Systematic Predicate Abstraction using Variable Roles

Yulia Demyanova[1], Philipp Rümmer[2], and Florian Zuleger[1⋆]

[1] Vienna University of Technology
[2] Uppsala University

Abstract. Heuristics for discovering predicates for abstraction are an essential part of software model checkers. Picking the right predicates affects the runtime of a model checker, or determines if a model checker is able to solve a verification task at all. In this paper we present a method to systematically specify heuristics for generating program-specific abstractions. The heuristics can be used to generate initial abstractions, and to guide abstraction refinement through templates provided for Craig interpolation. We describe the heuristics using variable roles, which allow us to pick domain-specific predicates according to the program under analysis. Variable roles identify typical variable usage patterns and can be computed using lightweight static analysis, for instance with the help of of-the-shelf logical programming engines. We implemented a prototype tool which extracts initial predicates and templates for C programs and passes them to the Eldarica model checker in the form of source code annotations. For evaluation, we defined a set of heuristics, motivated by Eldarica's previous built-in heuristics and typical verification benchmarks from the literature and SV-COMP. We evaluate our approach on a set of more than 500 programs, and observe an overall increase in the number of solved tasks by 11.2%, and significant speedup on certain benchmark families.

1 Introduction

Analysis tools, in particular software model checkers, achieve automation by mapping systems with infinite state space to *finite-state abstractions* that can be explored exhaustively. One of the most important classes of abstraction is *predicate abstraction* [13], defined through a set of predicates capturing relevant data or control properties in a program. Picking the right predicates, either upfront or dynamically during analysis [5], is essential in this setting to ensure rapid convergence of a model checker, and is in practice achieved through a combination of "systematic" methods (for CEGAR, in particular through Craig interpolation) and *heuristics.* For instance, SLAM extracts refinement predicates from counterexamples using domain-specific heuristics [16]; YOGI uses machine learning to choose the default set of heuristics for picking predicates [19]; CPAchecker

⋆ The first and third author were supported by the Austrian National Research Network S11403-N23 (RiSE) of the Austrian Science Fund (FWF).

uses domain types to decide whether to represent variables explicitly or using BDDs [2], and to choose refinement predicates [4]; and ELDARICA uses heuristics to guide the process of Craig interpolation [18]. Similar heuristics can be identified in tools based on abstract interpretation, among others.

The goal of the present paper is to systematise the definition of abstraction heuristics, and this way enable easier and more effective adaptation of analysis tools to specific domains. In order to effectively construct program abstractions, it is essential for an analysis tool to have (semantic) information about variables and data-structures used in the program. We propose a methodology in which heuristics are defined with the help of *variable roles* [9], which are features capturing typical variable usage patterns and which can be computed through lightweight static analysis. Knowledge about roles of variables can be used to generate problem-specific parameters for model checkers, or other analysis tools, and thus optimise the actual later analysis process.

As a case study, we describe how variable roles can be used to infer code annotations for the CEGAR-based model checker ELDARICA [20]. ELDARICA has two main parameters controlling the analysis process: *initial predicates* for predicate abstraction, and *templates* guiding Craig interpolation during counterexample-based refinement [18]. Both parameters can be provided in the form of source-code annotations. We focus on the analysis of C programs defined purely over integer scalar variables, i.e., not containing arrays, pointers, heap-based data structures and bitvectors. By manually inspecting a (small) sample of such programs from SV-COMP [3], we were able to identify a compact set of relevant variable roles, and of heuristics for choosing predicates and templates based on those roles. To evaluate the effectiveness of the heuristics, we compared the performance of ELDARICA (with and without the heuristics), and of other model checkers on a set of over 500 programs taken from the literature and SV-COMP. We observe an increase in the number of solved tasks by 11.2% when using our heuristics, and speedups on certain benchmark families.

Contributions of the paper are: 1. We introduce a methodology for defining abstraction heuristics using variable roles; 2. we define 8 roles and corresponding heuristics for efficiently analysing C programs with scalar variables; 3. we implement our approach and perform an extensive experimental evaluation.

Related Work Patterns of variable usage were studied in multiple disciplines, e.g. in teaching programming languages [21] (where the patterns were called *variable roles*), in type systems for inferring equivalence relations for types [22], and others. In [9] a set of patterns, also called variable roles, was defined using data-flow analysis, based on a set of C benchmarks[3]. In [7, 8] variable roles were used to build a portfolio solver for software verification. Similarly to variable roles, code patterns recognised with light-weight static analyses are used in the bug-finding tool Coverity [11] to devise heuristics for ranking possible bugs. Domain types in CPACHECKER [4] can be viewed as a restricted class of variable roles. Differently from this work, where variable roles guide the generation of interpolants, the domain types are used in [4] to choose the "best" interpolant

[3] http://ctuning.org/wiki/index.php/CTools:CBench

```
1 extern char nondet_char();
2 void main() {
3   int id1 = nondet_char();
4   int id2 = nondet_char();
5   int id3 = nondet_char();
6   int max1=id1, max2=id2, max3=id3;
7   int i=0, cnt=0;
8
9   assume(id1!=id2 && id1!=id3 &&
10    id2!=id3);
11
12  while (1) {
13    if (max3 > max1) max1 = max3;
14    if (max1 > max2) max2 = max1;
15    if (max2 > max3) max3 = max2;
16
17    if (i == 1) {
18      if (max1 == id1) cnt++;
19      if (max2 == id2) cnt++;
20      if (max3 == id3) cnt++;
21    }
22    if (i>=1) assert(cnt==1);
23    i++;
24  }
25 }
```

(1) Roles input, dynamic enumeration and extremum

```
1 extern int nondet_int();
2 int main() {
3   int n = nondet_int();
4   int k, i, j;
5
6   for (k=0,i=0; i<n; i++,k++);
7   for (j=n; j>0; j--,k--) {
8       assert(k > 0);
9   }
10  return 0;
11 }
```

(2) Role local counter

Fig. 1: Motivation examples illustrating variable roles.

from a set of generated interpolants. In addition, our method generates role-based initial predicates, while the method of [4] does not.

There has been extensive research on tuning abstraction refinement techniques, in such a way that convergence of model checkers is ensured or improved. This research in particular considers various methods of Craig interpolation, and controls features such as interpolant strength, interpolant size, the number of distinct symbols in interpolants, or syntactic features like the magnitude of coefficients; for a detailed survey we refer the reader to our previous work [18].

1.1 Introductory Examples of Domain-Specific Abstraction

We introduce our approach on two examples. These and all further examples in this paper are taken from the benchmarks of the software competition SV-COMP'16 [3]. We simplified some of the examples for demonstration purposes.

Motivation example 1. The code in Fig. 1.1 initializes variables `max1`, `max2` and `max3` to `id1`, `id2` and `id3` respectively, which are in turn initialized non-deterministically. The `assume` statement at lines 9-10 is an Eldarica-specific directive, which puts a restriction that control reaches line 12 only if `id1!=id2 && id1!=id3 && id2!=id3` evaluates to `true`. In the loop the value `max{id1,id2,id3}`, which is the maximum of `id1`, `id2` and `id3` is calculated: At the first iteration, `max1` is assigned the value `max{id1,id3}`, and `max2` and `max3` are assigned the value `max{id1,id2,id3}`. After the second iteration `max1`, `max2` and `max3` all store the value `max{id1,id2,id3}`. Since `id1`, `id2` and `id3` have

distinct values, only one of the conditions in lines 19-21 evaluates to `true`. The assertion checks that the value of exactly one of variables `max1`, `max2` and `max3` remains unchanged after two iterations, namely $\texttt{max}_i$, where $\texttt{i}=\arg\max_j\{\texttt{id}_j\}$.

It takes ELDARICA 27 CEGAR iterations and 19 sec to prove the program safe. However, for 88 out of 108 original programs from SV-COMP with this pattern in category "Integers and Control Flow", of which the code in Fig. 1.1 is a simplified form[4], ELDARICA does not give an answer within the time limit of 15 minutes. Predicate abstraction needs to generate for these programs from 116 to 996 predicates, depending on the number of values, for which the maximum is calculated. Since predicates are added step-wise in the CEGAR loop, checking these benchmarks is time consuming. We therefore suggest a method of generating the predicates upfront.

In order to prove that exactly one condition in lines 18-20 evaluates to `true` and `cnt` is incremented by one, predicate abstraction needs to track the values assigned to variables `max1`, `max2` and `max3` with 9 predicates: `max1==id1`, `max1==id2`, `max1==id3`, etc. Additionally, in order to precisely evaluate conditions in lines 13-15, abstraction needs to track the ordering of variables `id1`, `id2` and `id3` with 6 predicates which compare variables `id1`, `id2` and `id3` pairwise: `id1<id2`, `id1>id2`, and so on.

To generate the above mentioned 15 predicates our algorithm uses the following variable roles. Variable is *input* if it is assigned a return value of an external function call. This pattern is often used in SV-COMP to initialize variables non-deterministically, e.g. `id1=nondet_char()`, where variables `id1`, `id2`, `id3` are inputs. Variables which are assigned only inputs are run-time analogues of compile-time enumerations. A variable is *dynamic enumeration* if it is assigned only constant values or input variables, i.e. variables `max1`, `max2` and `max3` are dynamic enumerations. For each dynamic enumeration `x` which takes values `v1`,...,`vn`, our algorithm generates n equality predicates: `x==v1`, ..., `x==vn`.

Variable `x` is *extremum* if it is used in the pattern `if(comp_expr)x = y`, where `comp_expr` is a comparison operator `>` or `<` applied to `y` and some expression `expr`, e.g. `y>expr`. For every variable `x` which is both dynamic enumeration and extremum, our algorithm generates pairwise comparisons for all pairs of input values `v1`,...,`vn` assigned to `x`, e.g. `v1<v2`, `v1>v2`, and so on.

ELDARICA proves the program in Fig. 1.1 annotated with the 15 predicates in 8 sec and 0 CEGAR iterations, and it takes ELDARICA from 21 to 858 sec (and from 0 to 4 CEGAR iterations) to prove 53 programs from SV-COMP with this pattern annotated analogously. For the remaining 55 benchmarks with this pattern from SV-COMP the number of abstract states becomes too large for ELDARICA to be checked within the time limit.

Motivation example 2. The code in Fig. 1.2 increments variables `i` and `k` in the loop at line 6 until `i` reaches `n`, and decrements variables `j` and `k` in the loop at lines 7–9 until `j` reaches `0`. The assertion checking that the value of variable `k` remains positive in the loop can be proven using the predicates

[4] e.g. `seq-mthreaded/pals_opt-floodmax.3_true-unreach-call.ufo.BOUNDED-6.pals.c`

`k>=i` and `k>=j`. These predicates are difficult to find, e.g., the baseline version of ELDARICA [20] keeps generating a sequence of pairs of predicates `(i<=1,k<=1)`, `(i<=2,k<=2)`, etc. As demonstrated by this example, heuristics are needed to guide interpolation towards finding suitable refinement predicates. The community has suggested various heuristics for the above example, e.g., the most recent version of ELDARICA [18] proves the program safe in 5 sec and 6 CEGAR iterations.

We suggest to generate predicate templates *demand-driven* from the code under analysis. For the above example, we propose a heuristic which tracks the dependencies between loop counters: The heuristic searches for variables `x` assigned in a loop in a statement matching the pattern `x=x+expr`, where `expr` is an arbitrary expression. For each pair `x1` and `x2` of such variables the heuristic generates a predicate template `x1-x2`. This template restricts the search space of the interpolation solver to predicates of the form `x1-x2>=n,` $\mathtt{n} \in \mathbb{N}$. To formalise the heuristic we introduce the following role: *local counter* is a variable assigned in a loop in a statement `x=x+expr`, where `expr` is an arbitrary expression. Note that we do not restrict `expr` to be a constant, in contrast to *induction variables* [1], since the heuristic is a trade-off between generality and computational cost and performs well in practice.

Methodology for choosing roles. To choose roles and role-based predicates and templates, we investigated benchmarks of the competition SV-COMP'16 from categories "Integers and Control Flow" and "Loops" and loop invariant generation benchmarks (appr. 30 benchmarks altogether) on which ELDARICA did not give an answer within the time limit of 15 minutes. We manually inspected the code of these benchmarks and annotated the benchmarks with a minimum set of predicates and templates so that ELDARICA checks the benchmarks within the time limit. We then derived new variable roles which captured specific code patterns in which the annotated variables were used.

2 Predicate Abstraction and Refinement

We outline the algorithm implemented by predicate abstraction-based software model checkers, in particular the ELDARICA tool [20] used as test-bed. As the core procedure, ELDARICA applies predicate abstraction [13] and counterexample-guided abstraction refinement [5] to check the satisfiability of *Horn constraints* expressing safety properties of a software program [14, 20, 15]. The procedure has two main parameters that can be used to tune the abstraction process:

- **initial predicates** Π_0 for predicate abstraction (see Sect. 2.1);
- **interpolation templates** T that guide Craig interpolation towards meaningful predicates during abstraction refinement (see Sect. 2.2).

The pair (Π_0, T) can be computed with the help of variable roles, as outlined in the previous section. It is important to note that neither parameter has any effect on *soundness* of a model checker, only termination is affected.

2.1 Solving Horn Clauses with Predicate Abstraction

A *Horn clause* is a formula of the form $\varphi \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$, with constraint φ, body literals $B_1 \wedge \cdots \wedge B_n$ containing uninterpreted *relation symbols,* and head literal H. Eldarica has a C/C++ front-end that translates software programs to sets HC of Horn clauses. In this setting, relation symbols represent state invariants Inv_c associated with a control location c of a program, and Horn clauses express 1. pre-conditions $Pre(\bar{s}) \rightarrow Inv_c(\bar{s})$ for program entry points c; 2. Floyd-style inductiveness conditions $T(\bar{s}, \bar{s}') \wedge Inv_c(\bar{s}) \rightarrow Inv_{c'}(\bar{s}')$, for transitions between control locations c, c'; and 3. safety assertions $\neg P(\bar{s}) \wedge Inv_c(\bar{s}) \rightarrow \mathit{false}$ for control locations c. The translation from software programs to Horn clauses HC is defined such that the program is *safe* if and only if the clauses HC are *satisfiable,* i.e., if and only if the predicates Inv_c can be interpreted in such a way that all clauses become valid.

Model checkers like HSF [14] or Eldarica [20] construct solutions of Horn clauses in disjunctive normal form by building an *abstract reachability graph* (ARG) over a set of given predicates. For this, a Horn solver maintains a mapping $\Pi : \mathcal{R} \rightarrow \mathcal{P}_{\text{fin}}(For)$ from relation symbols $p \in \mathcal{R}$ to finite sets of predicates. The solver starts from some initial mapping $\Pi = \Pi_0$; for instance, mapping every relation symbol to an empty set of predicates. The solver will then attempt to construct a closed ARG by means of fixed-point computation, which can either succeed (in which case a solution of the Horn clauses has been derived), or fail because some assertion clause $\varphi \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow \mathit{false}$ is violated during the construction. In the latter case, a connected acyclic ARG fragment can be extracted that leads from entry clauses (clauses $\varphi \rightarrow H$ without relation symbols in the body) to the violated assertion clause. A theorem prover is then used to verify that the counterexample is genuine; spurious counterexamples are eliminated by generating additional predicates by means of Craig interpolation, leading to an extended mapping $\Pi = \Pi_1$ and refined abstraction.

2.2 Craig Interpolation with Templates

Predicate abstraction-based model checkers rely on theorem provers to find suitable interpolants, or interpolants containing the right predicates, in a generally infinite lattice of interpolants for every extracted counterexample (represented as acyclic ARG fragments). Eldarica uses *interpolation abstraction* [18] as a semantic way to guide the interpolation procedure towards "good" interpolants; in this method, interpolation queries are instrumented to restrict the symbols that can occur in interpolants, ranking the interpolants with the help of *templates.* It has previously been shown that interpolation abstraction can significantly improve the performance of Horn solvers [18].

In the scope of this paper, we focus on templates in the form of *terms.* As an example, consider the binary interpolation query $A \wedge B$ with $A = (x = 1 \wedge y = 2)$ and $B = (x > y)$. The interpolation problem has multiple solutions I (with the property that $A \Rightarrow I$ and $B \Rightarrow \neg I$), including $I_1 = (x = 1 \wedge y = 2)$ and $I_2 = (y = x + 1)$. In a software model checker, clearly I_2 is preferable, since it abstracts

from concrete values of the variables. Interpolation abstraction can be used to distinguish between I_1 and I_2, by preventing theorem provers, e.g., to compute I_1 as an interpolant. For this, template terms are used to capture the expressions that an interpolant might contain. In the example, given templates $\{x, y\}$, a theorem prover could compute either of I_1, I_2; with the template $\{x - y\}$, a theorem prover could return $(x - y = -1) \equiv I_2$, but no longer I_1.

In ELDARICA, software programs can be annotated to express preference of certain interpolants. For instance, line 4 of the code in Fig. 1.2 can be annotated to express that the differences `i-k` and `j-k` are preferred templates:

```
4 int k, /*@ terms_tpl {i-k} @*/ i, /*@ term_tpl{j-k} @*/ j;
```

Annotations are attached to variable declarations, and are then applied when computing interpolants at control points in the scope of the variable. If no interpolant can be constructed using this template, a conventional interpolant will be used. Besides manual annotation, ELDARICA also has a set of inbuilt heuristics to choose meaningful templates automatically [18].

3 Role-based Predicates and Templates

Specification language for roles. In this section we describe a framework for the specification and computation of role-based initial predicates and predicate templates. Roles are usage patterns of variables, we introduce and formalize them as data-flow analyses in our previous work [9]. Here we re-formulate roles as logic queries on the control-flow graph (CFG) of a program. We choose logic programming as a formalism for two reasons: first, its notation is well known, and second, we can use of-the-shelf logic engines for the computation of roles. Specifically, we use the syntax and standard fixed point semantics of Datalog.

Preliminaries on Datalog. A rule in Datalog is of the form $\texttt{A}_0\texttt{:-L}_1, \ldots, \texttt{L}_n$. The head of a rule $\texttt{A}_0$ is an atom. The body of a rule $\{\texttt{L}_i\}$ is a set of literals, and each literal $\texttt{L}_i$ is of the form `A` or `not A` for an atom `A`, where the connective `not` corresponds to default negation. An atom takes boolean values and is of the form 1. $\texttt{p(t}_1, \ldots, \texttt{t}_m\texttt{)}$, or 2. $\texttt{t}_0\texttt{=f(t}_1, \ldots, \texttt{t}_k\texttt{)}$, or 3. $\texttt{t}_1$ `op` $\texttt{t}_2$, where `p` is a predicate symbol, `f` is a function symbol, $\texttt{t}_j$ are term symbols and `op` is a comparison operator (e.g. `>, !=`, etc.). Atom $\texttt{t}_0\texttt{=f(t}_1, \ldots, \texttt{t}_k\texttt{)}$ always evaluates to `true` and assigns to term $\texttt{t}_0$ the result of function $\texttt{f(t}_1, \ldots, \texttt{t}_k\texttt{)}$. Each term $\texttt{t}_j$ is a constant symbol (i.e. a function symbol with arity 0), a variable, or an integer. Predicate and function symbols start with a small letter, and variables start with a capital letter. A rule is evaluated as follows: if every literal $\texttt{L}_i$ in the body evaluates to `true`, then the atom $\texttt{A}_0$ in the head evaluates to `true`. A rule with empty body is called a fact.

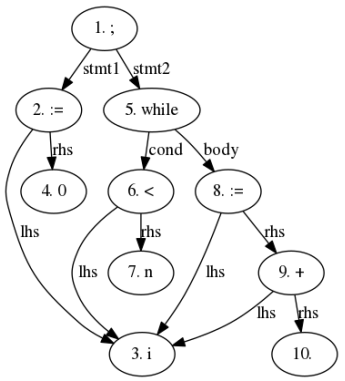
Translation of C code to a logic program. We assume a C program to be given as a logic program, where each node and edge in the control-flow graph is translated to one or more facts in the logic program. For example, the code in Fig. 2a is translated to a logic program in Fig.2b (see the CFG in Fig. 2c). In particular, the loop condition `i<n` is represented with nodes 6, 3 and 7 in the

```
1 for(i=0; i<n; i++);
```

(a) Source code

```
1  sequence_stmt(1).
2  stmt1(1,2).
3  stmt2(1,5).
4  assign_stmt(2).
5  lhs_expr(2,3).
6  rhs_expr(2,4).
7  var(3).
8  name(3,"i").
9  const_literal(4).
10 text(4,"0").
11 while_stmt(5).
12 cond(5,6).
13 body(5,8).
14 bop(6).
15 opcode(6,"<").
16 lhs_expr(6,3).
17 rhs_expr(6,7).
18 var(7).
19 name(7,"n").
20 assign_stmt(8).
21 lhs_expr(8,3).
22 rhs_expr(8,9).
23 bop(9).
24 opcode(9,"+").
25 lhs_expr(9,3).
26 rhs_expr(9,10).
27 const_literal(10).
28 text(15,"1").
```

(b) Logic program



(c) Control flow graph

Fig. 2: Translation of C code to a logic program

CFG and lines 7-8 and 15-19 in the logic program. Below we will denote a node corresponding to variable `x` in the control-flow graph with $\texttt{node}_x$.

We define roles *local counter*, *extremum*, *input* and *dynamic enumeration* in Fig. 3. Specifically, in Fig. 3a we define role local counter which is used to generate templates, and in Fig. 3b we define roles which are used to generate initial predicates. Due to the lack of space we introduce the remaining roles and the generated predicates and templates informally in Table 1. We explain the definitions of roles in Section 3.1, and the generation of predicates and templates for these roles in Section 3.2.

3.1 Definition of Roles

Role local counter. Role local counter (line 2-4 in Fig. 3) is defined in the scope of one loop. The set of variables to which this role is ascribed is encoded with a binary relation `local_cnt` with a parameter corresponding to the resp. loop statement `WhileStmt`. The parameter is needed, because we later define a template for pairs of local counters, such that the counters have the same parameter. A variable `X` is ascribed role local counter if `X` is there is a loop statement `WhileStmt`, in the body of which `X` is assigned the sum of `X` and some other expression. Term `sub_stmt(Stmt,SubStmt)` encodes that in the control flow graph `SubStmt` is a descendant of `Stmt`. Term `assigned(X,Expr,AsgnStmt)` encodes that variable `X` is assigned expression `Expr` in statement `AsgnStmt`. Term `operand(Expr,Bop)` encodes that `Expr` is an operand of binary operator `Bop`. For example, for code in Fig. 2a the evaluation of the rule derives the fact `local_cnt(3)` for node $\texttt{node}_i$=3. For clarity we omit rules for terms `sub_stmt`, `assigned`, `operand` and a rule for the case when the counter is decremented.

Role extremum. Role extremum (lines 9-11) is ascribed to variable `X`, denoted with term `extremum(X)`, if there is an if statement `IfStmt`, the condition `Cond` of which is a binary operator greater-than or less-than (encoded with term `rel_opcode(Opcode)`), s.t. `Cond` contains a variable `Y` which is assigned to `X` in

```
% local counter
local_cnt(X,WhileStmt):- while_stmt(WhileStmt),
  sub_stmt(WhileStmt,AsgnStmt), assigned(X,SumExpr,AsgnStmt),
  bop(SumExpr), opcode(SumExpr,"+"), operand(SumExpr,X).

% difference templates for local counters
tpl(TplStr):-local_cnt(X,WhileStmt),local_cnt(Y,WhileStmt),
  X!=Y, name(X,Xname), name(Y,Yname), TplStr=@concat(Xname,"-",Yname).
```

(a) Role *local counter* and templates.

```
% extremum
extremum(X):- if_stmt(IfStmt), condition(IfStmt,Cond), bop(Cond),
  opcode(Cond,Opcode), strict_rel_opcode(Opcode), operand(Cond,Y),
  var(Y), assigned(X,Y,AsgnStmt), then(IfStmt,AsgnStmt).

% input
input(X):- assigned(X,CallExpr,AsgnStmt), call_expr(CallExpr),
  function(CallExpr,Func), not body(Func).

% dynamic enumerations
dyn_enum(X):- var(X), not not_dyn_enum(X).
% the complement of dyn_enum
not_dyn_enum(X):- assigned(X,Y,AsgnStmt), var(Y), not_dyn_enum(Y).
not_dyn_enum(X):- assigned(X,Expr,AsgnStmt), not var(Expr),
  not dyn_enum_expr(Expr).
% cases for dynamic enumerations
dyn_enum_expr(Expr):- const_literal(Expr).
dyn_enum_expr(Expr):- input(Expr).

% predicates for dynamic enumerations
pred(PredStr):- dyn_enum(X), assigned(X,Y), var(Y),
  name(X,Xname), name(Y,Yname), PredStr=@concat(Xname,"==",Yname).

% ordering predicates for dynamic enumerations
pred(PredStr):- extremum(X), dyn_enum(X), assigned(X,Y),
  var(Y), assigned(X,Z), var(Z), Y!=Z, name(Y,Yname),
  name(Z,Zname), PredStr=@concat(Yname,"<",Zname).
```

(b) Roles *dynamic enumeration*, *input* and *extremum*, and initial predicates.

Fig. 3: Simplified specification of roles and role-based templates and initial predicates.

the body of `IfStmt`. For example, for code `if (max3>max1) max1=max3` (line 13 in Fig. 1.1), the result of evaluating the rule is `extremum(node`$_{max1}$`)`. Relation `rel_opcode` encodes that its parameter is a greater-than or less-than operator.

Role input. Role input (lines 14-15) is ascribed to variable `X` if `X` is assigned the result of a call `CallExpr` to a function `Func`, the body of which is not defined (encoded with atom `not body(Func)`). For example, for the C code

Table 1: Informal description of remaining roles with examples.

Role name	#	**Description of role**	Π/ T	**Example**	
				Code	**Generated predicates Π /templates T**
Assertion condition	1	Variable is used in pattern `assert(expr)`	Π = {`expr`}	`assert( cnt==1)`	Π ={`cnt==1`}
	2	Statement `assert(expr)` is nested in an `if` statement with condition `cond`	Π = {`cond`}	`if(x<1) assert(0)`	Π ={`x<1`}
Parity variable	3	Variable `x` is used in remainder operator `x%c`	T = {`x%c`}	`x%2`	T ={`x%2`}
	4	Variable `x` is incremented in a loop by constant `c`, s.t. `c!=1`	T = {`x%c`}	`for(i=0;i<n; i+=2)`	T ={`x%2`}
Loop iterator	5	Variable `x` is modified in a loop and is used in the loop condition `cond`	Π = {`cond`}	`while(i<n) i++`	Π ={`i<n`}
	6	In addition to 5), `cond` matches pattern `expr1!=expr2`	Π = {`expr1<expr2`, `expr1>expr2`}	`for(i=0; i!=n;i++)`	Π ={`i<n`, `i>n`}
	7	In addition to 5), `cond` matches pattern `expr1<expr2` (resp. `expr1>expr2`) and loop iterator is changed by 1 in the loop	Π = {`expr1<=expr2`} (resp. {`expr1>=expr2`}).	`for(i=0;i<n; i++)`	Π ={`i<=n`}
Loop bound	8	Variable `bnd` is compared to loop iterator `it` in loop condition: `it`$\circ$`bnd`, where $\circ$ $\in$ {`<`,`<=`,`>`,`>=`,`!=`,`==`}; and `bnd` is assigned in statement `bnd=expr`	Π = {`bnd<=expr`, `bnd>=expr`}	`n=k-2; for(i;i<n; i++);`	Π ={`n<=k-2`, `n>=k-2`}

`id11=nondet_char()` where `nondet_char()` is defined as an external function (lines 1 and 3 in Fig. 1.1), evaluation of the rule derives fact `input(node`$_{id1}$`)`.

Role dynamic enumeration. Role dynamic enumeration (lines 18-22) is defined via its complement `not_dyn_enum` (line 18). Fact `not_dyn_enum(X)` is generated if variable `X` is assigned an expression `Expr` which does not belong to relation `dyn_enum_expr` (line 19). The unary relation `dyn_enum_expr` includes constant literals and input and dynamic enumeration variables (lines 20-22). For example, for code in Fig.1.1 evaluation of rules derives facts `dyn_enum(node`$_{max1}$`)`, `dyn_enum(node`$_{max2}$`)` and `dyn_enum(node`$_{max3}$`)`.

3.2 Role-based Predicates and Templates

Our algorithm generates initial predicates Π_{roles} = {`p` | `pred(p)`} and templates T_{roles} = {`t` | `tpl(t)`}, where `pred(p)` and `tpl(t)` are the facts derived by the

Table 2: Characteristics of the benchmarks

#	Name	Number of files			Size, KLOC
		Total	Safe	Unsafe	
1	SV-COMP CFI	234	91	143	226.4
2	SV-COMP Loops	95	68	27	6.5
3	VeriMAP	153	133	20	13.2
4	Llreve	21	16	5	0.6
5	HOLA	46	46	0	1.4
Total		549	354	195	248.0

Table 3: ELDARICA configurations. T_{Eld} denotes the templates generated by built-in heuristics of ELDARICA.

Name	Π_0	T
Eld	$\emptyset$	$\emptyset$
Eld+B	$\emptyset$	T_{Eld}
Eld+R	Π_{roles}	T_{roles}
Eld+BR	Π_{roles}	$T_{roles} \cup T_{Eld}$

logic program (see line 7 in Fig. 3a and lines 21-22 and 25-27 in Fig. 3b). We now describe the role-based initial predicates and templates in detail.

Local counter. For every pair of local counters `X` and `Y` s,t. `X` and `Y` are modified in loop `WhileStmt`, a template `X-Y` is derived (lines 5-6). For example, for code in Fig. 1.2 the evaluation of the rule derives templates `i-k` and `j-k`.

Dynamic enumeration. For every pair of a dynamic enumeration `X` and input `Y`, s.t. `Y` is assigned to `X`, predicate `X==Y` is derived (lines 23-24). Term `@concat` encodes a call to a function which concatenates its parameters. For example, for code in Fig. 1.1 the evaluation of the rule derives predicates `max1==id1`, `max2==id2` and `max3==id3`.

Input variables. For every pair of input variables `Y` and `Z`, s.t. both `Y` and `Z` are assigned to dynamic enumeration and extremum `X`, predicate `Y<Z` is derived (lines 25-27). For example, for code in Fig.1.1 the evaluation of rules derives predicates `id1<id2`, `id1>id2`, `id1<id3`, `id1>id3`, `id2<id3` and `id2>id3`.

4 Evaluation

We implemented our approach in a prototype tool and evaluated the tool on altogether 549 C benchmarks[5].

Benchmarks. Table 2 lists the benchmarks and gives their characteristics. Specifically, the benchmarks contain (listed in the same order as in Table 2):

1. Benchmarks of the competition SV-COMP'16 from the "Integers and Control Flow" category. We excluded the Recursive sub-category and 75 benchmarks which contain C structures and arrays;
2. Benchmarks from the Loops category of SV-COMP'16 (we excluded 50 benchmarks for same reasons);
3. Benchmarks of the verification tool *VeriMAP*[6]. We excluded 234 duplicate benchmarks contained in SV-COMP CFI, and 2 benchmarks, for which the transition relations cannot be expressed with Presburger arithmetic;

[5] The tool, the set of used benchmarks and the results of our evaluation are available at http://forsyte.at/software/demy/nfm17.tar.gz

[6] http://map.uniroma2.it/vcgen/benchmark320.tar.gz

4. Simplified versions[7] of the benchmarks of tool llrêve for automated program equivalence checking [12];
5. Loop invariant generation benchmarks of the verication tool HOLA [10].

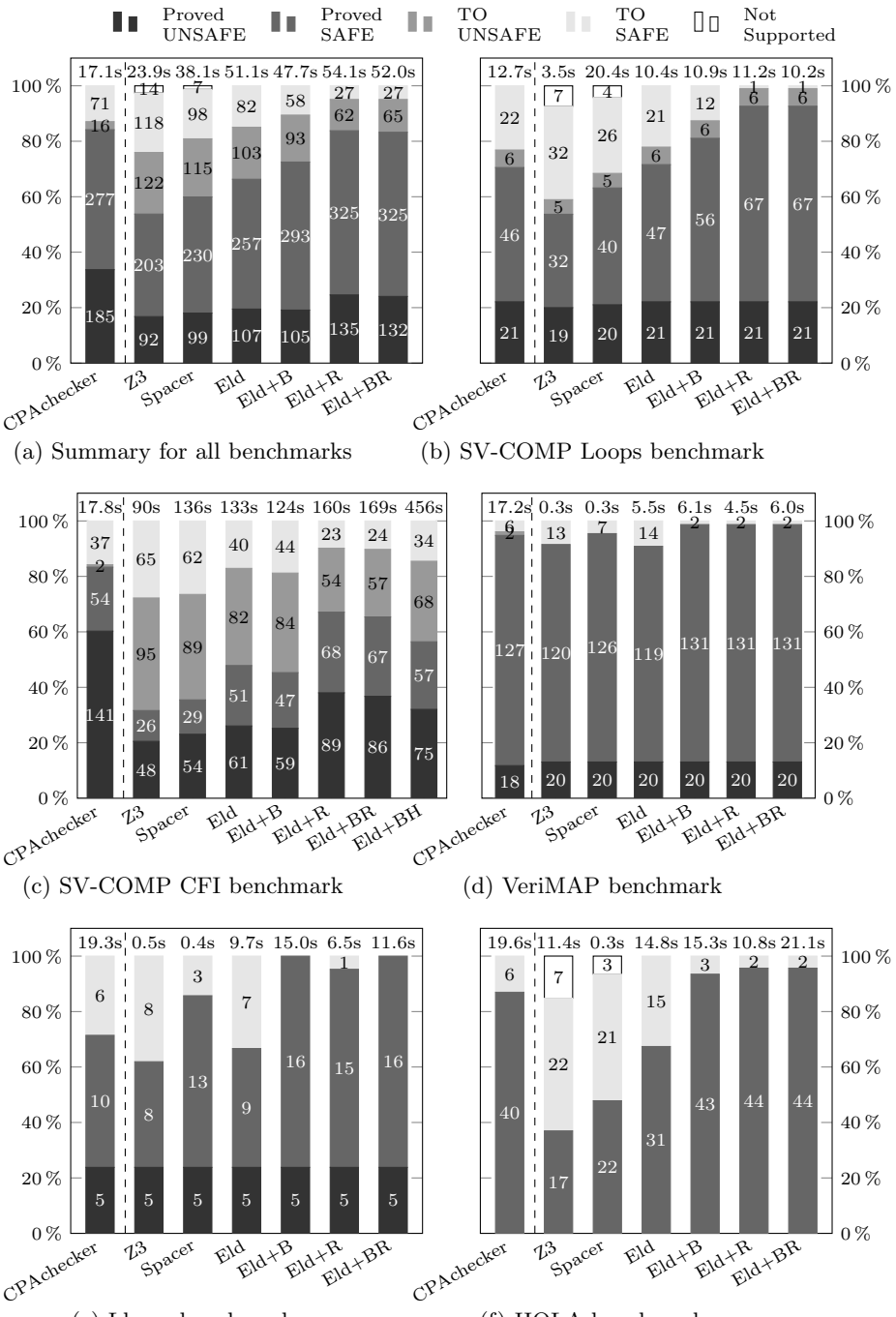
Tools for comparison. We evaluate the following configurations of ELDARICA: without interpolation abstraction (to which we refer by Eld), with templates (Eld+B), with roles (Eld+R), and with a combination of templates and roles (Eld+BR). Table 3 lists different choices for the parameters Π_0 and T described in Section 2. As a baseline we also compare ELDARICA to SMT solvers Z3 [6] and Spacer [17]. We could not compare to the duality engine of Z3 because of a bug in duality, which was not fixed by the time of paper submission. Finally, we compare ELDARICA to the model checker CPACHECKER, which is not based on Horn clauses. CPACHECKER has very successfully participated in the software competition in the recent years and thus provides an interesting choice for comparison.

Experimental setup. We performed our experiments on 2.0GHz AMD Opteron PC (31GB RAM, 64KB L1 cache, 512KB L2 cache). We did not restrict the number of cores on which the tasks were performed. We report the wall-clock time measured using the `date` shell utility. For evaluation we set the value of timeout for all tools to 15 minutes, which is the value of the timeout in the SV-COMP competition. We put no memory limit on the tools.

Overall improvement of Eldarica. The results of our evaluation are represented in Fig. 4, which shows the number of solved and unsolved tasks, with safe and unsafe tasks counted separately. Specifically, Fig. 4a gives a summary for all benchmarks, and Figures 4b-4f show detailed results for each benchmark. In the bar plots on top of each bar is the mean runtime of the respective tool, calculated *without timeouts*. The times for Eld+R include the times for computing roles: the mean and median time of annotating a program for all benchmarks amount to 3.8 sec and 0.8 sec resp. We observe that the best configuration of Eldarica is Eld+R, which solves the highest number of tasks for every benchmark separately and for all benchmarks. The second best configuration for most benchmarks is Eld+B. Overall Eld+R solves 11.2% more tasks than Eld+B: 4.6% more safe and 6.6% more unsafe tasks. *We conclude that the configuration Eld+R improves on the previous configurations of Eldarica (Eld and Eld+B).*

Comparison of runtimes. Overall, the runtime of Eld+R is comparable to the runtime of other Eldarica's configurations, but for the benchmarks SV-COMP CFI we observe a significant speedup of Eld+R, as shown in Fig. 5. SV-COMP CFI is a specific family of benchmarks because of their big size and a large number of enumeration variables, see e.g. the code in Fig. 1.1. Note that in Fig. 5 we compare Eld+R to Eld, which is the second best configuration, because for these benchmarks no heuristics are needed. The speedup of Eld+R for SV-COMP CFI is caused by a considerable decrease in the number of CEGAR iterations. To demonstrate this, we evaluate the configuration Eld+B with the timeout value of one hour (denoted as Eld+BH in Fig. 4c). We observe that

[7] Original benchmarks are accessible at http://formal.iti.kit.edu/projects/improve/reve and https://www.matul.de/reve



(a) Summary for all benchmarks

(b) SV-COMP Loops benchmark

(c) SV-COMP CFI benchmark

(d) VeriMAP benchmark

(e) Llreve benchmark

(f) HOLA benchmark

Fig. 4: Bar plots comparing the percentage of proved tasks for Z3 and different Eldarica configurations. Inside each bar is the percentage of the resp. answers. On top of each bar is the mean runtime computed *without timeouts* (for solved tasks).

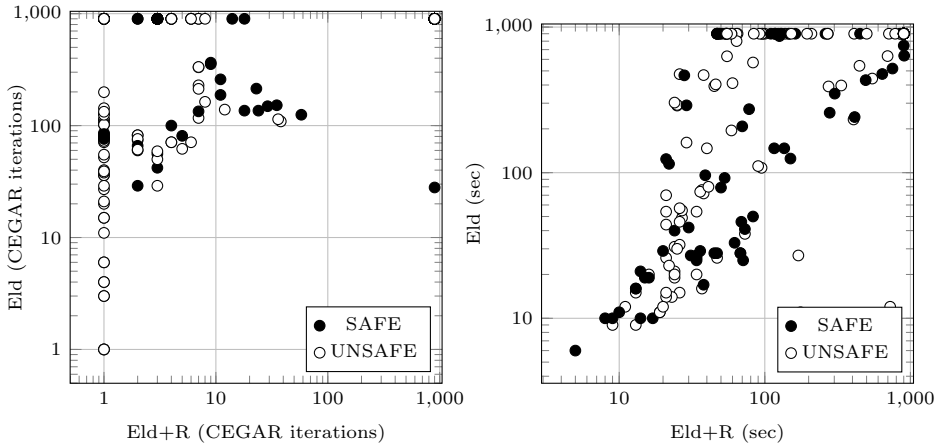


Fig. 5: Scatter plots comparing the number of CEGAR iterations and runtime, both in logarithmic scale, of configurations Eld+R and Eld for benchmark SV-COMP CFI. The mean runtime of Eld+R is 1.5 times smaller than that of Eld, and the average number of CEGAR iterations of Eld+R is 19.0 times smaller than that of Eld, the four values calculated on the tasks solved by both Eld and Eld+R.

Eld+BH solves 12.8% more unsafe and 9.0% more safe tasks than Eld+B. *To conclude, Eld+R does not increase the runtime on all benchmarks, and even shows a significant speedup for the family of benchmarks from SV-COMP CFI.*

Comparison of roles with Eldarica's previous heuristics. A comparison of Eld+R to Eld+B shows that all but one benchmarks solved by old configurations of Eldarica can also be solved by Eld+R. The one benchmark not solved by Eld+R requires a predicate relating three variables in an equality, which according to our experience does not fall into frequently used patterns. Moreover, as Fig. 4 shows, the configuration Eld+BR, which combines roles and old heuristics of Eldarica, solves 3% less tasks than Eld+R. One possible reason for the slowdown (and consequently the lower number of solved benchmarks) of Eld+BR are redundant predicates generated by built-in heuristics of Eldarica. *These results confirm that our framework not only describes new heuristics but also captures all previous heuristics of Eldarica.*

Improvement on unsafe benchmarks. Surprisingly, the initial predicates also help to solve more unsafe benchmarks, as Fig. 4c shows. In principle, these predicates can be found by Eld+B with a higher value of runtime, as demonstrated by the configuration Eld+BH. *We conclude that when variable roles are used, the number of solved unsafe tasks does not decrease in general and even increases for SV-COMP CFI benchmarks.*

Comparison of Eldarica to SMT solvers. We compare Eldarica to SMT solvers Z3 and Spacer[8]. We note that a small number of tasks in benchmarks SV-COMP Loops and HOLA cannot be processed by Z3 and Spacer

[8] We evaluate the default configuration of Z3 without command-line options. To execute Spacer, we use the command-line option `fixedpoint.xform.slice=false`.

because of existential quantifiers in the SMT translation, which is not in the fragment handled by the PDR engine of Z3. We denote these benchmarks as "Not Supported" in Fig. 4. We observe that, on one hand, all configurations of Eldarica outperform both Z3 and Spacer in the number of solved tasks, in particalar Eld+R solves 30% more tasks than Z3. We note, however, that our method for guiding predicate abstraction uses the structure of a program, which is not preserved on the level of SMT formulae. On the other hand, the mean runtime of Z3 is 2.0 times lower than the mean runtime of Eld+R. *To conclude, Eldarica outperforms Z3 and Spacer in the number of solved tasks, but loses in speed.*

Comparison of Eldarica to CPAChecker. Finally, we compare Eldarica to the model checker CPAchecker. We observe that on safe and unsafe tasks the tools show complementary strengths. In particular, CPAchecker proves more tasks unsafe than Eldarica on CFI benchmarks, and on other benchmark sets shows comparable to Eldarica results. For safe benchmarks, however, on all benchmark sets CPAchecker can prove fewer programs safe than the Eldarica configurations Eld+B, Eld+R and Eld+BR. *To conclude,* Eldarica *with interpolation abstraction outperforms* CPAchecker *on safe benchmarks, while* CPAchecker *performs better on a family of unsafe benchmarks.*

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers, Principles, Techniques. Addison wesley (1986)
2. Apel, S., Beyer, D., Friedberger, K., Raimondi, F., von Rhein, A.: Domain types: Abstract-domain selection based on variable usage. In: Haifa Verification Conference. vol. 8244, pp. 262–278. Springer (2013)
3. Beyer, D.: Reliable and reproducible competition results with benchexec and witnesses (report on sv-comp 2016). In: Tools and Algorithms for the Construction and Analysis of Systems (TACAS). vol. 9636, pp. 887–904. Springer (2016)
4. Beyer, D., Löwe, S., Wendler, P.: Refinement selection. In: Model Checking Software, vol. 9232, pp. 20–38. Springer (2015)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
6. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems. vol. 4963, pp. 337–340. Springer (2008)
7. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. In: Computer Aided Verification (CAV). vol. 9206, pp. 561–579. Springer (2015)
8. Demyanova, Y., Pani, T., Veith, H., Zuleger, F.: Empirical software metrics for benchmarking of verification tools. Int. J. Form. Methods Syst. Des. pp. 1–28 (2017)
9. Demyanova, Y., Veith, H., Zuleger, F.: On the concept of variable roles and its use in software analysis. In: Formal Methods in Computer-Aided Design (FMCAD). pp. 226–230. IEEE (2013)
10. Dillig, I., Dillig, T., Li, B., McMillan, K.: Inductive invariant generation via abductive inference. In: ACM SIGPLAN Notices. vol. 48, pp. 443–456. ACM (2013)

11. Engler, D., Chen, D.Y., Hallem, S., Chou, A., Chelf, B.: Bugs as deviant behavior: A general approach to inferring errors in systems code. In: Operating systems principles (SOSP). vol. 35. ACM (2001)
12. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: Automated software engineering (ASE). pp. 349–360. ACM (2014)
13. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Computer Aided Verification (CAV). vol. 1254, pp. 72–83. Springer (1997)
14. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Programming Language Design and Implementation (PLDI). pp. 405–416. ACM (2012)
15. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Theory and Applications of Satisfiability Testing (SAT). vol. 7317, pp. 157–171. Springer (2012)
16. Jhala, R., Majumdar, R.: Software model checking. ACM Computing Surveys (CSUR) 41(4), 21 (2009)
17. Komuravelli, A., Gurfinkel, A., Chaki, S., Clarke, E.M.: Automatic abstraction in smt-based unbounded software model checking. In: Computer Aided Verification (CAV). vol. 8044, pp. 846–862. Springer (2013)
18. Leroux, J., Rümmer, P., Subotić, P.: Guiding craig interpolation with domain-specific abstractions. Acta Informatica 53, 1–38 (2016)
19. Nori, A.V., Rajamani, S.K.: An empirical study of optimizations in YOGI. In: Software Engineering (ICSE). vol. 1, pp. 355–364. ACM (2010)
20. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for Horn-clause verification. In: Computer Aided Verification. vol. 8044, pp. 347–363. Springer (2013)
21. Sajaniemi, J.: An empirical analysis of roles of variables in novice-level procedural programs. In: Human-Centric Computing Languages and Environments (HCC). pp. 37–39. IEEE (2002)
22. Van Deursen, A., Moonen, L.: Type inference for cobol systems. In: Reverse Engineering (RE). pp. 220–230. IEEE (1998)