. We now evaluate the effectiveness of using refinement selection for a combination of analyses. We compare four different analyses: (1) a sole predicate analysis 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

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

Analysis \ Tasks	All Tasks	DD64	ECA	Loops	ProductLines	Seq.
	3 568	1 245	1 139	120	597	261
PA	1 826	1 027	161	**80**	325	42
VA ‖ PA	2 288	992	495	69	**421**	115
$\text{VA}^+ \parallel \text{PA}^+$	2 386	**1 074**	517	68	404	**126**
$(\text{VA}^+ \parallel \text{PA}^+)^+$	**2 389**	1 068	**519**	79	404	121

value analysis and the predicate analysis is refined only if the value analysis cannot eliminate an infeasible 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 that is defined in Sect. 4 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, 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 [11], 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 a weak spot, as it fails often in "DeviceDrivers64" due to state-space explosion where the predicate analysis alone succeeds. Row "$\text{VA}^+ \parallel \text{PA}^+$" 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 "$(\text{VA}^+ \parallel \text{PA}^+)^+$") 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 eliminate 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 just barely beats 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 "$\text{VA}^+ \parallel \text{PA}^+$"). 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.

6 Conclusion

We presented *refinement selection*, a method that guides the construction of an abstract model in a direction that is beneficial for the effectivity and efficiency of the verification process. The refinement selection works as follows: We start with a given infeasible error path as it occurs in CEGAR. Then, we extract for this infeasible error path a set of sliced paths, and, instead of computing a refinement precision for the original path only, we compute a refinement precision for each sliced path. Next, we assess all refinement precisions according to some heuristics that implement design choices of what is considered a "good" refinement precision. Finally, we select the most promising precision for the model construction.

This paper defines a variety of heuristics for utilizing the potential of refinement selection and we evaluated the ideas on a large benchmark set and two commonly-used verification methods: predicate analysis and value analysis. The experimental results demonstrate 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. Furthermore, if using a combination of a value and a predicate analysis, refinement selection can now be used to systematically select the most appropriate domain for refining the abstract model.

Refinement selection opens a fundamentally new view on verification of models with different characteristics: Instead of using portfolio checking, or trying several different abstract domains, we can, in *one* single tool, fully automatically *self-configure* the verifier, according to the property to be verified and the abstract domain that can best analyze the paths that are encountered during the analysis.

Outlook. It would be interesting to investigate heuristics that use *dynamic* information from the analysis. For example, instead of penalizing a loop-counter variable according to its domain type, we could delay the penalty until a certain threshold is reached on the number of values for this variable, similar to dynamic precision adjustment [8]. Especially for the predicate analysis, it is interesting to investigate heuristics that not only look at the domain type, but also how the variables are referenced in the precision (e.g., an equality predicate for a loop counter usually leads to loop unrolling, an inequality might avoid loop unrolling).

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, V. Levin, and S. K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.
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, and G. Théoduloz. Program analysis with dynamic precision adjustment. In *Proc. ASE*, pages 29–38. IEEE, 2008.
9. D. Beyer and M. E. Keremoglu. CPAchecker: A tool for configurable software verification. In *Proc. CAV*, LNCS 6806, pages 184–190. Springer, 2011.
10. D. Beyer, M. E. Keremoglu, and P. Wendler. Predicate abstraction with adjustable-block encoding. In *Proc. FMCAD*, pages 189–197. FMCAD, 2010.
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. Benchmarking and resource measurement. In *Proc. SPIN*, LNCS. Springer, 2015.
13. 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.
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 *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. Y. Demyanova, H. Veith, and F. Zuleger. On the concept of variable roles and its use in software analysis. In *Proc. FMCAD*, pages 226–230. IEEE, 2013.
20. V. D'Silva, D. Kröning, M. Purandare, and G. Weissenbacher. Interpolant strength. In *Proc. VMCAI*, LNCS 5944, pages 129–145. Springer, 2010.
21. S. Graf and H. Saïdi. Construction of abstract state graphs with Pvs. In *Proc. CAV*, LNCS 1254, pages 72–83. Springer, 1997.
22. T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. Abstractions from proofs. In *Proc. POPL*, pages 232–244. ACM, 2004.
23. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. POPL*, pages 58–70. ACM, 2002.
24. R. Jhala and R. Majumdar. Path slicing. In *Proc. PLDI*, pages 38–47. ACM, 2005.
25. A. V. Khoroshilov, V. Mutilin, A. K. Petrenko, and V. Zakharov. Establishing Linux driver verification process. In *Proc. Ershov Memorial Conference*, LNCS 5947, pages 165–176. Springer, 2009.
26. K. L. McMillan. Interpolation and SAT-based model checking. In *Proc. CAV*, LNCS 2725, pages 1–13. Springer, 2003.
27. P. Rümmer and P. Subotic. Exploring interpolants. In *Proc. FMCAD*, pages 69–76. IEEE, 2013.
28. J. Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *Proc. HCC*, pages 37–39. IEEE, 2002.
29. A. van Deursen and L. Moonen. Understanding COBOL systems using inferred types. In *Proc. IWPC*, pages 74–81. IEEE, 1999.