Die approbierte Originalversion dieser Dissertation ist in der Hauptbibliothek der Technischen Universität Wien aufgestellt und zugänglich.
http://www.ub.tuwien.ac.at

TU UB WIEN Universitätsbibliothek

The approved original version of this thesis is available at the main library of the Vienna University of Technology.
http://www.ub.tuwien.ac.at/eng

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Systematic Study of Variable Roles and their Use in Software Verification

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Yulia Demyanova, Msc.
Registration Number 01029734

to the Faculty of Informatics
at the TU Wien
Advisor: Ass.Prof. Dipl.-Math. Dr. Florian Zuleger

The dissertation has been reviewed by:

Assoc.Prof. Dr. Philipp Rümmer	Ass.Prof. Dipl.-Ing. Dr. Zvonimir Rakamarić

Vienna, 23rd January, 2018

Yulia Demyanova

DEDICATION

This dissertation is dedicated to the late Prof. Helmut Veith
who supervised me from 2011 to 2016
and who inspired most of the ideas presented here.

Erklärung zur Verfassung der Arbeit

Yulia Demyanova, Msc.
Barichgasse 21/13, 1030 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 23. Jänner 2018

Yulia Demyanova

Acknowledgements

I would like to express my gratitude to a number of people with whose support and encouragement this dissertation was written. First of all, I am very grateful to Prof. Helmut Veith who supervised me for 5 years until he tragically passed away in 2016. Helmut allowed me to join his group and shared with me his idea of using variable roles in software verification. Helmut was always full of ideas, in many circumstances he was extremely helpful and attentive to all people he came across.

It is hard to overestimate the help of Ass.Prof. Florian Zuleger, who co-supervised me from the first years of my work and later on took over supervision for my thesis. Florian patiently went through many iterations of reviewing our papers and this thesis; thanks to his deep mathematical insights the formalisation of variable roles became much more concise and clear. From Florian I tried to learn to precisely express my ideas and to always give examples.

Finishing this work would be hardly possible without the help of Assoc.Prof. Georg Weissenbacher, who supported me in different matters and patiently allowed me to continue using my office after then end of my contract up to the last day of writing the thesis. I am also very grateful to Prof. Thomas Eiter and Florian Zuleger for dealing with bureaucratic hurdles during the last months before submitting the thesis.

I would also like to thank Igor Konnov, Ass.Prof. Zvonimir Rakamarić, Assoc.Prof. Philipp Rümmer and Thomas Pani for their invaluable comments on the thesis. I thank Philipp and Thomas for joint work, of which I have good memories, and for their readiness to help at any time. Especially I thank Philipp for his help with finishing our paper and parts of this thesis in spite of acute shortage of time and Thomas for translating the abstract of the thesis into German. I am also grateful to Prof. Laura Kovács, Jakob Zwirchmayr and Prof. Helmut Seidl for useful discussions about variable roles, and to Andreas Holzer for his help at the early stage of my PhD studies.

I thank Moritz Sinn for carrying out seminars together and for providing his thesis as a guideline. I was happy to have Mitra Tabaei, Ivan Radiček and Annu Gmeiner as room mates, and I especially thank Mitra for being a good friend and for our interesting discussions during lunch time.

I thank all the members of the FORSYTE group, especially Toni Pisjak for patiently solving all my technical problems and Eva Nedoma for oftentimes preparing the official

papers for me to travel to the U.K. to my husband.

Finally, I thank all my Russian friends and the clergy of St. Nicholas Russian Orthodox Cathedral in Vienna, especially Fr.Vladimir, for filling my everyday life in Vienna with joy. I am very grateful to my husband for his unconditional love and support, especially during difficult weeks before deadlines. I express my deep gratitude to my parents who put all their love into me and patiently waited until I finished this dissertation.

Kurzfassung

Eine der größten Herausforderungen in der Softwareverifikation ist die Auswahl geeigneter Abstraktionen – also vereinfachter Programmmodelle, die zumindest jedes Verhaltens des Originalprogramms erlauben. Die meisten Softwareverifikationstools sind aufgrund der Unentscheidbarkeit des Verifikationsproblems darauf angewiesen, eine *brauchbare* Abstraktion zu finden, die einerseits ein effizient analysierbares, gleichzeitig aber dennoch genügend prezises Programmmodell liefert. Die Wahl einer solchen brauchbaren Abstraktion verlangt jedoch nicht-triviales Verständnis des Originalprogramms und wird daher entweder durch menschliche Unterstützung oder durch Heuristiken erreicht. Dadurch liefert das Verifikationstool nur noch auf einer eingeschränkten Menge von Originalprogrammen, der *Anwendungsdomäne*, erfolgreich Ergebnisse. Beispiele für Anwendungsdomänen sind Gerätetreiber, nebenläufige Programme, Integer-Programme, usw. Durch die Einschränkung von Verifikationstools auf bestimmte Anwendungsdomänen entsteht eine weitere Herausforderung: die Auswahl eines optimalen Verifikationstools für ein gegebenes Verifikationsproblem.

Der Unterschied zwischen einem Verifikationstool und menschlichem Programmverständnis liegt darin, dass moderne Softwareverifikationstools Programme nicht als Artifakte menschlicher Ingenieursarbeit betrachten, sondern sie blindlings in logische Formeln übersetzen. In dieser Dissertation formalisieren und untersuchen wir das Konzept der *Variablenrollen*, welches implizites Wissen über typische Muster der Variablenverwendung in Programmen erfasst. Dieses Wissen wird von erfahrenen EntwicklerInnen sowohl zum Schreiben als auch zum Verstehen von Programmen eingesetzt. Beispiele für solche Muster sind Bitvektoren, Zählvariablen, Schleifeniteratoren, usw.

Wir stellen die Hypothese auf, dass mittels Kenntnissen über die Verwendung der Variablenrollen im betrachteten Programm die zwei oben genannten Herausforderungen – also die automatische Auswahl von brauchbaren Programmabstraktionen und die automatische Auswahl einens optimalen Verifikationstools für ein gegebenes Problem – zu lösen sind. Dazu formalisieren wir den Begriff der Variablenrolle: Wir erstellen eine Klassifikation der häufigsten Variablenrollen in anwendungsorientierten Open-Source-Programmen und definieren ein Framework zur formalen Spezifikation derselben. Als Spezifikationsmechanismus kommt dabei Datalog zum Einsatz.

Weiters untersuchen wir die Anwendung von Variablenrollen in der Softwareverifikation anhand zweier Szenarien: Erstens erstellen wir ein Portfolio von Softwareverifikations-

tools, das zur Auswahl eines Verifikationstools für ein gegebenes Verifikationsproblem Programmmetriken verwendet, welche auf Variablenrollen basieren. Wir konstruieren das Portfolio mithilfe eines Algorithmus zum maschinellen Lernen, und evaluieren unsere Implementierung anhand der "Competition on Software VerificationSSV-COMP als Fallstudie.

Zweitens verwenden wir Variablenrollen zur automatischen Generierung von Heuristiken zur Programmabstraktion: Wir definieren Heuristiken für den Model-Checker Eldarica, um Predikate für die initiale Abstraktion zu erstellen und die Abstraktionsverfeinerung durch Templates für Craig-Interpolation zu lenken. Unter Verwendung der Variablenrollen ersetzen wir die existierenden, in Eldarica eingebauten Heuristiken und definieren einige neue Heuristiken. Wir evaluieren diesen Ansatz auf Verifikationsbeispielen aus dem SV-COMP-Wettbewerb und der Literatur.

Abstract

A major challenge in software verification is choosing an abstraction — a simplified program model which comprises a superset of behaviours of the original program. Since software verification is undecidable, the efficiency of most software verification tools is determined by choosing a *suitable* abstraction which yields a tractable yet still precise enough program model. However, picking a suitable abstraction requires non-trivial insights and is either done with human intervention, or implemented using heuristics, tighting a verification tool to a restricted set of programs, called *application domain*. Examples of application domains are device drivers, concurrent programs, integer programs, etc. The limitedness of application domains of verification tools causes another verification challenge — an optimal choice of a verification tool for a given task.

The difference between a verification tool and a human reading a program is that most modern software verification tools do not treat programs as human-engineered entities and blindly translate a program to a logical formula. In this dissertation, we formalise and study the concept of a *variable role*, which captures the implicit knowledge about typical patterns of variable use in programs. This knowledge is employed by experienced programmers to write and understand programs. Examples of these patterns are bitvectors, counters, loop iterators, and so on.

We conjecture that the two challenges mentioned above, i.e. automatically choosing a suitable program abstraction and optimally choosing a verification tool for a given problem, can be solved with the knowledge of which variable roles are used in a program. To this end, we formalise the notion of a variable role: we create a classification of most frequent variable roles in practical open source programs and define a framework for a formal specification of variable roles. As a specification formalism for variable roles we use Datalog.

We explore the application of variable roles in software verification in two settings. First, we create a portfolio solver which chooses a software verification tool for a given verification task, using program metrics based on variable roles. We construct our portfolio solver using a machine learning algorithm. As a case study, we evaluate the implementation of our algorithm in the setting of the software verification competition SV-COMP.

Second, we use variable roles to define heuristics for the automatic generation of program abstraction. In particular, we define heuristics for the model checker Eldarica to

generate predicates for initial abstraction and to guide abstraction refinement through templates provided for Craig interpolation. Using variable roles, we re-implement the existing built-in heuristics of Eldarica and define several new heuristics. We evaluate our approach on a subset of benchmarks from the competition SV-COMP and verification benchmarks from the literature.

Contents

CHAPTER 1

Introduction

Software verification aims at systematically checking the conformance of a program to a specification. It uses methods of mathematical logic to construct a mathematical proof about a program. To this end, the techniques of software verification create a mathematical model of the program. Specifically, in order to reason about variable values, the techniques of software verification analyse (or *enumerate*) program states, where a *state* is an assignment of values to program variables. The set of all possible program states is called the *state space* of a program. Programs written using modern programming languages typically have infinite or intractably large state space. Therefore, in order to reason about a program, the techniques of software verification create a finite simplified model of a program under analysis, which captures the crucial properties of the program.

Let us consider the following situation. Software engineer John wrote a Linux device driver and wants to verify the driver. John knows that there exists more than a dozen state-of-the art verification tools, and he needs to choose the one which gives the correct answer for his task in shortest time. John also knows that according to Rice's theorem, any program property is undecidable, i.e. there is no algorithm which can solve every verification problem. This means that whichever tool he chooses, there are tasks which the tool cannot solve. To devise tools which efficiently solve the verification tasks arising in practice, tool developers optimise their tools to a restricted set of tasks, called the *problem domain* of a tool. With this in mind, John searches for a tool optimised to verify device drivers.

To make an informed choice, John studies the results of the most recent software verification competition SV-COMP'17. Luckily, the competition includes a dedicated category of benchmarks called "DeviceDriversLinux64 Safety", which consists of Linux device drivers and requires the analysis of pointer aliases and function pointers. To pick the best tool, John looks at the competition results and chooses four tools which get the highest score in the category "DeviceDriversLinux64 Safety". Specifically, these are the

tools Ultimate Taipan, CPAchecker, Ultimate Automizer and BLAST. However, when inspecting the results of the four tools for individual benchmarks in the category, John finds out that the tools exhibit unsoundness (i.e. report safe when the benchmark is unsafe), bugs (i.e. crash) and incompleteness (i.e. run out of memory or do not meet the time limit). The category "DeviceDriversLinux64 Safety" contains almost 2800 tasks with the average size of 14.8 KLOC, and John gives up trying to systematically analyse the results and the respective tasks to make an optimal choice. Since it is only one driver that he needs to verify, he decides to play safe and run all four tools to compare the results.

Having solved one problem, John confronts another one. He finds out that the four tools come with a set of configuration parameters. From the "Formal Methods" course at the university John knows that in order to reduce an infinite or intractably large state space of a program, tools create a simplified program model called *abstraction*. The abstraction tracks all behaviours of the original program, and if a property holds for program abstraction, it also holds for the program itself. In order to find a *suitable* abstraction, which is accurate enough (i.e. not too coarse) but still tractable (i.e. not too fine-grained), tool developers implement numerous strategies. It is these strategies that are configured with parameters. Now John is facing a problem of choosing the right parameters, and he solves the problem by trial and error in several iterations.

To conclude, this examples illustrates that at different stages of software verification empirical choices need to be done. First, in order to verify a program, a software engineer has to pick a suitable tool optimised for a given application domain. Second, tools use strategies to find a suitable abstraction, and the strategies are often configured with user-defined parameters. Therefore, we suggest in this thesis a *systematic* way to make these empirical choices in software verification.

We organise the rest of this chapter as follows. In Section 1.1 we discuss the two empirical choices in more detail. In Section 1.2 we explain that to make these choices, the implicit information of the program source code can be used. We informally introduce our method based on the implicit information, namely the patterns of variable use, in Section 1.3. We discuss other possible application domains for our method in Section 1.4. We state the aims of the thesis and describe our methodological approach in Section 1.5. We give an overview of the structure of the thesis and a short summary of each chapter in Section 1.6. Finally, we list the contributions of the thesis in Section 1.7 and discuss our publications on the topic of the thesis in Section 1.8.

1.1 Empirical Choices in Software Verification

In this section, we describe the two empirical choices which need to be done for software verification. Specifically, we first illustrate the problem of choosing a verification tool on example of the software verification competition SV-COMP. Then, we discuss the problem of choosing a suitable abstraction.

1.1.1 Problem 1: Choosing a Verification Tool

In the last two decades, there has been a remarkable practical advance in automated program verification using tools such as SLAM/SDV [BR02], BLAST [HJMS03], CBMC [CKL04], SATABS [CKSY05], ASTRÉE [CCF+05], SLAYER [BCC+07] and many others. The success and gradual improvement of these tools is a multidisciplinary effort — modern software verifiers combine methods from a variety of overlapping fields of research including model checking [JM09], static analysis [FO76], shape analysis [NNH99], SAT solving [BHvM09], SMT solving [KS08, BM07], abstract interpretation [CC77], termination analysis [CPR11], pointer analysis [Hin01] etc.

The mentioned techniques all have their individual strengths, and a modern software verification tool needs to pick and choose how to combine them into a strong, stable and versatile tool. The trade-offs are based on both technical and pragmatic aspects: many tools are either optimised for specific application domains (e.g. device drivers), or towards the in-depth development of a technique for a restricted program model (e.g. termination for integer programs[1]). Recent projects like CPA [BHT07] and FrankenBit [GB14] have explicitly chosen an eclectic approach which enables them to combine different methods more easily.

Example 1.1.1. For example, consider the annual *International Competition on Software Verification* (SV-COMP[2], since 2012) [Bey14, Bey15, Bey16]. SV-COMP is the most ambitious attempt to create a common setup for comparing software verification tools and, as of 2016, is based on more than 6,600 C source files. The files are manually partitioned into categories, by characteristic features such as usage of bitvectors, concurrent programs, Linux device drivers, etc. The scores of the competition SV-COMP'16, are shown in Table 1.1. The negative scores correspond to wrong answers, which are heavily penalised according to the scoring policy of the competition. The competition scores in Table 1.1 demonstrate that there is no tool which performs well in all categories. Specifically, 20 out of 35 participating tools are among the three best tools in at least one category, but in at least one category each of the 20 tools either has low scores or does not participate.

For example, the tool CPA-SEQ [DLW15] is among three best tools in the categories which include bit operations, heap data structures, integer variables and control-flow, and pointers and aliases, i.e. the categories BitVectors, Heap, IntegersControlFlow and DeviceDriversLinux64. On top of that, CPA-SEQ is the second best tool in the category Overall, which comprises the tasks of all competition categories. However, CPA-SEQ does not implement check for termination and has poor support for arrays, see the categories Termination and Arrays.

Similarly, the tool UAUTOMIZER [HDG+16] is among four best tools in 7 out of 9 categories and the winner of the category Overall, but has poor support for floating point

[1] An integer program is a pointer- and structure-free abstraction of a program. The transformation obtains a mathematical optimisation problem where program integer properties of interest are translated to constraint variables.

[2] `https://sv-comp.sosy-lab.org`. Accessed 23 January 2018.

Verifier	Arrays 316 points max. 183 tasks	BitVectors 92 points max. 60 tasks	Heap 382 points max. 239 tasks	Floats 140 points max. 81 tasks	IntegersControlFlow 3629 points max. 2331 tasks	Termination 1129 points max. 631 tasks	Concurrency 1240 points max. 1016 tasks	DeviceDriversLinux64 3977 points max. 2120 tasks	FalsificationOverall 2371 points max. 6030 tasks	Overall 10855 points max. 6661 tasks
2LS				**136**	1196				-2438	-38205
APROVE						**909**				
BLAST					-1653			**2704**		
CASCADE			**197**							
CBMC	62	46	8	**134**	-1239		882	1972	391	3386
CEAGLE				**136**						
CIVL							**1240**			
CPA-KIND	3	**77**	161	76	**2095**	0	0	2350	**707**	4094
CPA-REFSEL				35	1539	0	0	**3177**	36	2157
CPA-SEQ	-61	**87**	**234**	75	**2652**	0	282	**2801**	496	**4794**
ESBMC	**190**	**84**	163	-15	1217	0	742	1688	248	4145
LAZY-CSEQ							**1240**			
MU-CSEQ							**1240**			
PREDATORHP			**298**							
SEAHORN	-301	-131	-257	0	1572	**504**	-24659	1694	-4333	-22393
SMACK+CORRAL	**146**	44	155	0	**2013**	0	999	2206	**800**	**4223**
SYMBIOTIC	**101**	-2	105	-18	633	0	0	980	-370	1223
UAUTOMIZER	83	69	169	2	1865	**895**		2686	**823**	**4843**

Table 1.1 Results of SV-COMP'16. A number in i-th row and j-th column correspond to the score of i-th tool in j-th SV-COMP category. Empty cells correspond to the cases when a tool did not participate in a category. Dark grey, light grey and white+bold text respectively indicate the first, second and third highest places in each category. When several tools have equal scores in one category, tools with lower runtime in the category are ranked higher, according to the placing policy of SV-COMP. We omit the runtimes for the sake of clarity. We show only the tools which are among three best tools in at least one category.

operations and no support for concurrency, see the categories Floats and Concurrency.

The approach of optimising tools to a narrow domain is taken to the extreme by the winner of the category Floats the tool 2LS [SK16], which has positive score only in one more category besides Floats. Similarly, three best tools in the category Concurrency MU-CSeq [TNI+16], Lazy-CSeq [ITF+14] and CIVL [ZEL+16] do not participate in any other category than Concurrency. ▲

To summarise, numerous verification tools are implemented, and the tools have different strengths and weaknesses. To choose a verification tool for a program, a software engineer needs expertise in the state-of-the-art verification tools and understanding which challenges the program poses for software verification.

We therefore formulate the following challenge which we solve in this thesis.

C1 **– Challenge "Automatic choice of a verification tool"**. A method is needed which automatically chooses the best suited software verification tool for a given verification task. For choosing a tool, besides the application domain of a tool, different criteria can be taken into account, such as soundness, runtime and memory consumption of a tool, and so on.

1.1.2 Problem 2: Choosing Abstraction

Finding suitable abstractions is crucial for the efficiency of software verification. In practice, to choose a suitable abstraction verification tools implement numerous *heuristics*. Specifically, with heuristics we understand stategies, which tool developers implement so that the tool more efficiently solves a restricted class of problems.

In particular, software verification tools use heuristics to chose logical formulas to create an abstract model of a program, i.e. *predicates* and elements of *abstract domains*. In addition, after creating an initial model, heuristics are used to iteratively *refine* the model, i.e. obtain a more fine-grained model, in case the model is not precise enough to prove the verification property of interest. We formally introduce the notions of predicates and abstract domains, and the methods of automatic abstraction refinement in Chapter 2.

We give examples of heuristics in software verification tools in Sections 2.1.4 and 2.2.5. Here we only mention that heuristics are typically domain-specific, i.e. tie verification tools to a specific application area.

In addition, as argued in [NR10], heuristics are usually less explicitly documented than the core algorithms used in verification tools.

Example 1.1.2. In the year 2017, in the software verification competition SV-COMP'17 the tool ESBMC participated in four different configurations, namely ESBMC, ESBMC-FALSI, ESBMC-INCR and ESBMC-KIND. In the competition report, the organisers list the participating tools and for each tool give a link to a paper describing the tool. However, for the configurations ESBMC, ESBMC-FALSI and ESBMC-INCR the same link is provided, and the respective paper gives no details on the differences between the configurations. In the same time, the four configurations show different results in the competition. For example, in the category FalsificationOverall, which contains only unsafe tasks, the configurations ESBMC-INCR and ESBMC-KIND take second and third place respectively, the configuration ESBMC-FALSI does not take any place and the score of the configuration ESBMC is an order of magnitude lower than the score of the three other configurations. Therefore, the results show that the heuristics implemented in the different configurations have a serious impact on tool performance. ▲

Furthermore, heuristics are often manually configured by users of the verification tools.

Example 1.1.3. As of 2009 the static analyser ASTRÉE contained at least 12 abstract domains and 150 configuration options in order to specify which abstract domains to use and the parameters of the abstract domains [CCF+09]. ▲

Making the right choices to configure the heuristics of a verification tool needs understanding of the internal functioning of the tool and is specific for a program under analysis. The experimental results of rigorous evaluation of the heuristics are rarely reported with a few exceptions [NR10, SMM11, TKK+14].

To summarise, to search for abstraction, verification tools implement numerous heuristics. Heuristics have a big impact on the efficiency of tools. However, heuristics are not systematic, since heuristics are tight to specific application domains and are poorly documented. Furthermore, heuristics are not automatic, since some tools which implement heuristics require human guidance and expertise to be properly configured.

We therefore formulate the following challenge which we solve in this thesis.

***C2* – Challenge "Automatic choice of program-specific abstraction"**. A method is needed which allows to *automatically* generate a suitable abstraction for a program under analysis using *formally* specified heuristics.

1.2 Implicit Structure of the Source Code

Programs are human-engineered entities, and they have an *implicit structure* which makes them easier to comprehend by a human. In particular, programs use specific code patterns, are structured in procedures, have meaningful identifiers, are explained with comments and documentation, etc. However, for majority of software verification tools there is no difference between a human-written program and machine-generated or obfuscated code. A software verification tool typically translates a program to a simplified model thus losing the implicit structure. To solve the challenges mentioned in the previous section, we study in this thesis the artefacts of the implicit structure. Specifically, we study patterns of variable use, which we introduce in the next section.

1.3 Patterns of Variable Use

As argued above, there is a gap between what can be written in a programming language and the way programmers write their code. Programs written in commonly used imperative programming languages, such as C, Java, Perl, Python, share typical patterns of variable use, like flags, loop iterators, counters, array indices, bitvectors, and so on. However, the type systems in these languages do not capture this information: variables used in different patterns would typically be assigned same type, e.g. `int` in C. Experienced programmers have informal knowledge of the patterns, and recognising the patterns in a program helps them to understand the program. We call such patterns *variable roles*. We illustrate variable roles with the following examples.

Example 1.3.1. Consider the C program in Fig. 1.1a, which computes the number of non-zero bits of variable `x`. In every loop iteration, a non-zero bit of variable `x` is set to

```
1 // n: counter
2 // x, x_old: bitvector
3 extern int x;
4 int n = 0;
5 int x_old = x;
6
7 while (x) {
8   n++;
9   x = x&(x-1);
10 }
```

(a) roles *bitvector* and *counter*

```
1 //fd: file descriptor
2 int fd = open(path, flags);
3
4 // c: character
5 int c, val=0;
6
7 while (read(fd, &c, 1)>0 &&
8   isdigit(c)) {
9     val = 10*val + c-'0';
10 }
```

(b) roles *character* and *file descriptor*

Figure 1.1 Examples of usage patterns of integer variables in C programs.

zero and variable `n` is incremented. From statements `n=0` and `n++`, a programmer deduces that `n` is a *counter*. Similarly, from expression `x&(x-1)`, which contains bitwise-and operation, a programmer deduces that variable `x` is a *bitvector*. ▲

Example 1.3.2. Consider the code in Fig. 1.1b, which reads a decimal number from a text file and stores its numeric representation in variable `val`. Since the function `open` from the C standard library opens a file and returns its descriptor, a programmer deduces from statement `fd=open(path,flags)` that variable `fd` is a *file descriptor*. Similarly, since `isdigit()` is a character classification routine which checks whether its parameter is a decimal digit, a programmer deduces from statement `isdigit(c)` that `c` is a *character*. ▲

1.4 Application Domains of Variable Roles

We now motivate why variable roles are interesting to study and how the roles can be used to solve the challenges stated in Section 1.1, specifically *C1 (Automatic choice of a verification tool)* and *C2 (Automatic choice of program-specific abstraction)*.

In Section 1.4.1 we describe the applications of roles in Software Verification which we have investigated in this thesis and make connections to the respective sections of the thesis. In particular, we discuss how we use variable roles to

i. define metrics for software verification benchmarks;

ii. build a portfolio solver to automatically choose a software verification tool;

iii. formally specify heuristics in software verification to automatically generate program-specific abstraction.

We discuss further possible applications of variable roles in Section 1.4.2.

1.4.1 Applications of Roles Studied in this Thesis

Metrics for Software Verification Benchmarks

In Chapter 4, we use variable roles to compute code metrics to compare software verification benchmarks. Comparing benchmarks is an acute problem in software verification. First, there exists no common set of benchmarks in the software verification community. Therefore, benchmarks are often manually selected, handcrafted, or chosen a posteriori to support a certain technical insight. Second, oftentimes neither the tools nor the benchmarks are available to other researchers. For example, Windows device drivers from Microsoft SDV toolkit [TKK+14] are available only in part and in the form of Boolean programs, rather than the original C code.[3] Third, some sets of benchmarks are very large, as for example the benchmarks of the competition SV-COMP, which makes it difficult to analyse the source code of the benchmarks manually.

We suggest that the three problems formulated above can be solved by devising variable role based metrics over benchmarks. The metrics will give the following benefits:

1. understanding the presence of language constructs in benchmarks;
2. characterising benchmarks not publicly available and
3. understanding large benchmarks without manual inspection.

Portfolio Solvers for Software Verification

In Chapter 4, we also use variable-role-based program metrics to build a portfolio solver for software verification tools and to explain the results of software verification competitions. Here a portfolio solver is a software verification tool which uses heuristic preprocessing to select one of the existing tools [HLH97, GS01, Ric76]. For the software verification community, portfolio solving brings interesting advantages:

1. A portfolio solver *optimally uses available resources.* While in theory one may run all available tools in parallel, in practice the cost of setup and computational power makes this approach infeasible. A portfolio predicts the n tools it deems best-suited for the task at hand, allowing better resource allocation.
2. It can *avoid incorrect results of partially unsound tools.* Practically every existing software verification tool is partially incomplete or unsound. A portfolio can recognise cases in which a tool is prone to give an incorrect answer, and suggest another tool instead.
3. Portfolio solving allows us to *select between multiple versions of the same tool.* A portfolio is not only useful in deciding between multiple independent tools, but also between the same tool with different runtime parameters (e.g. command-line arguments).

[3] `https://www.microsoft.com/en-us/download/details.aspx?id=52338`. Accessed 23 January 2018.

```
1 extern int nondet_int();
2 int main() {
3   int n = nondet_int();
4
5   // k,i,j : local counter
6   int k, i, j;
7
8   for (k=0,i=0; i<n; i++,k++);
9   for (j=n; j>0; j--,k--) {
10      assert(k > 0);
11  }
12
13  return 0;
14 }
```

Figure 1.2 Code example for role-based heuristics in software verification: role *local counter*.

4. The portfolio solver *gives insight into the state-of-the-art* in software verification. As argued in [XHHL12] for SAT solving and in Example 1.1.1 for software verification, the state-of-the-art can be set by a combination of available solvers, rather than the *single best solver* (e.g. a competition winner). This accounts for the fact that different techniques have individual strengths and are often complementary.

Systematic Specification of Heuristics in Software Verification

In Chapter 3 we use variable roles to build a framework for the formal specification of heuristics. In particular, we specify in the framework heuristics based on variable roles independently from the implementation in the form of configuration files. In addition, in Chapter 5, we use variable roles to generate program-specific abstraction from the code under analysis. We illustrate this idea with the following example.

Example 1.4.1. The code in Fig. 1.2 increments variables `i` and `k` in the loop at line 8 until `i` reaches `n`, and in the loop at lines 9–11 variables `j` and `k` are decremented until `j` reaches `0`. The assertion checks that the value of variable `k` remains positive in the loop. The assertion can be proven using the predicates `k>=i` and `k>=j`. However, these predicates are difficult to find, e.g., the baseline version of model checker Eldarica [RHK13] 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 this kind of verification problems. For example, recall from Example 1.4.1 in Section 1.1.2 that the model checker Eldarica uses Craig interpolation to generate predicates. To find suitable predicates, Eldarica restricts the shape of predicates with user-specified templates. The template x_1-x_2 passed to Eldarica guides the interpolation solver used by Eldarica to the predicates of the form $x_1-x_2>=n$, $n \in \mathbb{N}$. Using the templates `i-k` and `k-j`, the most recent

version of Eldarica [LRS16] finds the predicates `i-k≤0` and `k-j≥0` respectively and proves the program safe in 5 seconds and 6 CEGAR iterations.

With the goal of systematising and extending the previous heuristics of Eldarica, we propose a heuristic which tracks the dependencies between loop counters as follows: The heuristic searches for variables `x` assigned in a loop in a statement which matches 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` guiding the interpolation solver to predicates of the form `x1-x2>=n, n`$\in \mathbb{N}$.

We formalise the heuristic using the variable role *local counter* which we informally define as follows. Variable `x` is a *local counter* of loop L if `x` is assigned in a statement `x=x+expr`, where `expr` is an arbitrary expression, and the statement `x=x+expr` is nested in a body of the loop L. We give a formal definition of the role *local counter* in Section 3.3.3 and the formal definition of the above heuristic based on this role in Section 5.2. Our algorithm uses the role to infer the template as follows. For every pair of local counters `x1` and `x2` of same loop L, generate the template `x1-x2`. ▲

1.4.2 Further Applications of Roles

Pre-Analysis in Multi-Stage Verification

Variable roles can be used to choose problem-specific parameters for model checkers, or other analysis tools.

Example 1.4.2. In Section 2.2.5 we describe the packing heuristic implemented in the tool Astrée for octagon abstract domain. To reduce the number of relations, the heuristic splits the variables into packs and relates only the variables from same pack. In particular, the heuristic puts into one pack the variables incremented or incremented within the same loop. For example, for the code

```
x=10; for (i=0;i<=10;i++) x++;
```

the heuristic allows to find an invariant `x-i=10`.[4] To formalise this heuristic, we use the role local counter introduced in Example 1.4.1 as follows. Given variables `x` and `y` which have the role *loop counters* in same loop L, configure the octagon abstract domain to generate a dependency for the two variables. ▲

Program Understanding

Variable roles can be used in program understanding, e.g. in a program visualisation tool which visualises program variables with images corresponding to their roles [SK04].

In addition, since roles capture typical patterns of variable usage, roles can be used in teaching programming languages [SK05]. In the experiment of [SK05] three groups of

[4] We take this example from [CCF+06].

students were taught Pascal programming language in different conditions. In particular, for the first group the concept of variable roles was used in the process of teaching, for the second group role-based program visualisation was used in addition, and for the third group the concept of variable roles was not used. The comparison of grades along with a number of measures assessing program comprehension show that the group which used the role-based program visualisation tool demonstrated the deepest knowledge of the programming language.

In addition, roles can be used for algorithm recognition, e.g. using decision trees based on the number of variables having specific roles and other structural code metrics [TKM11]. The role-based method of algorithm recognition [Tah10] was also used for automatic assessment of programming assignments, specifically the implementations of sorting algorithms [TMK08].

Bug Finding

A change of a variable role between consecutive code fragments can be used as an indicator of a possible bug. The bug-finding tool Coverity [HCXE02] uses a similar idea. In particular, Coverity infers a hypothetical expected order of statements, e.g. the function call `spin_lock()` should be followed by the call to `spin_unlock()`. Then, Coverity detects the cases when the order of statements is not respected and classifies these cases as possible bugs. We discuss this work in more detail in Chapter 6.

1.5 Aim of the Work and Methodological Approach

In this thesis we aim to formalise and study the notion of variables roles. In particular, we are interested in the use of variable roles for program verification, because we believe that variable roles carry information which can be used for solving the challenges of Section 1.1, namely *C1 (Automatic choice of a verification tool)* and *C2 (Automatic choice of program-specific abstraction)*.

Tasks. We split the challenges into the following tasks:

T1 – *Role specification framework.* We develop a framework for the specification of variable roles. Each specification in the framework is effective, i.e. a specification allows to automatically infer which variables have the specified role.

We explore the use of variable roles in software verification (SV) :

T2 – *Portfolio solver for Software Verification.* First, we build a portfolio solver for software verification to automatically choose a verification tool for a program using variable roles identified in the program. As a case study, we build a portfolio solver for the competition SV-COMP'14.

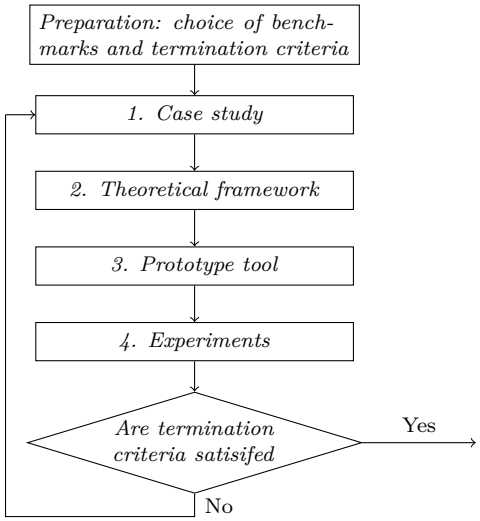


Figure 1.3 Research cycle undertaken in the thesis.

T3 – Systematic heuristics for Software Verification. Second, we devise role-based heuristics for choosing predicates in software model checking. As a case study, we choose the model checking tool Eldarica. We reformulate the existing heuristics of the tool, and we adapt the tool to different application domains using new heuristics.

Research cycle. In order to solve the three tasks, we develop a theoretical framework and an implementation of our ideas, going through the steps of a research cycle illustrated in Fig. 1.3. Before entering the loop, we choose a set of benchmarks for the class of tasks we are interested in and define the criteria which the resulting framework and implementation should satisfy. We then take the following steps in a loop. First, we do a case study on a small subset of the benchmarks. Second, we use our findings to devise a theoretical framework which solves the tasks of interest. Third, we implement our framework in a prototype tool. Fourth, we evaluate the tool on the subset of benchmarks not used in the case study. Finally, we use the results of the evaluation as a feedback for refining our framework: if our termination criteria (which we discuss below) are not satisfied, we go back to step 1.

In the course of this dissertation, we refined the framework and the set of roles in several iterations. In particular, we changed the underlying formalism for the task *T1 (Role specification framework)* from data-flow analysis to logic programming (see Section 3.2 for details). We also added several roles specific for the tasks *T2 (Portfolio solver for SV)* and *T3 (Systematic heuristics for SV)* in a number of iterations so that the respective termination criteria were met. We now describe in detail the steps of the cycle and define the termination criteria.

Case study. To identify typical patterns of variable use, we do a case study on a subset

of a comprehensive code base of real-world software. To identify roles for the tasks *T2 (Portfolio solver for SV)* and *T3 (Systematic heuristics for SV)*, we analyse the challenges which the benchmarks of the SV-COMP competition exhibit for the participating tools and the challenging benchmarks for the tool ELDARICA respectively.

Theoretical framework. We develop a framework for the specification and inference of roles, and specify in this framework the variable roles identified in the previous step. Using these roles, we devise an algorithm for a portfolio solver for SV and define heuristics for ELDARICA.

Prototype tool. First, we implement a prototype tool for automatic inference of roles. Second, we implement a prototype portfolio solver. Third, we implement a tool which automatically annotates C programs for ELDARICA according to the heuristics defined in the previous step.

Experiments and termination criteria. Finally, we evaluate the prototype tools on a set of benchmarks.

We evaluate the portfolio solver on the benchmarks of the competition SV-COMP'14 and compute the score of the portfolio solver in the competition. We analyse the cases of non-optimal choices of the portfolio solver and refine the respective set of roles. We continue until the portfolio solver beats other tools by a clear margin.

We evaluate ELDARICA with role-based heuristics on a set of benchmarks. We refine the respective set of roles and the heuristics until we obtain a substantial increase in the number of tasks solved by ELDARICA with role-based heuristics. Specifically, we set ourselves the goal to increase the number by 10%.

Requirements. We aim to devise a specification language for variable roles which is both concise and expressive, so that typically used patterns of variable use in real-word software can be formulated and the role definitions are concise.

Our implementation is intended to handle programs written in the C programming language. The concepts and algorithms that we develop shall, however, be general enough such that they can be applied also to other imperative languages.

Restrictions. To make the *Role specification framework* task feasible within the scope of a PhD dissertation, we make the following assumptions. First, we restrict our study to imperative languages. Second, the roles we devise apply to scalar and pointer variables, and structure fields of a scalar type and pointer type — we leave out variables of structure type. Finally, for the sake of efficiency of role inference, we only consider flow-insensitive analyses for roles. All three restrictions are not intrinsic, and extending our work in any of the three directions would make for interesting future work.

Methodology. We formulate the inference algorithm as a light-weight static analysis, i.e. an efficient analysis executed at compile time. In our approach, we define a variable role as a data-flow analysis which assigns to each variable at each program control location zero or more variable roles.

In particular, we follow a common approach and specify the analysis for a role using logic programming [CGT89], as e.g. in [HVdM06, Rep95]. Our algorithm represents the program transition relation as a database of facts and defines each role with a set of logic rules. The inference of roles reduces to the inference of facts which encode the assignment of roles to variables.

1.6 Structure of the Thesis

We organise the thesis as follows. In Chapter 2 we give preliminaries on software verification and give examples of empirical choices in verification tools.

In Chapter 3 we define variable roles. We first informally describe the variable roles which we identified for the tasks of the thesis (see Section 1.5 for the definition of the tasks). We then define a framework for the specification and inference of roles and formally define the roles. Finally, we describe an extension of our framework to inter-procedural analysis and discuss the details of our implementation.

In Chapters 4 and 5 we explore the application of roles in software verification. Specifically, in Chapter 4 we define a portfolio solver for the SV-COMP software verification competition. We first introduce our program metrics based on variable roles and other features extracted with data-flow analyses. We then describe the algorithm of our portfolio solver which uses the program metrics and evaluate the portfolio solver on the SV-COMP benchmarks from the years 2014, 2015 and 2016.

In Chapter 5 we define a framework for the specification of role-based heuristics for software verification and do a case study using the model checker Eldarica. We first describe the preliminaries of the model checking technique employed by Eldarica, which reduces program verification to the problem of satisfiability of a system of Horn clauses, i.e. logical formulae of a special kind. We then describe our heuristics for Eldarica based on variable roles and, finally, evaluate Eldarica on a set of benchmarks from SV-COMP and related literature.

In Chapter 6 we give an overview of related work. We conclude and discuss threats to validity and possible directions for future work in Chapter 7.

1.7 Contributions

In this dissertation we make the following contributions:

1. We give a *formal* definition of the concept of variable roles for imperative programming languages:

   a) We devise a set of variable roles which capture typical usage patterns of variables in open-source industrial benchmarks;

b) We propose a concise specification formalism for variable roles based on logic programming which at the same time lends itself as a technique for automatic inference of roles;

2. We explore the application of variable roles in software verification:

   a) First, we identify variable roles important for the benchmarks of the SV-COMP software competition. We devise source code metrics based on these roles. Using the metrics, we build a portfolio solver for software verification. We show that the portfolio solver would be a hypothetical overall winner of the competition in three consecutive years (2014–2016).

   b) Second, we suggest a method of specifying heuristics based on variable roles for choosing program-specific abstraction. We do a case study on the model checker Eldarica and identify variable roles important for Eldarica on a set of software verification benchmarks. We show that not only all the existing heuristics of Eldarica can be expressed using variable roles, but also that the extended tool solves 11.2% more tasks on a set of benchmarks for software verification, and shows a significant speedup on certain benchmark families.

1.8 Our Publications on the Topic of the Thesis

We made our first attempt to formally define variable roles in [DVZ13]. In particular, we proposed a set of 14 roles which capture frequent usage patterns of variables in imperative programs. We defined roles using intra-procedural data-flow analysis, and used a standard fixed-point algorithm [NNH99] to infer the roles. We implemented the role inference algorithm in a prototype tool. As a proof of meaningfulness of the variable roles we used the roles to classify the benchmarks of the Software Verification Competition SV-COMP'13, specifically to predict membership of the benchmarks to different categories of the competition. My contributions to [DVZ13] are as follows:

- inspecting benchmarks and identifying frequently used code patterns;
- devising role specifications;
- implementing and evaluating the prototype tool.

Next, in [DPVZ15] and [DPVZ16], we used variable roles to devise program metrics and, based on these metrics, built a portfolio solver for software verification. For the metrics we used the variable roles from [DVZ13] as well as 12 new roles which capture usage patterns of variables important for software verification. My contributions to [DPVZ15] and [DPVZ16] are as follows:

- identifying and specifying new roles for the portfolio solver;
- implementing and evaluating an algorithm for the portfolio solver.

Finally, in [DRZ17], we used variable roles to systematically specify heuristics in software verification for creating program-specific abstractions. In particular, we defined 5 heuristics for the generation of predicates and predicate templates, the latter guiding Craig interpolation during abstraction refinement. Differently from [DVZ13], [DPVZ15] and [DPVZ16], we introduced a new formalism for the specification of variable roles. Specifically, we defined roles as logic queries on the structure of a program, which allowed to separate role specification from implementation into configuration files. My contributions to [DRZ17] are as follows:

- analysing the cases of failure of Eldarica on a set of SV-COMP benchmarks and devising *missing* predicates which (along with the predicates generated by Eldarica) would allow Eldarica to verify the benchmarks;
- identifying variable roles capturing the missing predicates and devising role-based heuristics to generate the predicates;
- devising the new logic-based framework for the specification of variable roles and translating the existing specifications of roles to the new formalism;
- implementing the algorithms for role inference and predicate generation in a prototype tool.

This dissertation is a cumulative summary of [DVZ13], [DPVZ15], [DPVZ16] and [DRZ17]. The thesis extends, however, our previous works by giving a full specification of variable roles in Chapter 3 and auxiliary definitions in Section 8.A. In addition, in Chapter 2 we give an overview of the notions and techniques which introduce the reader to the basics of software verification. Finally, in Chapter 6 we provide a detailed review of the related literature, which is not part of our previous works.

CHAPTER 2

Background

In Chapter 1 we stated that state-of-the-art software verification tools implement different heuristics, and that these heuristics are not systematically specified and described. In this chapter we discuss this problem in more detail. To this end, in Sections 2.1 and 2.2 we first give preliminaries on the model checking and abstract interpretation verification methods respectively. with examples of heuristics for both methods in several state-of-the-art software verification tools.

2.1 Model Checking

Model checking is a method of verification which solves the following problem: given a model of a system, exhaustively and automatically check whether this model meets a given specification. Specifically, methods of model checking check whether a structure representing a model of a system satisfies a logical formula representing a specification.

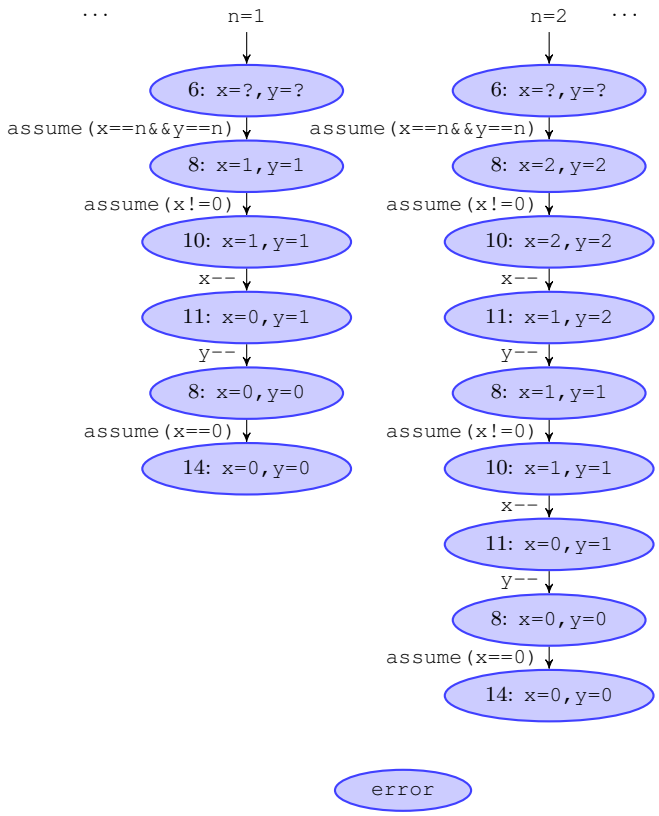
2.1.1 Labelled Transition System

An important class of model checking techniques reduces a verification problem to a graph search. In particular, assertion properties check that some condition holds in a given control location. Examples of assertion properties are checks that there is no division by zero, there is at most one process in a critical section, or the result of computations satisfies a functional specification. Model checking techniques translate an assertion property to a dedicated *error* state in the program state space, and reduce the verification task to a graph reachability problem. These techniques represent a program as a labelled transition system (S, S_{init}, R, T), with a set of states S, a set of initial states $S_{init} \subseteq S$, the transition relation $R \subseteq S \times Stmt \times S$ and the set of transitions $T \subseteq Loc \times Stmt \times Loc$.

The set of program states $S = Loc \times Val$ contains pairs (ℓ, v) of a program location $\ell \in Loc$ and a valuation $v \in Val$, where $Val = [Var \to I]$ is a set of mappings from

```
1 extern int n;
2
3 void main()
4 {
5   int x, y;
6   assume(x==n && y==n);
7
8   while (x!=0)
9   {
10    x--;
11    y--;
12  }
13
14  assert(y==0);
15 }
```

(a) Source code



(b) Labelled transition system

Figure 2.1 Concrete model of program in model checking.

program variables Var to values I. The set of program states is called program *state space*.

The transition relation is defined as $R = \{((\ell_1, v_1), stmt, (\ell_2, v_2)) \mid (\ell_1, stmt, \ell_2) \in T \wedge [\![stmt]\!](v_1) = v_2\}$. Here $[\![-]\!] : Stmt \to (Val \rightharpoonup Val)$ is a semantic function which transforms a statement *stmt* to a partial function, which in turn transforms an initial valuation v_1 to a valuation v_2 after executing *stmt*.

Example 2.1.1. For example, consider the program in Fig. 2.1a. We show the labelled transition system of the program in Fig. 2.1b. The labelled transition system contains an infinite number of graphs, one for each initial value of the variable `n`, indicated with labels `n=1`, `n=2` etc., on top of each graph.

We show the graphic representation of a labelled transition system as follows:

- The nodes in the labelled transition system correspond to states S. We label each node corresponding to a state $s = (\ell, v)$ with a pair of a location $\ell \in Loc$

(specifically, the line number in source code) and a valuation $v \in Val$, separated by a column.

- The edges correspond to the transition relation R. Each edge corresponding to a relation $r = (s_1, stmt, s_2)$ goes from the node corresponding to the state s_1 to the node corresponding to the state s_2, and we label such an edge with the statement $stmt$.

  We show every evaluation of a loop condition `cond` as an edge labelled with statement `assume(cond)` or statement `assume(!cond)` for the cases when the condition holds and does not hold respectively. For example, in Fig. 2.1b the loop condition corresponds to edges `assume(x!=0)` and `assume(x==0)`.

Consider the case when the value of variable `n` is 1. Initially, the values of the variables `x` and `y` are undefined, which we denote with `x=?` and `y=?` in state labels. Then the statement `assume(x==n && y==n)` is executed, leading to the state `x=1,y=1`. Next, the loop condition `x!=0` is evaluated to `true`, leading the system to a state with same variable values. In the first loop iteration the decrement statement `x--` in line 10 leads to the state `x=0,y=1` and the statement `y--` in line 11 leads to the state `x=0,y=0`. Then the loop condition is evaluated to `false`, leading the system to the state with same variable values. Finally, the control proceeds to the assertion in line 14. The assertion condition `y==0` evaluates to `true`, with variable values not changed, and the program terminates.

Similarly, if `n=2`, the loop terminates after two iterations. For any `n`, executing the loop `n` times proves that the error state is not reachable for this `n`. However, since the labelled transition system contains infinitely many states, it is not possible to exhaustively explore the state space. ▲

State space explosion problem. Some model checkers, e.g. CBMC [CKL04], use the fact that integers manipulated by machines have a fixed bit width, and their values are bounded. For example, for n-bit counter, the number of states is 2^n. However, the number of program states is exponential in the number of variables, and is typically intractably large. For example, if a program contains k n-bit variables, its state space contains $2^{n \cdot k}$ states. In case of dynamic data structures or loops, the number of executions for which is unknown at compile time, the number of program states becomes infinite. Therefore, in order to reduce the number of program states to a finite and tractable number, verification tools are typically combined with *abstraction*.

2.1.2 Predicate Abstraction

An important class of abstraction is *predicate abstraction* [GS97], defined with a set of predicates P, which are first-order formulae, e.g. `x>0`, or `x+y<z`, or `∀i(arr[i]>0)`, etc. Given a set of predicates P, we define an extended set of predicates $P_e = P \cup \{\neg p \mid p \in P\}$ and a set of abstract states $S^\# = Loc \times \mathcal{P}(P_e)$, the latter called abstract state space.

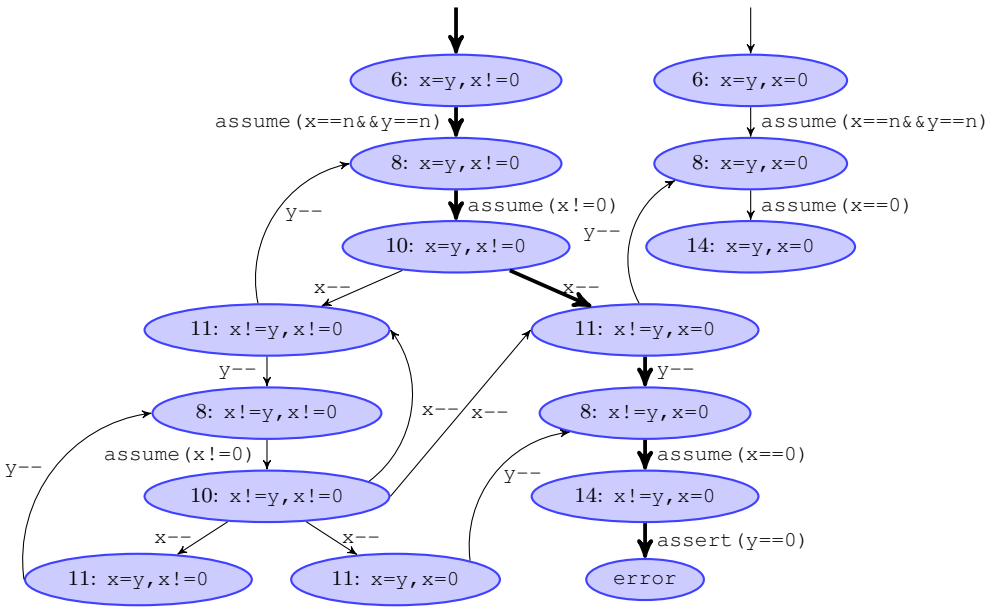


Figure 2.2 Abstract labelled transition system of the program in Fig. 2.1: $P_1 = \{\texttt{x=y,x=0}\}$

An abstract state is computed with an abstraction function $\alpha : S \to S^{\#}$ which maps a concrete state to the corresponding abstract state as follows: $\alpha((\ell, a)) = (\ell, \{p \in P_e \mid a \models p\})$.

Abstract labelled transition system. The abstract labelled transition system for a labelled transition system (S, S_{init}, R, T) is then defined as $(S^{\#}, S^{\#}_{init}, R^{\#}, T)$, where

- the set of abstract states is defined as $S^{\#} = \{\alpha(s) \mid s \in S\}$,
- the set of abstract initial states is defined as $S^{\#}_{init} = \{\alpha(s) \mid s \in S_{init}\}$ and
- the abstract transition relation $R^{\#} \subseteq S^{\#} \times Stmt \times S^{\#}$ is defined as $R^{\#} = \{(\alpha(s_1), stmt, \alpha(s_2)) \mid (s_1, stmt, s_2) \in R)\}$.

We will now illustrate predicate abstraction on example of the program in Fig. 2.1a. We will create abstract labelled transition systems for two sets of predicates: $P_1 = \{x = y, x = 0\}$ and $P_2 = \{x = y, x = 0, x = y - 1\}$. We will show that the abstract system for P_1 is not precise enough to prove the property, while the abstract system for P_2 proves the program safe.

Example 2.1.2 ($P_1 = \{x = y, x = 0\}$). Consider the code in Fig. 2.1a. A crucial fact for proving that the assertion condition holds is that the variables x and y have the same initial value and get decremented by the same value in the loop, and also that the value of x is 0 at the loop exit. The predicates x=y and x=0 are suitable to prove the

property, because from the facts that x=y holds at the beginning of every loop iteration, and x=0 holds at loop exit, follows that the assertion condition y==0 evaluates to true.

We show the labelled transition system for the program abstraction with the set of predicates P_1 ={x=y, x=0} in Fig. 2.2. To denote equality in node labels, we use the symbol =, instead of C equality operator ==.

We will now explain step-wise how this abstraction is created, for a while not making difference between thick and normal arrows. Initially, the variable values are unknown, and the statement assume(x==n&&y==n) in line 6 leads to a state where the predicate x=y holds. We do not show the initial states where x!=y holds, since the transition relation does not include elements for these states.

Since the value of n is defined externally, it is not possible to reason at compile time whether x=0 or its negation holds at line 6. Therefore, the set of concrete initial states corresponds to two abstract states 6:(x=y,x=0) and 6:(x=y,x!=0) (here and below in this chapter, we separate the control location in a state from the variable assignment with a colon). When the loop condition x!=0 at line 8 is evaluated in the state 8:(x=y,x=0), the condition does not hold. The assertion statement assert(y==0) at line 14, evaluated in the same state, does not lead to an error state and the program terminates.

When the loop condition is evaluated in the abstract state 8:(x=y,x!=0), the loop condition evaluates to true and the control reaches the loop body at line 10. Executing the statement x-- leads to a state in which x!=y holds and either x=0 or its negation holds, since x=y∧x!=0∧x′=x-1 is satisfiable both for x′=0 and for x′!=0. Therefore, the abstract transition relation contains elements (10:(x=y,x!=0), x--, 11:(x!=y,x=0)) and (10:(x=y,x!=0), x--,11:(x!=y,x!=0)).

In state 11:(x!=y,x=0), executing the statement y-- either leads to the state 8:(x=y,x=0), or to the state 8:(x!=y,x=0), since the abstraction does not tracks the fact that the variables x and y were decremented by the same value in the loop. Similarly, executing the statement y-- in the abstract state 11:(x!=y,x!=0) either leads to the state 8:(x=y, x!=0), or to the state 8:(x!=y, x!=0).

In a similar way, executing the loop condition assume(x!=0) in state 8:(x!=y,x!=0) leads to a (not yet visited) state 10:(x!=y,x!=0). Executing the statement x-- leads to two (not yet visited) states 11:(x=y,x!=0) and 11:(x=y,x!=0), as well as to two (already visited) states 11:(x!=y,x!=0) and 11:(x!=y,x=0). Executing the statement y-- in the states 11:(x=y,x!=0) and 11:(x=y,x!=0) leads to the already visited states 8:(x!=y,x!=0) and 8:(x!=y,x=0) respectively.

Finally, after one or more iterations, evaluating the loop condition in the state 8:(x!=y, x=0) leads to the state 14:(x!=y,x=0). Executing the statement assert(y==0) in the state 14:(x!=y,x=0) leads to the error state. Therefore, the error state is reachable in this abstract labelled transition system. ▲

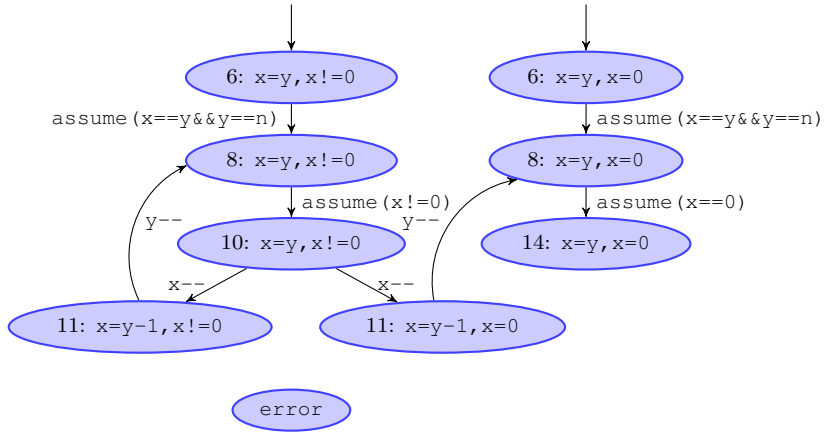


Figure 2.3 Abstract labelled transition system of the program in Fig. 2.1: $P_2 = \{\texttt{x=y}, \texttt{x=0}, \texttt{x=y-1}\}$

Example 2.1.3 ($P_2 = \{x = y, x = 0, x = y - 1\}$). Now we consider the set of predicates $P_2 = P_1 \cup \{x = y - 1\}$. We show the abstract system for P_2 in Fig. 2.3. We will also explain how the abstract system for P_2 is constructed and show that in this abstract system the `error` state is not reachable, i.e. the assertion property holds.

First, when constructing the abstract labelled transition system, the states 6:(x=y,x=0), 6:(x=y,x!=0), 8:(x=y,x=0), 8:(x=y,x!=0), 14:(x=y,x=0) and 10:(x=y,x!=0) are created in the same way as in the system for P_1. Similarly to the system for P_1, program execution from the abstract state 14:(x=y,x=0) terminates in two steps without reaching the error state. For the sake of brevity, we do not show the predicate x!=y-1 in these states, since it is implied by the predicate x=y.

Then, in the state 10:(x=y, x=0), evaluating the statement x-- leads either to the state 11:(x=y-1,x=0), or to the state 11:(x=y-1,x!=0). In both states we do not show the predicate x!=y, since it is implied by the predicate x=y-1.

Finally, executing the statement y-- in the states 11:(x=y-1,x=0) and, 11:(x=y-1,x!=0) leads to the states 8:(x=y,x=0) and, 8:(x=y,x!=0) respectively.

After one or more iterations, evaluating the loop condition in the state 8:(x=y,x=0) leads to the state 14:(x=y,x=0). Executing the assertion statement in the state 14:(x=y,x=0) does not lead to an error state.

To summarise, exploring all possible transitions shows that the error state is not reachable in this abstract labelled transition system. ▲

2.1.3 Abstraction Refinement

Finally, we show how to construct more precise abstractions incrementally using an automatic technique called conterexample-guided abstraction refinement (CEGAR) [CGJ+03].

Counterexample. First, a *counterexample* is located in the abstract system, i.e. a path from an initial state to an error state. If there is at least one path in the concrete model which is mapped to a counterexample using the function α, then the counterexample is called *feasible*, otherwise it is called *spurious*. For example, in Fig 2.2 a counterexample is shown with bold arrows. This counterexample is infeasible and thus is spurious. To eliminate a spurious counterexample, model checking methods refine the set of predicates using systematic techniques like *Craig interpolation*, or heuristics.

Craig interpolation. Craig interpolation [Cra57, McM05] finds for a mutually inconsistent pair of formulae (A, B), i.e. $A \wedge B \rightarrow false$, a formula I which is

1. implied by A, i.e. $A \rightarrow I$,
2. inconsistent with B, i.e. $I \wedge B \rightarrow false$ and
3. expressed over the common variables of A and B.

To formulate a Craig interpolation problem, model checking techniques construct a *path formula* for the counterexample in the concrete system as follows: for each edge of the counterexample, the action statement (with which the edge is labelled) is added as a conjunct to the path formula. In particular, each assignment statement is replaced with an equality and the new value of the assigned variable is replaced with a newly introduced primed variable. For example, the concrete path formula of the counterexample in Fig. 2.2 is

$$x \overset{6}{=} n \wedge y \overset{6}{=} n \wedge x \overset{8}{\neq} 0 \wedge x' \overset{10}{=} x - 1 \wedge y' \overset{11}{=} y - 1 \wedge x' \overset{8}{=} 0 \wedge y' \overset{14}{!=} 0. \tag{2.1}$$

For demonstration purposes, we put over each (dis-)equality sign the line number of the corresponding action statement. Note the newly introduced variables x' and y' for the assignment statements in lines 10 and 11 and that the primed variables are used instead of the variables x and y respectively in the conjuncts which are added afterwards.

There are several possible ways to split the path formula to formulae A and B. A possible Craig interpolation problem for this counterexample is (A, B) s.t

$$A = (x = n \wedge y = n \wedge x \; != 0 \wedge x' = x - 1) \text{ and}$$
$$B = (y' = y - 1 \wedge x' = 0 \wedge y' \; != 0).$$

The formulae A and B are inconsistent since the counterexample is infeasible. For this Craig interpolation problem a possible interpolant is the formula $I = (x' = y - 1)$, since

1. $x = n \wedge y = n \wedge x' = x - 1 \rightarrow \boldsymbol{x' = y - 1}$ and
2. $\boldsymbol{x' = y - 1} \wedge y' = y - 1 \wedge x' = 0 \wedge y' \; != 0 \rightarrow false$ and

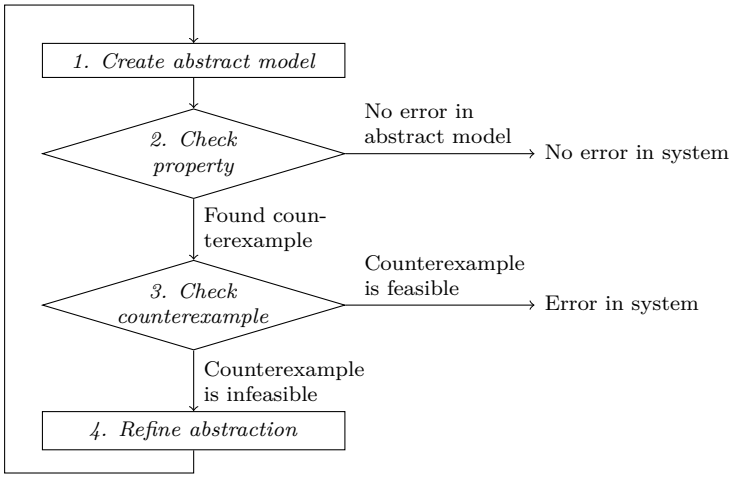


Figure 2.4 CEGAR (Counterexample-guided abstraction refinement)

3. I uses the common variables of A and B.

CEGAR. As mentioned above, some model checking tools, e.g. Blast, CPAchecker, UltimateAutomizer etc., use interpolants to refine a set of predicates. To make the process of abstraction refinement (and model checking as a whole) automatic, many software model checking tools, e.g. Blast, CPAchecker, Eldarica etc., use the technique *counter-example abstraction refinement* (CEGAR).

CEGAR iteratively solves a verification task as shown in Fig. 2.4. In the 1 and 2 steps of each iteration, an abstract system is created and the property is checked. By the definition of the function α, if the property holds in the abstract system, then it also holds in the concrete system (denoted as "No error in system"). If the property does not hold in the abstract system, a counterexample is extracted and checked for feasibility in step 3. If the counterexample is feasible, then there is an error in the concrete model (denoted as "Error in system"), with the counterexample illustrating how the property is violated in the model. If the counterexample is infeasible, then the abstraction is refined in step 4, and the loop is repeated from step 1.

2.1.4 Heuristics in Model Checking

Heuristics in model checking are used both to find *initial abstraction*, e.g. the set P_1 in Example 2.1.2, and during *abstraction refinement*, e.g. to find the predicate `x=y-1` in the set P_2 in Example 2.1.3. In this section we give examples of such heuristics in state-of-the-art model checking tools.

SLAM. The model checker Slam, based on predicate abstraction with CEGAR, is targeted at verifying Windows device drivers. Slam is included into Windows

```
1 #define Locked 0
2 #define Unlocked 1
3
4 void AcquireSpinLock(int state) {
5   assert(state == Unlocked);
6   state = Locked;
7 }
8
9 void ReleaseSpinLock(int state) {
10  assert(state == Locked);
11  state = Unlocked;
12 }
13
14 void foo() {
15  int protect = 1;
16  int state = Unlocked;
17  int *a;
18  ...
19  if (protect)
20    AcquireSpinLock();
21
22  for (i=0; i < 1000; a[i] = 0, i++);
23
24  if (protect)
25    ReleaseSpinLock();
26 }
```

Figure 2.5 Code example illustrating the heuristics of Yogi.

Static Driver Verifier (SDV) toolkit and uses built-in domain-specific heuristics to extract refinement predicates from particular counterexamples [JM09, p. 24]. In particular, Slam queries the theorem prover Zapato to check the feasibility of a counterexample and, in case the counterexample is infeasible, returns a minimal set of unsatisifiable constraints. The atomic formulae from this set are then used by Slam as refinement predicates.

Zapato takes a quantifier-free first-order logic query and transforms it to constrains of two types. The first set contains equality constraints for uninterpreted function symbols. The second set contains the constraints of the form $ax + by \leq d$, where x and y are variables, d is an integer and a and b are elements of the set $\{-1, 0, 1\}$. Even though the obtained refinement predicates are of a restricted form, the experimental results show, that the logic used by Zapato is sufficient for the verification of Windows device drivers [BCLZ04].

Yogi. The model checker Yogi is also targeted at verifying Windows device drivers and combines static analysis with symbolic execution. Symbolic execution computes the result of executing a program path by consecutively applying to a symbolic input the semantic function of the program statements which occur on the path. Yogi

implements an iterative algorithm, which on one hand uses symbolic execution to generate a (symbolic) input leading the program execution to the error state, and on the other hand performs static analysis to prove that the error state is unreachable.

The initial abstraction created by Yogi corresponds to an abstract labelled transition system which contains one abstract state for each control location in a program.

1. The first heuristic Yogi implements adds to the initial abstraction the predicates extracted from the conditionals of the verification property.

   Example 2.1.4. Consider the code in Fig. 2.5, which initialises the array `a` in a loop in line 22 and protects the initialisation by assigning to the variable `state` the value `Locked`. The dots in Fig. 2.5 denote that part of the code was omitted. The heuristic adds to the initial abstraction the predicates `state==Locked` and `state==Unlocked`. ▲

In case the static analysis performed by Yogi generates a counterexample, Yogi uses symbolic execution to check whether the counterexample is feasible. If the counterexample is infeasible, Yogi uses the following two heuristics to refine predicates at the point of `assume` statements; the `assume` statements being used by Yogi to represent the conditions of `if-then-else` statements and of looping constructs:

2. First, to create an abstraction of a program, the heuristic replaces the `assume` statements on which the error state is not control dependent by `skip` statements. However, in case the symbolic execution performed during feasibility check determines that some of the removed `assume` statements occur on the executed path, the heuristic adds back to the abstraction the respective `assume` statements.

   Example 2.1.5. Consider again the example in Fig. 2.5. The heuristic replaces with a `skip` statement the statement `assume(i<1000)`, which represents the loop condition in line 22.

   Suppose that Yogi needs to perform a refinement at line 22. In this case Yogi adds the statement `assume(i<1000)` back to the abstraction. ▲

3. Second, the algorithm of Yogi performs the refinement as follows. For an infeasible abstract path $(s_{k-1}, \texttt{assume}(\phi), s_k)$ in the counterexample, s.t. $s_{k-1} = (\ell_{k-1}, P_{k-1}), s_k = (\ell_k, P_k)$ and $P_{k-1}, P_k \in P_e$, Yogi refines the abstract state s_{k-1} with the predicates $P_k \cup \{\phi\}$.

   The heuristic replaces the refinement $P_k \cup \{\phi\}$ with the set P_k if the latter is strong enough to eliminate the counterexample. A smaller set is preferable because it makes the evaluation less computationally expensive.

   Example 2.1.6. Continuing the example 2.1.5, suppose Yogi needs to perform a refinement at line 22. Suppose the infeasible abstract path in the

counterexample is $(s_{k-1}, \texttt{assume(i<1000)}, s_k)$, where $s_{k-1} = (20, \{\})$ and $s_k = (22, \{\texttt{state==Locked}\})$.

The algorithm of YOGI would refine the state s_{k-1} with the set of predicates $\{\texttt{state==Locked}, \texttt{i<1000}\}$. The heuristic is able to eliminate the predicate `i<1000` from the refinement and simplify the abstraction. ▲

The evaluation of YOGI on SDV benchmark suite showed that the three heuristics improve the performance of YOGI on Windows device drivers [NR10].

Eldarica. The model checker ELDARICA [HKG+12, LRS16] is based on predicate abstraction with CEGAR. To eliminate spurious counterexamples, ELDARICA uses Craig interpolation, which is implemented in a theorem prover. In order to guide the theorem prover to the interpolants containing the suitable predicates in a generally infinite lattice of interpolants, ELDARICA uses *interpolation abstraction* [LRS16].

In this method, ELDARICA instruments interpolation queries with *templates*, which are formulae restricting the symbols that can occur in interpolants. Specifically, the interpolant is built only using the terms in the set of templates and numbers.

Example 2.1.7. 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, 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, e.g. by preventing theorem provers to compute I_1 as an interpolant. For this, template terms are used which capture the expressions that an interpolant should 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 can return $(x - y = -1) \equiv I_2$, but no longer I_1.

To achieve this, ELDARICA rewrites the interpolation query. In particular, given the template $x - y$, ELDARICA rewrites the formula A to A_r and the formula B to B_r as follows:

$$\begin{array}{rcl} A_r & = & (x' = 1 \wedge y' = 2) \wedge \boldsymbol{x' - y' = x - y}, \\ B_r & = & \boldsymbol{x'' - y'' = x - y} \wedge (x'' > y'') \end{array} .$$

The rewriting consists of two parts:

- The variables x and y are renamed to x' and y' in the formula A and to x'' and y'' in the formula B.
- In this way, the knowledge about the exact values of the variables x and y is lost. Instead, the limited knowledge about the difference $x - y$ is introduced (see the bolded parts of the formulae).

As a result, I_1 is no longer a valid interpolant of A_r and B_r (because $A_r \rightarrow I_1$ is not valid any more), but I_2 is. ▲

2.2 Abstract Interpretation

Abstract interpretation is another method of verification which creates an approximation of reachable program states. Specifically, reachable states are the states which can be reached from an initial state in zero or more steps, and an approximation of the set of reachable states is a set which includes all reachable states, and possibly other (unreachable) states. We will now define basic notions used in abstract interpretation [NNH99].

2.2.1 Basic Definitions

A *partial order* $(L, \sqsubseteq)$ is a set L equipped with a binary relation $\sqsubseteq \subseteq L \times L$, s.t. the relation $\sqsubseteq$ is

- reflexive (i.e. $\forall x \in L.x \sqsubseteq x$),
- transitive (i.e. $\forall x, y, z \in L.x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$) and
- anti-symmetric (i.e. $\forall x, y \in L.x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow y \sqsubseteq x$).

Element $y \in L$ is an *upper bound* of subset X of L if $\forall x \in X.x \sqsubseteq y$. Similarly, element $y \in L$ is a *lower bound* of subset X of L if $\forall x \in X.y \sqsubseteq x$.

A *least upper bound* $\sqcup X$ of X is an upper bound of X which for all upper bounds y of X satisfies $\sqcup X \sqsubseteq y$. Similarly, a *greatest lower bound* $\sqcap X$ of X is a lower bound of X which for all lower bounds y of X satisfies $y \sqsubseteq \sqcap X$. We will write $x \sqcup y$ instead of $\sqcup\{x, y\}$ and $x \sqcap y$ instead of $\sqcap\{x, y\}$.

A *lattice* is a partial order $(L, \sqsubseteq)$ s.t. for every two elements $x, y \in L$ there is a least upper bound $x \sqcup y$ and a greatest lower bound $x \sqcap y$. We denote the largest element of L with $\top$, if it exists: $\top = \sqcup L$. Similarly, we denote the smallest element of L with $\bot$, if it exists: $\bot = \sqcap L$.

A *chain* Y is a subset of a lattice L s.t. Y is totally ordered:

$$\forall \ell_1, \ell_2 \in Y.(\ell_1 \sqsubseteq \ell_2) \vee (\ell_2 \sqsubseteq \ell_1).$$

A *finite* chain Y is a chain which is a finite subset of L. A lattice L has *finite height* if all its chains are finite.

An *ascending chain* is a sequence $(\ell_n)_n$ of elements in L s.t.

$$\forall n, m \in \mathbb{N}.n \leq m \Rightarrow \ell_n \sqsubseteq \ell_m.$$

We say that a sequence $(\ell_n)_n$ *eventually stabilises* if

$$\exists n_0 \in \mathbb{N}.\forall n \in \mathbb{N}.n \geq n_0 \Rightarrow \ell_n = \ell_{n_0}.$$

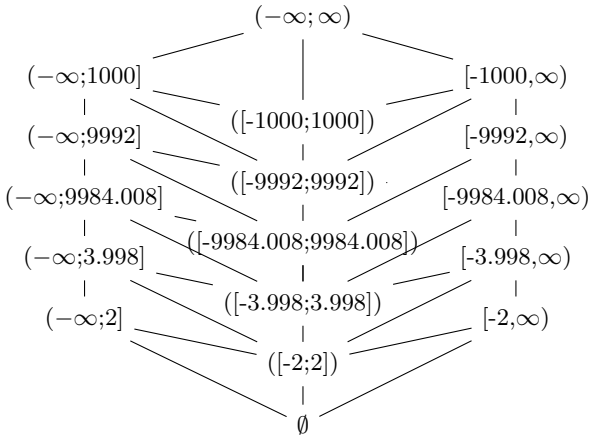


Figure 2.6 Lattice of a subset of intervals of rational numbers.

2.2.2 Abstract Elements

An abstract domain in abstract interpretation is a lattice of *abstract elements*.

Example 2.2.1 (Interval analysis). For example, the abstract domain which over-approximates variable values with intervals is called *interval domain*. Elements of the corresponding lattice are intervals. We illustrate the lattice of intervals of rational numbers $\{[\ell, h] \mid \ell, h \in \mathbb{Q} \cup \{-\infty, \infty\} \wedge \ell \leq h\}$ in Fig. 2.6. We will later use this lattice in our examples. ▲

Given a concrete domain of states S, an abstract domain L and functions $\alpha : \mathcal{P}(Val) \to L$ and $\gamma : L \to \mathcal{P}(Val)$, a *Galois connection* is a tuple $(L, \alpha, \gamma, \mathcal{P}(Val))$ s.t. γ preserves arbitrary meets:

$$\gamma(\bigsqcap X) = \bigcap_{x \in X} \gamma(x)$$

The function α is called *abstraction* function, and the function γ is called *concretisation* function.

2.2.3 System of Equations

Given a labelled transition system (S, S_{init}, R, T) of a program and a lattice L of abstract elements, the methods of abstract interpretation construct a recursive system of equations

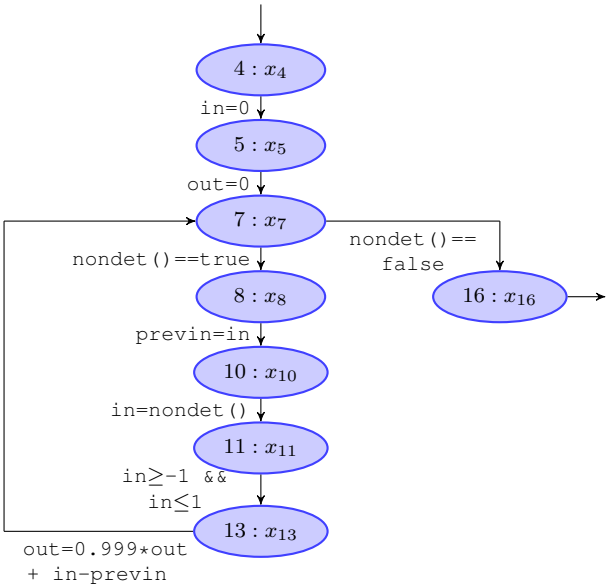
$$(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n)) \tag{2.2}$$

where

- $n = |Loc|$ is the number of control locations of the program;

```
1 extern int nondet();
2
3 void main() {
4   int in = 0;
5   int out = 0;
6
7   while (nondet()) {
8     previn = in;
9
10    in = nondet();
11    assume(in>=-1 && in<=1);
12
13    out=0.999*out+in-previn;
14  }
15
16  exit:
17 }
```

(a) Source code.



(b) Control flow graph.

Figure 2.7 High bandpass filter. The example is taken from a preprint[1] with minor modifications.

- For each control location $\ell_i \in Loc$, s.t. $1 \leq i \leq n$, abstract element $x_i \in L$ is an abstraction of a set of concrete valuations *reachable* at location ℓ_i (recall that we define a state s_i as a pair of a location ℓ_i and variable valuation v_i):

$$\begin{aligned} x_i = \alpha(\{v^i \mid\ & \exists \ell^0, \ldots, \ell^{i-1} \in Loc.\ \exists v^0, \ldots, v^{i-1} \in Val. \\ & \exists stmt^1, \ldots, stmt^i \in Stmt.\ (\ell^0, v^0) \in S_{init} \wedge \\ & ((\ell^0, v^0), stmt^1, (\ell^1, v^1)) \in R \wedge \ldots \wedge \\ & ((\ell^{i-1}, v^{i-1}), stmt^i, (\ell_i, v^i)) \in R\}). \end{aligned}$$

- For each control location $\ell_i \in Loc$, s.t. $1 \leq i \leq n$, vector function $F_i : L^n \mapsto L$ computes the abstract element x_i from the abstract elements x_j in predecessor locations (i.e. $\exists j.\ \exists stmt.\ (\ell_j, stmt, \ell_i) \in T$):

$$F_i(x_1, \ldots, x_n) = \bigsqcup_{(\ell_j, stmt, \ell_i) \in T} [\![stmt]\!]^\#(x_j).$$

  Here the function $[\![-]\!]^\# : Stmt \to (L \to L)$ computes abstract element $x_i \in L$ in state s_i obtained by executing statement $stmt$ in state s_j.

[1] https://pdfs.semanticscholar.org/8af5/9378233fd8ba49b6b1925b430e5662411dd0.pdf. Accessed 23 January 2018.

Example 2.2.2. For example, consider the program in Fig. 2.7a. The code describes the functioning of a digital filter – a system processing discrete-time signals. The variables `in` and `out` denote the values of the current filter input and output respectively. The variable `previn` stores the value of the previous filter input. The filter output is iteratively calculated from the last output value and the last two input values. The verification task is, given that the interval of the filter input $in \in [-1, 1]$, to compute the interval to which filter output `out` belongs.

In Fig. 2.7b we show the *control-flow graph* of the code in Fig. 2.7a:

- The nodes of the control-flow graph correspond to control locations in the program. We label each node i with a pair (ℓ_i, x_i) of the location ℓ_i and an abstract element x_i in location ℓ_i.
- The edges of the control-flow graph correspond to transitions $t \in T$: for each $t = (\ell_i, stmt, \ell_j) \in T$ there is an edge going from the node i to the node j and labelled with statement *stmt*.

For this example, interval analysis constructs a system of equations 2.3 which looks as follows:

$$
\begin{aligned}
x_4(\texttt{in}) &= \top \\
x_4(\texttt{out}) &= \top \\
x_4(\texttt{previn}) &= \top \\
\\
\boldsymbol{x_5(\texttt{in})} &= \boldsymbol{[0,0]} \\
x_5(\texttt{out}) &= \top \\
x_5(\texttt{previn}) &= \top \\
\\
x_7(\texttt{in}) &= x_5(\texttt{in}) \sqcup x_{13}(\texttt{in}) \\
\boldsymbol{x_7(\texttt{out})} &= \boldsymbol{[0,0] \sqcup 0.999 * x_{13}(\texttt{out}) + x_{13}(\texttt{in}) - x_{13}(\texttt{previn})} \\
x_7(\texttt{previn}) &= x_5(\texttt{previn}) \sqcup x_{13}(\texttt{previn}) \\
\\
x_8(\texttt{in}) &= x_7(\texttt{in}) \\
x_8(\texttt{out}) &= x_7(\texttt{out}) \\
x_8(\texttt{previn}) &= x_7(\texttt{previn}) \\
\\
x_{10}(\texttt{in}) &= x_8(\texttt{in}) \\
x_{10}(\texttt{out}) &= x_8(\texttt{out}) \\
\boldsymbol{x_{10}(\texttt{previn})} &= \boldsymbol{x_8(\texttt{in})} \\
\\
\boldsymbol{x_{11}(\texttt{in})} &= \boldsymbol{\top} \\
x_{11}(\texttt{out}) &= x_{10}(\texttt{out}) \\
x_{11}(\texttt{previn}) &= x_{10}(\texttt{previn})
\end{aligned}
\tag{2.3}
$$

$$\begin{aligned}
\boldsymbol{x_{13}(\texttt{in}) = [-1, 1]} \\
x_{13}(\texttt{out}) &= x_{11}(\texttt{out}) \\
x_{13}(\texttt{previn}) &= x_{11}(\texttt{previn}) \\
\\
x_{16}(\texttt{in}) &= x_7(\texttt{in}) \\
x_{16}(\texttt{out}) &= x_7(\texttt{out}) \\
x_{16}(\texttt{previn}) &= x_7(\texttt{previn})
\end{aligned}$$

▲

where $x_i(var)$ is the value of a variable $var \in Var$ in abstract state x_i, and Var is the set of program variables. An abstract state is an element of the *lattice product*, the latter defined with set $\underbrace{L \times \ldots \times L}_{|Var| \text{ times}}$ and a pointwise order $\sqsubseteq^*$, s.t.

$$(x_1, \ldots, x_{|Var|}) \sqsubseteq^* (x'_1, \ldots, x'_{|Var|}) \quad \Leftrightarrow \quad \forall i.\ 1 \leq i \leq |Var|.\ x_i \sqsubseteq x'_i.$$

The system 2.3 contains three types of equations:

- The equations highlighted with bold correspond to the action statements in the transition system;
- The equation $x_7(\texttt{in}) = x_5(\texttt{in}) \sqcup x_{13}(\texttt{in})$ states, that in location 7 the variable `in` can take either of the values of the variable `in` in predecessor locations 5 and 13. The equations for $x_7(\texttt{out})$ and $x_7(\texttt{previn})$ are defined similarly;
- The remaining equations are of the form $x_i(\texttt{var}) = x_j(\texttt{var})$ and describe the cases when the value of a variable `var` is not modified between states (ℓ_j, x_j) and (ℓ_i, x_i).

2.2.4 Solving a System of Equations

We now describe how abstract interpretation computes the solution of the system of equations 2.2.

Fixed point. By definition, X is a *fixed point* of a function F if $F(X) = X$. Abstract interpretation techniques reduce the solution of the vector equation 2.2 to finding a fixed point of the function $F : L^n \mapsto L^n$, defined as follows:

$$F(x_1, \ldots, x_n) = (F_1(x_1, \ldots, x_n), \ldots, F_n(x_1, \ldots, x_n)). \tag{2.4}$$

If $F_1, \ldots, F_n$ are monotone functions, then there exists the unique least fixed point of function F [NNH99]. The unique least fixed point is computed by iteratively solving the equation 2.2, starting from the initial values $(\bot, \ldots, \bot)$, i.e.

$$(x_1, \ldots, x_n) = \bigsqcup_{i \geq 0} F^i(\bot, \ldots, \bot). \tag{2.5}$$

If the lattice L has a finite height, then the sequence $(x_1, \ldots, x_n)$ eventually stabilises.

If the length of the lattice is infinite, abstract interpretation computes an over-approximation of the least fixed point using a technique called *widening*.

Example 2.2.3. To illustrate the computation of the system of equations, we describe how interval analysis of the program in Fig. 2.7a is computed by the abstract interpretation tool Astrée. First, by iteratively solving the system of equations 2.3 with the initial value $x_7(\texttt{out}) = \emptyset$, Astrée obtains for $x_7(\texttt{out})$ a sequence of intervals

$$(\emptyset, [-2; 2], [3.998; 3.998], \ldots).$$

The above sequence does not stabilise after two iterations. Therefore, according to a heuristic implemented in Astrée, a widening technique is applied. ▲

Widening. An operator $\nabla : L \times L \mapsto L$ is a *widening operator* if the following conditions hold:

- Operator ∇ computes an upper bound of its operands, i.e. for $\ell_1 \nabla \ell_2$ it holds that

$$\ell_1 \sqsubseteq \ell_1 \nabla \ell_2 \quad \wedge \quad \ell_2 \sqsubseteq \ell_1 \nabla \ell_2.$$

- For an ascending chain $(\ell_n)_n$ the chain $(\ell_n^{\nabla})_n$, defined as

$$\begin{array}{rcl} \ell_0^{\nabla} & = & \ell_0 \\ \ell_{i+1}^{\nabla} & = & \ell_i^{\nabla} \nabla \ell_{i+1}, \end{array}$$

  is also ascending and eventually stabilises [NNH99, p. 226].

Each such widening operator represents a heuristic. We give an example for a widening operator later in this section on page 35.

2.2.5 Heuristics in Abstract Interpretation

Below we summarise the stages of abstract interpretation where heuristics are used:

1. Since using all available abstract domains is too costly, an abstract domain or a subset of abstract domains for a program under analysis needs to be chosen. The choice of suitable abstract domains is usually done manually by a user of an abstract interpretation tool.

2. Abstract domains are typically parametrised. The parameters are chosen using one of two ways:

```
1 void main()
2 {
3   int a, b, x, y;
4   if (a>b) {while (a<=0) {a++; b++;}}
5   if (x<y) {while (x>=0) {x--; y--;}}
6 }
```

Figure 2.8 Code example illustrating the packing heuristic for octagon in Astrée, taken from [CCF+09].

a) manually by a user of a tool, e.g. Astrée analyser takes an optional parameter which fixes the array size limit above which arrays are abstracted in a field-insensitive way (if the parameter is not specified, the default value is used); or

b) automatically using heuristics (see below in this section).

Recall from Example 1.1.3 that the static analyser ASTRÉE implements at least 12 abstract domains and 150 configuration options in order to scale to hundreds of thousands of lines of critical embedded C programs. Specifically, using the options a user specifies which abstract domains to use and the parameters of the abstract domains.

We will now give examples of such heuristics implemented in ASTRÉE [CCF+09].

Packing heuristic for octagon abstract domain. The octagon abstract domain allows representing conjunctions of constraints of the form $\pm x \pm y \leq c$, where X and Y are program variables and c is a constant.

However, it is prohibitively expensive to compute the dependencies for all possible combinations of program variables, e.g. for thousands of global variables at each program point. Therefore, Astrée abstract analyser implements a heuristic which splits variables in packs and relates the variables in one pack but not variables in different packs.

The heuristic puts in one pack the variables:
- which are used together in linear expressions and
- which are incremented or decremented within loops,

and computes the transitive closure of linear dependencies within syntactic blocks. Moreover, the heuristic puts a restriction that a variable appears in at most three packs, which limits the number of packs updated at each statement. As a result, the octagon analysis exhibits a linear (rather than theoretical quadratic) *memory* worst-time cost [CCF+09, Section 4.3].

Example 2.2.4. For the code in Fig. 2.8, the heuristic creates two packs: `{x,y}` and `{a,b}`, thus computing the lower and upper bounds for `x-y`, `x+y`, `a-b` and `a+b`. ▲

Widening heuristic. We've mentioned in Section 2.2 on page 32 that Astrée implements a heuristic which applies widening when a fixed point is not reached after two iterations. In particular, the widening operator which Astrée implements is called widening *with thresholds*.

Widening with thresholds replaces each unstable interval bound with the next bound in a finite list T of thresholds. A heuristic of Astrée computes the list of thresholds as a sequence $T = (\pm\alpha\lambda^k)_{0\leq k\leq N}$ where the numbers α, λ and N are the parameters of the heuristic.

In addition, the widening heuristic is parametrised with a pair of numbers (q, r). The parameter q defines the number of times the constraint is unstable before the widening is applied. Second, in order to prevent the intervals from being widened too much, Astrée *reduces* the intervals after widening by applying the function F from Eq. 2.4 to the solution q times.

Example 2.2.5. Recall the Example 2.2.3. For this program, the parameters $\alpha = 1$, $\lambda = 10$, $q = 2$ and $r = 2$ are chosen (the authors of the article[2] do not explicitly state whether these are the default parameters). To over-approximate the fixed point of the system 2.3, Astrée finds the smallest bounds $t_1, t_2 \in T$ which approximate the fixed point:

$$[t_1, t_2] \sqsubseteq [0; 0] \sqcup 0.999 * x_{13}(\texttt{out}) + x_{13}(\texttt{in}) - x_{13}(\texttt{previn}),$$

where the state $s_i^\#$ is an abstract state at code line i and $x_i(v)$ is the abstract value of variable v in abstract state $s_i^\#$. The smallest such interval is $[-10000; 10000]$.

Then, Astrée iteratively reduces the interval by applying the abstract transfer functions of the System 2.3 and obtains the sequence of intervals (see Fig. 2.6 on page 29):

$$\begin{gathered}([-10000; 10000],\\ [-9992; 9992],\\ [-9984.008; 9984.008]).\end{gathered}$$

As a result, Astrée reports for the variable `out` the interval $[-9984.008; 9984.008]$. ▲

Disjunction heuristics. The main source of imprecision in abstract interpretation is approximating disjunctions in non-distributive abstractions, i.e. abstraction which may loose information when computing a join [CCF+09, Section 3.1]. For example, in the interval domain a union of the intervals [1,2] and [5,6] cannot be expressed and is represented by an abstract element [1,6]. To recover the disjunctive information Astrée implements the *trace partitioning* strategy [MR05].

[2]The preprint is available at `https://pdfs.semanticscholar.org/8af5/9378233fd8ba49b6b1925b430e5662411dd0.pdf`. Accessed 23 January 2018.

```
1 int v, sgn;
2
3 if (v<0)
4   sgn = -1;
5 else
6   sgn = 1;
7
8 int y=v/sgn;
```

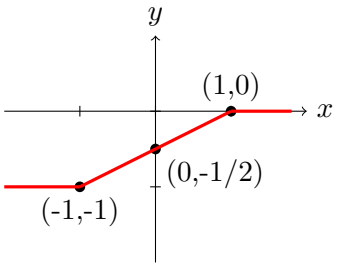
(a) Partitioning strategy.

```
1 int mratio;
2 float x=0, xp;
3
4 while (1) {
5   xp = random(-100,100);
6   mratio = random(0,50);
7
8   x = (x*mratio + xp) / (mratio+1);
9 }
```

(b) Heuristic for division.

```
1 int tc[3]={0; 0.5; 0};
2 int tx[3]={-1; 1; INT_MAX};
3 int ty[3]={-1; -0.5; 0};
4
5 int i=0;
6 assume(x>=-100 && x<=0);
7
8 while (i<2 && x>tx[i]) i++;
9
10 y = tc[i]*x + ty[i];
```

(c) Heuristic for linear interpolation.



(d) Plot of the polygonal path from the program in Fig. 2.9c.

Figure 2.9 Examples illustrating the partitioning strategy in Astrée: (a) the example illustrating the functioning of the strategy, (b),(c) the examples illustrating two heuristics for enabling the strategy.

Trace partitioning. The trace partitioning strategy allows to make a case split on the program traces. A *trace* of a program $(S, I, F, Stmt, T)$ is a sequence of states $(s_i)_{0 \leq i \leq n}$, s.t.

$$\begin{gathered} s_0 \in I \wedge \\ \forall i.\ 0 \leq i \leq n.\ (s_i \in S \quad \wedge \quad (s_i, s_{i+1}) \in T) \wedge \\ s_n \in F. \end{gathered}$$

An *abstract trace* is obtained from a concrete trace $(s_i)_{0 \leq i \leq n}$ by applying the abstraction function $\alpha()$ to every state: $(\alpha(s_i))_{0 \leq i \leq n}$.

Example 2.2.6. Consider the code in Fig. 2.9a, taken from [RM07]. In line 8 the value of `sgn` is either `-1` or `1`. However, an interval analysis would not discover it, since the corresponding abstract value of `sgn` in interval analysis will be $[-1, -1] \sqcup [1, 1]$ which evaluates to $[-1, 1]$. Therefore the program can not be proven safe from division by zero.

With the trace partitioning heuristic, Astrée distinguishes the following two abstract traces:
1. $(s_3^\#, s_4^\#, s_8^\#)$, s.t. $s_8^\# = (8, x_8)$ and $x_8(v) = (-\infty, 0) \wedge x_8(sgn) = [-1, -1]$;

2. $(s_5^\#, s_6^\#, s_8^\#)$, s.t. $s_8^\# = (8, x_8)$ and $x_8(v) = [0, \infty) \wedge x_8(sgn) = [1, 1]$.
Both traces of the program can be proven safe from division by zero. ▲

To automatically determine the cases when trace partitioning should be enabled, Astrée implements several heuristics which we describe below. We take the description of the heuristics as well as the examples from [MR05, Section 4].

1. Barycenter heuristic. In a division operation, when a variable occurs both in the divident and divider and ranges in a finite set with the size smaller than a thousand, then Astrée creates separate partitions for each possible value of the variable.

Example 2.2.7. Consider the code in Fig. 2.9b. The code computes in a loop the barycenter (i.e. the center of mass) of several objects:

- At line 2 `x` is initialised with `0`;
- At each loop iteration at line 8 the barycenter `x` of two objects is computed:
  - The coordinate of the first object is the old value of `x`;
  - The coordinate `xp` of the second object is initialised at line 5 with a value in the interval `[-100,100]` (the function `random(v1,v2)` returns a pseudo-random integer in the interval `[v1,v2]`);
  - The mass of the first object is `mratio` times greater than the mass of the second object, where the value `mratio` is initialised in line 6 with a pseudo-random integer in the interval `[0,50]`.

The verification task is to prove that the value of `x` does not overflow.

Using non-relational domains abstract interpretation techniques cannot prove the task. For example, in an interval analysis the value of `x` diverges to the interval $(-\infty, \infty)$: suppose the old value `x` is in the interval `[-100,100]`, then the new value of `x`, `(x*mratio + xp)/(mratio+1)`, lies in the interval $[-5100, 5100]$.

To obtain a more precise result, a heuristic of `Astrée` creates a separate partition for 51 possible values of the variable `mratio` and computes that `x`$\in$`[-100,100]`.
▲

2. Linear interpolation heuristic. Astrée uses the trace partitioning technique, when a sum of several expressions is computed, s.t.

- one summand is the `i`-th array element and
- another summand uses the variable `i`.

In particular, the partitioning starts from the point of the last non-trivial assignment of the variable `i`, i.e. of a conditional assignment or assignment in a loop.

Example 2.2.8. Consider the code in Fig. 2.9c. The code computes linear interpolation of a function with a polygonal path, which we plot in Fig. 2.9d. The polygonal path consists of 3 line segments, each defined with the arrays `tx, ty` and `tc`:

- `tx[i]` stores the x-coordinate of the right end point of `i`-th segment, for $0 \leq$ `i` <2 (`tx[2]` is not used and is initialised with `INT_MAX` which is the maximum value for an `int`)
- `ty[i]` stores the y-coordinate of the intersection point of `i`-th line segment with the y-axis, for $0 \leq$ `i` ≤ 2;
- `tc[i]` stores the slope of `i`-th line segment, for $0 \leq$ `i` ≤ 2.

The interpolation value `y` is computed as follows. In the loop in line 8 the code searches for a line segment, s.t. the condition `tx[i-1]`$<$`x`$\leq$`tx[i]` holds. Then in line 10 the code computes the y-coordinate of a point on the `i`-th line segment, given the x-coordinate of the point. The verification task is: given the interval in which the variable `x` lies, determine the interval of the variable `y`.

Without the linear interpolation heuristic, given the condition `x`$\in$`[-100,0]` at line 6, the interval analysis of ASTRÉE gets the following result:

- at line 10: `i`$\in$`[0,1]`;
- after line 10:
  - for `i=0`: `y`$\in$`[-1,-1]`;
  - for `i=1`: `y`$\in$`[-0.5+0.5*(-100),-0.5]=[-50.5,-0.5]`;

  therefore `y`$\in$`[-1,-1]`$\cup$`[-50.5,-0.5]=[-50.5,-0.5]`.

With the linear interpolation heuristic, ASTRÉE applies the tracing technique, since in line 10 a sum of an array element `ty[i]` and the expression `tc[i]*(x-tx[i])`, which uses the variable `i`, is computed. The heuristic creates a separate partition for traces in which the loop in line 8 terminates in `i` iterations. The interval analysis of ASTRÉE with the linear interpolation heuristic gets a more precise result than without the heuristic:

- at line 10: `i`$\in$`[0,1]`;
- after line 10:
  - for `i=0`: `x`$\in$`[-100,0]`$\cap$`(-`∞`,-1]=[-100,-1]` (from the conditions in lines 6 and 8) and `y`$\in$`[-1,-1]`;
  - for `i=1`: `x`$\in$`[-100,0]`$\cap$`(-1,1]=(-1,0]` (from the conditions in lines 6 and 8) and `y`$\in$`[-0.5+0.5*(-1),-0.5]=[-1,-0.5]`;

  therefore `y`$\in$`[-1,-1]`$\cup$`[-1,-0.5]=[-1,-0.5]`. ▲

3. Loop unrolling heuristic. Without a partitioning technique, `Astrée` creates one abstract state per control location in a loop. As a result, one abstract state corresponds to all loop iterations, which leads to imprecise results.

To gain precision, ASTRÉE implements a heuristic which creates separate partitions and computes abstract states for the first `n` loop iterations (the authors of [MR05] do not state how the value `n` is defined, we suppose it is a parameter of `Astrée`). Then an abstract state for all the following iterations is achieved using widening techniques.

Example 2.2.9. Some families of embedded programs, as those addressed by ASTRÉE, are executed in a loop. In the first one or several iterations of this loop some variables are initialised, and it is helpful to analyse the first iterations separately. ▲

For small loops ASTRÉE performs the complete unrolling of the loop.

Example 2.2.10. Consider the following program:

```
1 for (i=0; i<max; i++) {t[i] = i;}
```

which initialises the array `t` in a loop. Without unrolling of the loop, an interval analysis computes the result $\texttt{i} \in [0, \texttt{max}-1]$ and ASTRÉE infers that $\texttt{t[i]} \in [0, \texttt{max}-1]$. With a separate partition for each loop iteration, ASTRÉE infers a more precise result $\texttt{t[i]} \in [\texttt{i}, \texttt{i}]$. ▲

As an alternative to the loop unrolling heuristic, ASTRÉE implements a heuristic which starts from a single partition for the whole loop and iteratively refines the result by increasing the number of partitions.

CHAPTER 3

Definition and Computation of Variable Roles

In this chapter we study the notion of variable role. In particular, we envision a set of roles as a collection of patterns used by a programmer to compose a program. On the other hand, depending on the application in which roles are used, different roles can be of interest.

We identify *domain-independent* variable roles which capture frequently used patterns of variable use and occur in different types of programs, independently of a program domain. Our preliminary experiments [DVZ13] showed that the domain-independent set of roles capture the typical patterns of variable use. For the tasks *T2 (Portfolio solver for SV)* and *T3 (Systematic heuristics for SV)* the domain-independent set of roles, however, does not suffice. We therefore identify additional roles for each of the tasks. To summarise, we split our roles into three sets, namely the domain-independent set of roles and the roles for tasks *T2* and *T3* respectively.

We introduce variable roles step by step. In particular, in Section 3.1 we give an overview of roles. In Section 3.2 we define a framework for the specification and inference of variable roles, thereby solving Task *T1 (Role specification framework)*. In Section 3.3 we formally define the roles introduced in Section 3.1.

In Section 3.4 we extend the role definitions given in Section 3.3 from intra- to inter-procedural analyses. Finally, in Section 3.5 we discuss the implementation of the role inference algorithm.

3.1 Overview of Variable Roles

In this section we give an overview of roles:

- we discuss the criteria which we used when choosing roles;
- we give verbal definitions of roles;
- we illustrate the roles with C examples, in which we provide role annotations.

For each verbal role definition which we introduce in this section, we give a Datalog program in Section 3.3.

We structure this section as follows:

- In Section 3.1.1 we make an overview of the domain-independent roles.

  We give experimental results proving that the identified roles capture frequently used code patterns in Chapter 4. In particular, we show that the set of roles suffices to classify the benchmarks of the Software Verification Competition SV-COMP.

- In Section 3.1.2 we introduce the roles for the task *Portfolio solver for SV*.

  We prove the usefulness of this set of roles in Chapter 4 by building a portfolio solver based on these roles. We evaluate the portfolio solver in the setup of the competition SV-COMP.

- Finally, in Section 3.1.3 we make an overview of the roles for the task *Systematic heuristics for SV*.

  We use this set of roles in Chapter 5 to define heuristics for creating program-specific abstraction for software verification, specifically for the model checker Eldarica.

On completeness and soundness of role definitions. Before describing the roles, we note that our definitions of roles are neither complete (i.e. definitions do not cover all possible cases in which a role can be used), nor sound (i.e., given a role definition in the form of patterns, some variables satisfying the patterns, do not actually have the role). Nevertheless, our definitions are good for solving the challenges *C1 (Automatic choice of a verification tool)* and *C2 (Automatic choice of program-specific abstraction)*.

On soundness of role inference. In Section 3.6 we will comment on the soundness of the algorithm which solves the system of equations describing variable roles.

3.1.1 Domain-Independent Roles

Domain-independent roles capture the most common patterns used in imperative programming languages. In particular, the roles capture the concepts which a programmer acquires when learning an imperative programming language and which he later uses to write new programs. Examples of such concepts are *array index*, *counter*, *boolean*, etc.

To identify the domain-independent roles, we manually inspected the source code of the cBench benchmarks.[1] The benchmarks contain altogether approximately 5.2 KLOC of C

[1] `http://ctuning.org/wiki/index.php/CTools:CBench`. Accessed 23 January 2018.

C type	Role name	Informal definition
int	*array index*	used in an array subscript
	boolean	assigned and compared only to `0`, `1`, and the result of a relational operator
	constant-assigned	assigned only constant literals or *constant-assigned* variables
	enumeration	assigned and compared to only constant values, not used in relation operators other than (dis)-equality;
	branch condition	a variable used in a relation operator in a branch condition
	syntactic constant	not assigned in the program (a global or an unused variable, or a formal parameter to a external function)
	loop iterator	used in a loop condition, updated in the loop body
	loop bound	compared to a *loop iterator* in the loop condition
	counter	changed only in increment/decrement (by a constant value) statements, or assigned a constant value
	linear	assigned linear combinations of *linear* variables
	pointer offset	used in pointer arithmetic
	bitvector	used in a bitwise operation
	file descriptor	used in a library function which manipulates files
	character	used in a library function which manipulates characters, or assigned a character literal
	allocation size	passed to a memory allocation function
int, float	*arithmetic*	used in arithmetic or relational operators
	input	passed by address to an external function or assigned the return value of an external function
	unresolved	1) variable assigned the result of an external function call, or passed to an external function as a parameter 2) structure field or global variable

Table 3.1 Informal definition of domain-independent variable roles.

code. For a subset of benchmarks we read the code line by line, trying to understand what the code was doing. Then, having a good understanding of the code, we identified a set of roles such that at least one role is assigned to every program variable.

In total we identified a set of 14 domain-independent roles. We will now give verbal definitions of the roles and illustrate the roles with examples. We give a summary of the roles in Table 3.1.

Array index. A variable `x` has the role *array index* if one of the following conditions holds:

- `x` is used as an array subscript;;
- an array subscript is the expression `x-c` or `x+c`, where `c` is a constant number.

The condition for the moment is informal, we give a precise definition in Section 3.3.

Example 3.1.1. Consider the code in Fig. 3.1a from the LAME library[2] which converts audio to MP3 file format. We took the code from a cBench benchmark[3]

[2] `http://lame.sourceforge.net`. Accessed 23 January 2018.
[3] `consumer_lame/src/util.c`.

```
1 int bitrate_table[15];
2
3 int BitrateIndex(int bRate)
4 {
5   // index: array index
6   int index = 0;
7
8   // found: boolean, enumeration,
9   //   branch condition,
10  //   constant-assigned
11  int found = 0;
12
13  while (!found && index<15) {
14    if (bitrate_table[index]==bRate)
15      found = 1;
16    else
17      ++index;
18  }
19
20  if (found)
21    return index;
22  else
23    return -1;
24 }
```

(a) Roles *boolean*, *constant-assigned*, *enumeration*, *branch condition* and *array index*

```
1 // count: loop bound, arithmetic
2
3 void byte_reverse(
4   unsigned long *buffer, int count)
5 {
6   unsigned char ct[4];
7
8   unsigned char* cp =
9     (unsigned char*) buffer;
10
11  // i: loop iterator, counter,
12  //   linear, arithmetic
13  int i;
14
15  for (i = 0; i < count; ++i) {
16    ct[0] = cp[0];
17    ct[1] = cp[1];
18    ct[2] = cp[2];
19    ct[3] = cp[3];
20    cp[0] = ct[3];
21    cp[1] = ct[2];
22    cp[2] = ct[1];
23    cp[3] = ct[0];
24    cp += 4;
25  }
26 }
```

(b) Roles *loop iterator*, *loop bound*, *counter*, *linear* and *arithmetic*.

Figure 3.1 Code examples for domain-independent variable roles.

with minor modifications. Here and later in this chapter we annotate variables only with the roles illustrated with the current example, omitting remaining variable roles.

The function `BitrateIndex()` takes as a parameter `bRate` the bitrate characteristic of the audio stream and determines the index `index` of the value `bRate` in the table `bitrate_table`. Whether the entry is found is stored in the variable `found`. Initially the variable `found` is set to `0`. The code iteratively compares the table entries `bitrate_table[index]` with the bitrate `bRate` for the `index` values from `0` to `15`. If a table entry equals `bRrate`, the variable `found` is set to `1` and the loop terminates. The function `BitrateIndex()` returns the value `index` if the value `bRate` was found in the table and `-1` otherwise.

In this example, the variable `index` has the role *array index*, since `index` is used in array subscript for the array `bitrate_table`. ▲

Boolean. A variable `x` has the role *boolean* if the following two conditions hold:

1. All the expressions which are assigned to `x` or to which `x` is compared are one of the following:
   - constant values `1` and `0`;
   - the result of a relational operator, e.g. <, ==, etc.;
   - a variable which has the role *boolean*;
2. The variable `x` is not used in operations other than equality, dis-equality and logical operators, e.g. logical AND, etc.

Note that since there is no boolean data type in C language, the values `1` and `0` are used to represent the values `true` and `false` respectively.

Example 3.1.2. In the code in Fig. 3.1a the variable `found` has the role *boolean*, since `found` is assigned only the constant values `0` and `1` and compared for equality to `0` in line 20. ▲

Constant-assigned. A variable `x` is *constant-assigned* if it is assigned only constant values and variables which have the role *constant-assigned*.

Example 3.1.3. In the code in Fig. 3.1a the variable `found` has the role *constant-assigned*. Note that in general *boolean* variable is not necessarily a *constant-assigned*. ▲

Enumeration. A variable `x` has the role *enumeration* if the following conditions hold:

1. The values which are assigned to `x` and to which `x` is compared are one of the following:
   - a constant value;
   - a variable which has the role *enumeration*;
2. The variable `x` is not used in relation operators other than equality and dis-equality.

Note that the role *enumeration* is a special case of the role *constant-assigned*.

Example 3.1.4. In the code in Fig. 3.1a the variable `found` is *enumeration*. ▲

Branch condition. A variable `x` is *branch condition* if `x` is occurs in a relation operator in a condition of an `if` statement.

Example 3.1.5. In the code in Fig. 3.1a the variable `found` is a *branch condition*, since `found` is implicitly compared to `0` in the condition of the `if` statement in the line 20. ▲

Syntactic constant. A variable `x` has the role *syntactic constant* if the following conditions hold:

- `x` is not assigned in any statement;
- `x` is not accessed indirectly.

Example 3.1.6. In the example in Fig. 3.1a the variable `bRate` has the role *syntactic constant*, since `bRate` is not modified anyhow. ▲

Loop iterator. A variable `x` has the role *loop iterator* of a loop L if the following two conditions hold:

1. `x` is updated inside the body of a loop L;
2. `x` is used in the condition of the loop L in a relational operator.

Note that the role *loop iterator* is different from the previous roles since it has a *parameter* L. We need this parameter first, because a *loop iterator* is used inside a loop differently than outside the loop. Second, we use this parameter to later define the role *loop bound*.

Example 3.1.7. Consider the code in Fig. 3.1b from the SHA cryptographic algorithm.[4] We took the code from a cBench benchmark[5] with minor modifications.

The function `byte_reverse()` takes two arguments: an array `buffer` of integer numbers and the length `count` of the array. The function changes the encoding of the elements of `buffer` from big endian to little endian or vice versa, i.e. reverses the order of bytes in every element of `buffer`. In every loop iteration the pointer `cp` points to the `i`-th element of `buffer`. In lines 16–19 the four bytes of the `i`-th element of `buffer` are stored in the array `ct`, and in lines 20–23 the four bytes are copied back to the `i`-th element of `buffer` in reverse order.

In this example, the variable `i` has the role *loop iterator*, since `i` is updated in the loop body and used in the loop condition in the less-than operator. ▲

Loop bound. A variable `x` has the role *loop bound* of loop L if `x` is compared in the condition of loop L to the *loop iterator* of loop L using a relational operator.

Example 3.1.8. In the code in Fig. 3.1b the variable *count* has the role *loop bound*, since `count` is compared to the *loop iterator* `i` in the condition of the loop. ▲

Counter. A variable `x` has the role *counter* is `x` is updated only in the following statements:

- `x` is assigned a constant value;
- `x` is incremented or decremented by a constant value.

[4] `http://csrc.nist.gov/groups/ST/toolkit/secure_hashing.html`. Accessed 23 January 2018.

[5] `security_sha/src/sha.c`.

```
1 char hasSuffix(char* s,
2   const char* suffix)
3 {
4   // ns, nx: offset, input,
5   int ns = strlen(s);
6   int nx = strlen(suffix);
7
8   if (ns < nx)
9     return 0;
10
11  if (!strcmp(s+ns-nx, suffix))
12    return 1;
13
14  return 0;
15 }
```

(a) Roles *pointer offset* and *input*.

Figure 3.2 Code examples for domain-independent variable roles (cont.).

Example 3.1.9. In the code in Fig. 3.1b the variable `i` has the role *counter*, since the statements in which `i` is updated, satisfy the definition: `i` is assigned `0` in line 15 and is incremented by `1` in the same line.

Linear. A variable `x` has the role *linear* if `x` is assigned only expressions $c_0 + \sum_i^n c_i * v_i$, where c_i are constant values and v_i are *linear* variables.

Example 3.1.10. In the code in Fig. 3.1b the variable `x` has the role *linear*, since it is assigned two linear expressions `0` and `x+1`. ▲

Arithmetic. A variable `x` has the role *arithmetic* if `x` is used in an arithmetic operation, such as $+$, $*$, $\geq$, etc.

Example 3.1.11. In the code in Fig. 3.1b the variables `i` has the role arithmetic, since `i` is used in an addition operator, and similarly `count` has the role *arithmetic*, since `count` is used in the operator $<$. ▲

Pointer offset. A variable `x` is *pointer offset* if `x` is added to or subtracted from a pointer.

Example 3.1.12. Consider the code in Fig. 3.2a from the `bzip2` compression algorithm. The code is taken with minor changes from a cBench benchmark.[6]

The function `hasSuffix()` determines whether string `s` ends with substring `suffix`. In line 8 a check is done whether the string `suffix` is longer than the

[6] `bzip2d/src/bzip2.c`.

string s, using library function strlen(), which calculates the length of a string. If suffix is shorter then s, then the last nx symbols of s are compared with suffix (in line 11). The library function strcmp() returns 0 in case the strings are equal.

In this example, the variables ns and nx are *pointer offsets*, since they are used in the pointer arithmetic expression s+ns−nx. ▲

Input. A variable x has the role *input* if x is assigned the result of a call to an external function.

Example 3.1.13. In the code in Figure 3.2a, the variables ns and nx have the role *input*, since both are assigned the result of a call to the library function strlen(). ▲

Bitvector. A variable x is a bitvector if x if x is used in a bitwise operator, e.g. bit AND, bit NOT, etc.

Example 3.1.14. Recall the example in Fig. 1.1a on page 7. In this example the variable x has the role *bitvector*, since x is used in bit AND operator. ▲

File descriptor. A variable x has the role *file descriptor* if at least one of the following cases holds:

1. x is assigned the result of a call to a standard library function which returns a file descriptor, e.g. open(), dup(), etc.
2. x is passed as a parameter to a standard library function which manipulates files, e.g. read(), lseek(), etc.

Example 3.1.15. Recall the example in Fig. 1.1b on page 7. In this example the variable fd has the role *file descriptor*, since fd is assigned the result of a call to the library function open. ▲

Character. A variable x has the role *character* if at least one of the following conditions hold:

1. x is assigned a character symbol;
2. x is assigned the result of a call to a standard library function which returns a character, e.g. getc(), etc.;
3. x is passed as a parameter to a standard library function which manipulates characters, e.g. putc(), isupper(), etc.

Example 3.1.16. In the example in Fig. 1.1b the variable c has the role *character*, since c is passed as a parameter to the function isdigit(). ▲

```
1 #define EINTR 4
2 extern int nondet_int();
3
4 int mutex_lock_interruptible(int ldv_mutex)
5 {
6   int nondetermined = nondet_int();
7   assert(ldv_mutex == 1);
8
9   if (nondetermined)
10  {
11     ldv_mutex = 2;
12     return 0;
13  }
14  else
15  {
16    return -EINTR;
17  }
18 }
```

Figure 3.3 Code examples for domain-independent variable roles (cont.): special role *unresolved*.

Allocation size. A variable `x` has the role *allocation size* if `x` is passed to a standard library function which allocates memory. We give an example for this role in Section 3.1.3 in Example 3.1.23 on page 53.

Unresolved. A variable `x` has the role *unresolved* if `x` is assigned the result of an external function call or passed to an external function as an argument.

Example 3.1.17. Consider the code in Fig. 3.3, taken with minor modifications from an SV-COMP'14 benchmark.[7] The code models the function `mutex_lock_interruptible()`, the parameter `ldv_mutex` of which holds the state of a mutex. If no signal arrives while waiting for the lock, the function acquires the lock and returns `0` (lines 11–12), otherwise the function returns the value `-EINTR` (line 16).

Whether a signal has come is modelled with a non-deterministic function `nondet_int()` assigned to the variable `nondetermined` (line 6). The variable `nondetermined` has therefore the role *unresolved*. Otherwise, `nondetermined` is used as a *boolean*, and the role *unresolved* expresses our assumption that the external function `nondet_int()` returns a *boolean*. ▲

We introduce the role *unresolved* for the following purpose: our method computes an *approximation* of the set of variables for the roles defined *with negation*. The exact

[7] `ldv-linux-3.4-simple/32_1_cilled_true_ok_nondet_linux-3.4-32_1-drivers--pci--hotplug--cpcihp_generic.ko-ldv_main0_sequence_infinite_withcheck_stateful.cil.out.c`.

C type	Role name	Informal definition
`int`	*thread descriptor*	passed as first argument to the `pthread` library function
	integral	the type is integral and can be cast to `int`
`float`	*floating point*	the type is floating point and can be cast to `float`
`int`, `float`	*scalar*	the type is scalar (integral or floating point)
`int*`, `float*`	*pointer to scalar*	the type is a pointer to scalar
*struct_type**	*pointer to structure*	the type is a pointer to a structure
	pointer to non-flat structure	the type is a pointer to a structure with a pointer field
	linked list	the type is a pointer to a recursively defined structure
	multi-linked list	the type is a pointer to a recursively defined structure with more than one pointer, e.g. doubly linked lists
*any_type**	*pointer*	the type is a pointer
	dynamic memory pointer	assigned the result of a memory allocation function
	recursive function result	a return value of a recursively defined function

Table 3.2 Informal definition of additional roles for the portfolio solver. Type *struct_type* stands for a C structure, *any_type* for an arbitrary C type.

solution for a role r defined with negation is computed as a set difference of the transitive closure TC of the relation `r` and `unresolved`: $TC(\texttt{r}) \setminus TC(unresolved)$.

3.1.2 Roles for a Portfolio Solver for Software Verification

To identify roles for the task *Portfolio solver for SV*, we started from the domain-independent set of roles. We observed, however, that these roles are not sufficient to build a portfolio solver which is competitive enough to beat the participants of the competition. Therefore, we extended the set of roles with several new roles capturing the code structures which are challenging for software verification tools.

To this end we analysed the results of the competition SV-COMP'14, in particular the following cases:

- a tool was unsound,
- a tool did not give an answer or
- a tool crashed.

We observed that particular code structures constitute strong or weak points of verification tools depending on whether the tools are optimised to handle these structures. Using this information, we extended the domain-independent set with 12 additional roles which we list in Table 3.2.

Below we give a summary of the identified challenging code structures and the additional roles which capture these structures. We illustrate the roles with examples.

Thread descriptor. A small number of the tools participating in the competition such as CSeq-Lazy, CSeq-MU and CBMC are narrowly optimised for concurrent

```
1 #include <pthread.h>
2 extern int nondet_int();
3 extern void* thr(void* arg);
4
5 void main() {
6   // t: thread descriptor
7   pthread_t t;
8
9   // len: scalar, integral scalar
10  int len = nondet_int();
11  assume(len>0);
12
13  // data: pointer, scalar pointer,
14  //   dynamic memory pointer
15  int* data = malloc(
16    sizeof(int)*len);
17
18  while(1) {
19    pthread_create(&t,0,thr,data);
20  }
21 }
```

(a) Roles *thread descriptor*, *integral*, *dynamic memory pointer*, *pointer to scalar* and *pointer*

```
1 // skip list node with two
2 // next pointers
3 struct sl_item {
4     struct sl_item *n1, *n2;
5 };
6
7 // skip list
8 struct sl {
9     struct sl_item *head, *tail;
10 };
11
12 // n1, n2, head, tail:
13 // pointer,
14 // pointer to structure,
15 // pointer to non-flat structure,
16 // linked list,
17 // multi-linked list
18
19 // skip_list: pointer,
20 // pointer to structure,
21 // pointer to non-flat structure,
22 struct sl* skip_list;
```

(b) Roles *multi-linked list*, *linked list*, *pointer to non-flat structure*, *pointer to structure* and *pointer*

Figure 3.4 Code examples for variable roles for portfolio solver.

code, while most other participants of the SV-COMP competition cannot handle concurrent code.

To capture multi-threaded code patterns we introduce a role *thread descriptor*. A variable `x` has the role *thread descriptor* if `x` is passed as a parameter to a function of the `pthread` library, which manipulates threads.

Example 3.1.18. Consider the code in Fig. 3.4a, taken from an SV-COMP'15 benchmark[8] with minor modifications. The program allocates memory for the array `data` in lines 15–16. In the loop in lines 18–20 the program starts new threads which execute the function `thr` with the argument `data`.

In this example, the variable `t` has the role *thread descriptor*, since `t` is passed to the function `pthread_create()` as the first parameter, in which the descriptor of a newly created thread is returned. ▲

Integral. A number of the participants, such as BLAST, FrankenBit, UFO and others, are optimised to handle integral scalar data, rather than pointers.

[8] `pthread-lit/sssc12_true-unreach-call.c`.

To capture code patterns with scalar data we introduce the role *integral*. A variable `x` has the role *integral* if the datatype of `x` is scalar integer, e.g. `int`, `long`, `char`, etc.

Example 3.1.19. In the code in Fig. 3.4a the variable `len` has the role *integral*, because the type of `len` is `int`. ▲

Pointer. In contrast to the tools which are optimised to handle integral data structures, a number of competition participants, such as Predator, CBMC and Symbiotic are optimised to reason about pointers.

To capture code patterns with pointer data we introduce the role *pointer*. A variable `x` has the role *pointer* if `x` is of a pointer data type.

Example 3.1.20. In the code in Fig. 3.4a the variable `data` has the role *pointer*. ▲

Pointer to scalar and *pointer to structure.* A number of participants which handle pointer data are optimised to arrays, e.g. BLAST and FrankenBit, while others implement the *shape analysis* to reason about dynamic data structures. To capture the difference between the two types of code patterns, we introduce the roles *pointer to scalar* and *pointer to structure*.

A variable `x` has the role *pointer to scalar* if the data type of `x` is a pointer to a scalar data type. If the data type of a variable `x` is a pointer to a structure, then `x` has the role *pointer to structure*.

Example 3.1.21. In the example in Fig. 3.4a, the variable `data` has the roles *pointer to scalar*. ▲

Example 3.1.22. Consider the code in Fig. 3.4b from an SV-COMP'15 benchmark.[9]

The code in Fig. 3.4b defines the data type `sl` for a skip list, which is a data structure for fast search within an ordered sequence of elements. The structure `sl` is defined in lines 8–10 and contains the fields `head` and `tail` which point to the beginning and to the end node respectively of the skip list. A variable `skip_list` of the type `sl` is defined in line 22.

For the nodes of the skip list the type `sl_item` is defined at lines 3–5. Each list node contains pointers to two other nodes `n1` and `n2` of the skip list.

In this example, the variable `skip_list` and the fields `n1`, `n2`, `head` and `tail` have the role *pointer to structure*. ▲

Dynamic memory pointer. In order to differentiate between statically and dynamically allocated data, we introduce the role *dynamic memory pointer*. A variable `x` has the role *dynamic memory pointer* if at least one of the following cases holds:

[9] `memsafety-ext/skiplist_2lvl_true-valid-memsafety.c`.

1. `x` is assigned the result of a call to a standard library function which returns a pointer to dynamically allocated memory, e.g. `malloc()`, `mmap()`, etc.;
2. `x` is passed as a parameter to a function which manipulates dynamically allocated memory, e.g. `free()`, `mlock()`, etc.

Example 3.1.23. In the code in Fig. 3.4a the variable `data` has the role *dynamic memory pointer*, since `data` is assigned the result of a call to the function `malloc()`. The variable `len`, which is multiplied by the size of an integer and passed to the function `malloc()` as a parameter, has the role *allocation size* (see the definition of the role *allocation size* on page 49). ▲

Pointer to non-flat structure, *linked list* and *multi-linked list.* From pointers to structure data types we separate two following cases:

- A variable `x` has the role *pointer to non-flat structure* if the type of `x` is a pointer to a structure which contains a pointer field;
- A variable `x` has the role *linked list* if the type of `x` is a pointer to a recursively defined structure.
- A variable `x` has the role *multi-linked list* if the type of `x` is a pointer to a recursively defined structure `s` with more than one pointers to `s`.

Example 3.1.24. In the code in Fig. 3.4b the variable `skip_list` and the fields `head`, `tail`, `n1` and `n2` have the role *pointer to non-flat structure*, since the structure `sl` contains pointers.

The fields `n1` and `n2` have the role *linked list*, since the structure `sl_item` contains pointers to structures of the same type `sl_item`.

Finally, since the structure `sl_item` contains two fields pointing to the structure `sl_item`, then the fields `n1` and `n2` also have the role *multi-linked list.* ▲

Recursive function result. Handling recursive functions is not supported by some competition participants such as BLAST, FrankenBit, Threader etc.

To capture recursion we introduce the role *recursive function result.* A variable `x` has the role *recursive function result* if at least one of the following conditions hold:

1. `x` is returned by a recursive function;
2. `x` is assigned the result of a call to a recursive function;

where a function `f()` is recursive if the definition of `f()` contains a call to `f()`.

Example 3.1.25. Consider the code in Fig. 3.5a from an SV-COMP'14 benchmark.[10] The code implements the Ackermann function

$$A(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ A(m-1,1) & \text{if } m > 0 \text{ and } n = 0 \\ A(m-1, A(m,n-1)) & \text{if } m > 0 \text{ and } n > 0. \end{cases}$$

[10] `recursive/Ackermann01_true.c`.

```
1 extern int nondet();
2
3 int ackermann(int m, int n) {
4   if (m==0) return n+1;
5   if (n==0) return ackermann(m-1,1);
6   return ackermann(m-1, ackermann(m,n-1));
7 }
8
9 int main() {
10  int m = nondet(), n = nondet();
11  int result = ackermann(m,n);
12 }
```

(a) Role *recursive function result*

```
1 float fabs(float n)
2 {
3   float f;
4
5   if (n >= 0) f = n;
6   else f = -n;
7   return f;
8 }
```

(b) Role *floating point*

Figure 3.5 Code examples for variable roles for portfolio solver (cont.).

In this example, the variable `result` at line 11 has the role *recursive function result*, since `result` is assigned the result of a call to the recursively defined function `ackermann()`. ▲

Floating point. Floating-point values are modelled by few competition participants, such as CBMC, CPAchecker and ESBMC. Most other participants, however, do not model floating point data.

To capture floating-point values we introduce a role *floating point*. A variable `x` has the role *floating point* if `x` is of a floating point type, i.e. `float`, `double` or `long double`.

Example 3.1.26. Consider the code in Fig. 3.5b taken from an SV-COMP'14 benchmark[11] with minor modifications. The code implements the function `fabs()` which returns an absolute value of its parameter.

In this example, the variables `n` and `f` have the role *floating point*, since the type of these variables is `float`. ▲

3.1.3 Roles for Heuristics in Predicate Abstraction

In the task *Systematic heuristics for SV* we use variable roles to define heuristics for software verification to find a suitable abstraction. Specifically, we do a case study on the model checker Eldarica. Eldarica is based on predicate abstraction and Craig interpolation and has two main parameters controlling the analysis process: initial predicates for predicate abstraction and templates which guide Craig interpolation [LRS16]. Both parameters can be provided in the form of source code annotations. We give examples of the annotations in Section 5.1.4.

[11] `loops/lu.cmp_true.i`.

C type	Role name	Parameter	Informal definition
int	*dynamic enumeration*	v_i	the transitive closure of *input* variables (can be assigned only *input* variables v_i and constant values)
	extremum	–	assigned the maximum or the minimum of two or more values
	local counter	L	incremented or decremented in a loop L by arbitrary expression
	parity	c	used in the remainder operator with a constant value c or incremented in a loop by a constant value c not equal to `1`
	assertion parameter	*expr*	used in the condition *expr* of an assume or assertion statement, or in a branch condition *expr* on the path to an assume/assertion statement

Table 3.3 Informal definition of variable roles for heuristics in predicate abstraction.

To derive role-based heuristics for Eldarica, we analysed appr. 30 benchmarks which could not be solved by Eldarica within the time limit of 15 minutes. In particular, we analysed the SV-COMP'16 benchmarks from categories "Integers and Control Flow" and "Loops" and loop invariant generation benchmarks.

To find the roles capturing the patterns which are challenging for Eldarica, we took the following steps:

1. We manually inspected the code of the benchmarks.
2. For each benchmark we manually found suitable predicates and templates P_{man}.
3. From the logs of Eldarica, we extracted the predicates and templates P_{Eld} which Eldarica generated in the process of verification.
4. We then computed the difference $P_{miss} = P_{man} \setminus P_{Eld}$ of the two sets to obtain the *missing* predicates.
5. Finally, we annotated the benchmarks with the predicates and templates P_{miss} and checked that Eldarica verifies the annotated benchmarks within the time limit.

From P_{miss} we derived 5 new variable roles which capture the specific code patterns in which the annotated variables are used.

Role parameters. Differently from the domain-independent roles and the roles for the *Portfolio solver for SV*, the roles for *Systematic heuristics for SV* collect pieces of information about the program like in a static analysis, which can be used then for program verification. We call these pieces of information *parameters* of a role.

We give a summary of the roles in Table 3.3. We will now give verbal definitions of the roles and illustrate the roles with examples.

Dynamic enumeration. A variable `x` has the role *dynamic enumeration* with parameter $\texttt{v}_i$ if `x` is assigned only constant values and *input* variables, and input variable $\texttt{v}_i$ is assigned to `x` (specifically, our algorithm computes the transitive closure of all values assigned to `x`). A *dynamic enumeration* can be seen as a run-time analogue

of a compile-time enumeration. Note that there can be multiple parameters for one *dynamic enumeration* variable.

Example 3.1.27. The code in Fig. 3.6a initializes variables `max1`, `max2` and `max3` with the values `id1`, `id2` and `id3` respectively, which are in turn initialized non-deterministically. The `assume()` statement at lines 13–14 puts a restriction that control reaches line 16 only if the condition `(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 at lines 22–24 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 $\texttt{i}=\arg\max_j \{\texttt{id}_j\}$.

In this program, the variables `id1`, `id2` and `id3` have the role *input*, since these variables are assigned the result of an external function call `nondet_char()`. Next, the variables `max1`, `max2` and `max3` have the role *dynamic enumeration*, each with parameters `id1`, `id2` and `id3`, since `max1`, `max2` and `max3` are assigned only constant values, and the variables `id1`, `id2`, `id3`.

Note that the variables `max1`, `max2` and `max3` are also assigned in the statements at lines 17–19, i.e. `max1=max3` etc. We implicitly add to the definition of every role r defined with negation a condition that a variable of role r can be assigned variables of same role; we make this condition explicit in Section 3.3. ▲

Heuristic for *dynamic enumeration*. We use the role *dynamic enumeration* in the following heuristic: for each variable `x` which has the role *dynamic enumeration* with a parameter $\texttt{v}_i$, our algorithm generates an equality predicate: $\texttt{x}==\texttt{v}_i$. We will define the heuristic formally in Section 5.2.

Example 3.1.28. For the program in Fig. 3.6a, our algorithm generates the predicates `max1==id1`, `max1==id2`, `max1==id3` for the *dynamic enumeration* `max1`, predicates `max2==id1`, `max2==id2`, `max2==id3` for `max2` and predicates `max3==id1`, `max3==id2` and `max3==id3` for `max3`. Eldarica uses these predicates in the predicate abstraction technique in order to prove that exactly one condition at lines 22–24 evaluates to `true` and `cnt` is incremented by exactly one.

To prove safe the program in Fig. 3.6a without annotations, Eldarica finds the 9 predicates during abstraction refinement, and the whole verification process takes 27 CEGAR iterations and 19 sec. However, for 88 out of 108 original programs from SV-COMP with this pattern in category "Integers and Control Flow", of which the

```
1 extern char nondet_char();
2 void main() {
3   // id1, id2, id3: input
4   int id1 = nondet_char();
5   int id2 = nondet_char();
6   int id3 = nondet_char();
7
8   // max1, max2, max3: dynamic
9   //  enumeration, extremum
10  int max1=id1, max2=id2, max3=id3;
11  int i=0, cnt=0;
12
13  assume(id1!=id2 && id1!=id3 &&
14    id2!=id3);
15
16  while (1) {
17    if (max3 > max1) max1 = max3;
18    if (max1 > max2) max2 = max1;
19    if (max2 > max3) max3 = max2;
20
21    if (i == 1) {
22      if (max1 == id1) cnt++;
23      if (max2 == id2) cnt++;
24      if (max3 == id3) cnt++;
25    }
26
27    if (i>=1) assert(cnt==1);
28    i++;
29  }
30 }
```

(a) Roles *dynamic enumeration*, *extremum* and *assertion parameter*

```
1 int nondet_int();
2 void main() {
3   int buflen = nondet_int();
4   int inlen = nondet_int();
5   assume(buflen > 1);
6   assume(inlen > buflen);
7
8   // buf: local counters
9   int buf = 0;
10  // buflim: loop bound
11  int buflim = buflen-2;
12
13  while (nondet_int()) {
14    if (buf == buflim) break;
15    assert(buf<buflen);
16    buf++;
17    assert(buf<inlen);
18  }
19 }
```

(b) Role *local counter*

```
1 extern int nondet_int();
2 void main() {
3   // i: parity
4   int i;
5   for (i=0; i<1000000; i+=2);
6   assert(i == 1000000);
7 }
```

(c) Role *parity*

Figure 3.6 Code examples for variable roles for software verification.

code in Fig. 3.6 is a simplified form[12], 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 without the heuristic for *dynamic enumeration* is time consuming. ▲

Extremum. A variable `x` has the role *extremum* if `x` is used in the pattern `if (y ∘ expr) x=y`, where $\circ \in \{<, >\}$ and `expr` is an arbitrary expression.

Example 3.1.29. In the program in Fig. 3.6a, the variables `max1`, `max2` and

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

`max3` have the role *extremum*, because these variables are used in the conditional assignments at lines 17, 18 and 19 respectively which match the pattern `if (y ∘ expr) x=y`, i.e. `if (max3>max1) max1=max3`, etc. ▲

Heuristic for *extremum* and *dynamic enumeration*. For every variable `x` which is both a *dynamic enumeration* and an *extremum*, our algorithm generates for every pair `y` and `z` of parameters of the role *dynamic enumeration* two inequality predicates `y<z` and `y>z`.

Example 3.1.30. For the code in Fig. 3.6a, our algorithm generates 6 predicates `id1<id2`, `id1>id2`, `id1<id3`, `id1>id3`, `id2<id3` and `id2>id3`. Eldarica uses these predicates in the predicate abstraction technique to precisely evaluate the conditions at lines 17–19.

Eldarica proves the program in Fig. 3.6a annotated with the 6 predicates and 9 predicates from Example 3.1.28, i.e. 15 predicates generated by our algorithm altogether, in 8 sec and 0 CEGAR iterations. To prove 53 programs from SV-COMP'16 with this pattern, annotated analogously by our algorithm, it takes Eldarica from 21 to 858 sec (and from 0 to 4 CEGAR iterations). For the remaining 55 benchmarks with this pattern from SV-COMP'16 the number of abstract states becomes too large for Eldarica to be checked within the time limit. ▲

Local counter. A variable `x` has the role *local counter* in loop L if `x` is assigned in the body of loop L in a statement `x=x+expr`, where `expr` is an arbitrary expression.

Example 3.1.31. Consider the code in Fig. 3.6b. We take the code from an SV-COMP'16 benchmark[13] in a simplified form.

The code increments in a loop the buffer index `buf` by `1` until `buflim` is reached, where `buflim` stores the size of the writeable part of the buffer. Alternatively, the loop can terminate on a non-deterministic event if the function `nondet_int()`, called in the loop condition at line 13, returns `0`. The assertion at line 15 checks that the buffer index `buf` does not exceed the buffer size `buflen`, and the assertion at line 17 checks that `buf` does not exceed the size of the input stream `inlen`.

Since the variable `buf` is incremented in the loop body by `1`, then `buf` is *local counter* (for the only loop at lines 13–18). ▲

Heuristics for *local counter*. We use the role *local counter* in two following heuristics:

1) *Inequality between loop counter and loop bound.* If the following conditions hold:
   a) a variable `x` has the roles *loop iterator* in loop L and *local counter* in loop L,

[13] `loop-invgen/sendmail-close-angle_true-unreach-call.i`.

b) a variable `b` has the role *loop bound* in loop L,

c) `x` is compared to `b` in a (dis-)equality `x==b` or `x!=b`,

then our algorithm generates the predicates `x<=b`, `x>=b`, `x<b` and `x>b`.

To be precise, for efficiency reasons in the first heuristic our algorithm generates only two of the four predicates (either `x<=b` and `x<b`, or `x>=b` and `x>b`). We omit the details for the sake of simplicity.

Example 3.1.32. Recall from Ex. 3.1.31 that the variable `buf` in the program in Fig. 3.6b has the role *local counter*. In addition, `buf` is *loop iterator*, since `buf` is assigned in the loop body and is compared in the expression `buf==buflim` at line 14, which our algorithm considers as a loop condition (we give the precise definition of *local counter* in Section 3.3.3).

Next, since the variable `buflim` is compared in the loop condition `buf==buflim` at line 14 to the *loop iterator* `buf`, then `buflim` is a *loop bound*. Using these roles, our algorithm applies the first heuristic for *local counter* and generates the predicates `buf<buflim` and `buf<=buflim`. ▲

2) *Splitting assignments to loop bound.* If the following conditions hold:

a) a variable `b` has the role *loop bound* in loop L,

b) `b` is modified in an assignment statement `b=expr`,

c) there is a variable `x` which has the role *loop iterator* and *local counter* in same loop L,

then our algorithm generates predicates `b>=expr` and `b<=expr`. The condition 2c) restricts the number of the loops to which the heuristic is applied.

Similarly to the first heuristic for *local counter*, for efficiency reasons our algorithm generates in the second heuristic for *local counter* only one of the two predicates (either `b>=expr`, or `b<=expr`).

Example 3.1.33. For the program in Fig. 3.6b, since the variable `buflim` has the role *loop bound* and `buflim` is assigned in the statement `buflim=buflen-2` at line 11, then our algorithm applies the heuristic and generates the predicate `buflim<=buflen-2`.

Without our heuristics, Eldarica can not check the program within the time limit: Eldarica generates a sequence of predicates `buf<=1`, `buf<=2`, `buf<=3`, etc, for ever growing number of loop unrollings. However, when we annotate the code with the predicates `buf<buflim`, `buf<=buflim` computed by the heuristic *Splitting assignments to loop bound*, the predicate `buflim<=buflen-2` computed by the heuristic *Inequality between loop counter and loop bound* (see Ex. 3.1.32) and the predicate `inlen>buflen`, which we will explain later in this section in Ex. 3.1.35, Eldarica proves the program safe.

A possible way to obtain the predicate `buflim<=buflen-2` is to rewrite each assignment, equality and dis-equality in the transition relation of a program

with two inequalities. For example, for code in Fig. 3.6b the assignment statement at line 11 will be represented by the conjunction of the inequalities `buflim>=buflen-2 ∧ buflim<=buflen-2`, and the inequality in the loop condition at line 14 will be represented by the disjunction of the inequalities `buf<buflim ∨ buf>buflim`. Eldarica implements a configuration which performs such re-writing, and the configuration proves the program safe in 6 sec and 5 CEGAR iteration. However, the re-writing can have a negative effect on the performance, because after the re-writing the system of constraints used to encode the transition relation has the size in the worst case exponential in the size of the system. Therefore, the second heuristic for *local counter* is needed to choose a subset of the transitions to be split. ▲

Assertion parameter. A variable `x` has the role *assertion parameter* with parameter `Expr` if `x` is used in the condition `Cond` of an `assert` or `assume` statement, and `Expr` is the smallest boolean-valued sub-expression of `Cond`, i.e. `Expr` which does not contain logical operators.

Example 3.1.34. In the program in Fig. 3.6b, the following variables have role *assertion parameter*:

- the variable `buflen` – with parameter `buflen>1` (statement `assume(buflen>1)` at line 5),
- the variables `inlen` and `buflen` – with parameter `inlen>buflen` (statement `assume(inlen>buflen)` at line 6),
- the variables `buf` and `buflen` – with parameter `buf<buflen` (statement `assert(buf<buflen)` at line 15) and
- the variables `buf` and `inlen` – with parameter `buf<inlen` (statement `assert(buf<inlen)` at line 17). ▲

Heuristic for *assertion parameter*. For every *assertion parameter* used in assertion condition `Cond`, our algorithm generates the predicate `Cond`.

Example 3.1.35. For the program in Fig. 3.6b, our algorithm generates the predicates `buflen>1`, `inlen>buflen`, `buf<buflen` and `buf<inlen`. ▲

Parity. A variable `x` has the role *parity* with parameter `n` if at least one of the following cases holds:

- `x` is incremented in a loop by a constant value `n`, s.t. `n!=1`;
- `x` is used in the expression `x%n`.

Example 3.1.36. Consider the code in Fig. 3.6c from an SV-COMP'16 benchmark.[14] The code iteratively increments the variable `i` by `2` starting from `0` until `i` reaches `1000000`. The assertion condition at line 6 checks that `i` equals `1000000`.

In this program, the variable `i` has the role *parity* with parameter `2`. ▲

Heuristic for *parity*. For each *parity* variable with parameter `n` our algorithm generates the predicate template `x%n`.

Example 3.1.37. For the program in Fig. 3.6c, our algorithm generates the template `x%2`.

Without this heuristic, Eldarica can not check the program within the time limit: similarly to Ex. 3.1.33, Eldarica generates a sequence of predicates `i==0`, `i==2`, `i==4`, etc. With the template annotation `i%2`, Eldarica proves the code safe in 2 CEGAR iterations and 1 sec. ▲

3.2 Framework for the Specification and Inference of Roles

In this section we define a framework to formally specify and compute variable roles. We define each variable role with a *data-flow analysis* [NNH99].

In particular, we formulate an insensitive data-flow analysis in Datalog. We specify the control-flow graph of a program as a database of facts, and roles as logic queries on this database. We then obtain the result of the analysis using the standard fixed-point semantics of Datalog. In this way, choosing logic programming as a specification formalism has two advantages: first, its notation is well known, and second, we can use off-the-shelf logic engines for the computation of roles.

We give preliminaries on Datalog in Section 3.2.1, describe our algorithm for the translation of C code to a logic program in Section 3.2.2 and give the formal specification of roles in Section 3.3.

3.2.1 Preliminaries on Datalog

A datalog program is constructed from the following items (listed in the order from simplest to most complex):

Term. A *term* t takes values in some domain D, e.g. $\mathbb{N}$, and is of the form:
- an integer; or
- a variable; or
- $t_1 \circ t_2$; or
- $f(t_1, \ldots, t_k)$,

[14] `loop-new/count_by_1_true-unreach-call.i`.

where t_j are terms, $\circ \in \{+, -, *, /\}$ is an arithmetic operator, and f is a function symbol (only interpreted functions are allowed).

Atom. An *atom* takes boolean values *true* and $false$ and is of the form:
- $p(t_1, \ldots, t_m)$, or
- $t_1 \circ t_2$, or
- $t_0 = f(t_1, \ldots, t_k)$ (the symbol = should not be confused with the comparison operator in the previous case, see the explanation below),

where p is a predicate symbol, f is a function symbol, t_j are terms and $\circ \in \{<, \leq, >, \geq, ==, != \}$ is a comparison operator. Atom $t_0 = f(t_1, \ldots, t_k)$ always evaluates to *true* and assigns to term t_0 the result of function $f(t_1, \ldots, t_k)$. Predicate and function symbols start with a small letter, and variables start with a capital letter.

Literal. A *literal* is of the form A or *not* A for an atom A, where the connective *not* corresponds to default negation.

Rule. A *rule* in Datalog is of the form $A_0 :- L_1, \ldots, L_n$. The *head* of a rule A_0 is an atom. The *body* of a rule $\{L_i\}$ is a set of literals. 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*.

In this thesis we only deal with programs with *stratified negation*. A program has stratified negation if the program contains no cyclic dependencies with negation. Formally, stratified negation is implicitly defined using a directed graph which is constructed for a logic program as follows:

- For each atom in a program there is a unique node $v \in V$ in the graph;
- For each rule $A_0 :- L_1, \ldots, L_n$ and for each literal L_j there is an edge (A_0, ℓ, A_j) from the node A_0 to the node A_j with the label ℓ, where $\ell = \begin{cases} p & \text{if } L_j = A_j \text{ and} \\ n & \text{if } L_j = not\ A_j, \end{cases}$

where V is the set of edges $R \subseteq V \times L \times V$ and a set of labels $L \in \{n, p\}$. A program has stratified negation if the graph does not contain cyclic paths containing edges labelled with n.

3.2.2 Translation of C Code to a Datalog Program

Syntax of C Language

Our algorithm handles a subset of the C language defined in Fig. 3.7. We use indices to differ between multiple instances of rule applications, e.g. d_1 and d_2 denote different instances of application of the rule for definition d.

The rules in Fig. 3.7 define the following syntactic constructs:

Program. A program p consists of a sequence of definitions d.

		Grammar rules	*Explanation of the rules*
		Program	
p	::=	$d; s$	
		Definitions	
d	:=	$t\ id$ |	variable definition
		$d_1; d_2$	sequence of definitions
		Types	
t	::=	`int` | ... |	integral types
		`float` | ... |	floating point types
		t^* |	pointer type
		`struct` $st\_id\{t_1\ id_1, \ldots, t_n\ id_n\}$	structure type
		Statements	
s	::=	$e_1 = e_2$ |	assignment statement
		$s_1; s_2$ |	sequence statement
		`if` (e) `then` $\{s_1\}$ `else` $\{s_2\}$ |	conditional statement
		`while` (e) $\{s\}$	loop statement
		Expressions	
e	::=	id |	variable
		n |	constant number
		$'ch'$ |	character symbol
		$e_1 \circ_2 e_2$ |	binary operator, $\circ_2 \in \{+, -, *, /, \%\} \cup \{<, \leq, >, \geq, ==, !=\} \cup \{<<, >>, \&, \|, \sim, \wedge\} \cup \{\&\&, \|\|\}$
		$\circ_1 e_1$ |	unary operator, $\circ_1 \in \{+, -, \&, *, !\}$
		$e_1[e_2]$ |	array element
		$f(e_1, \ldots, e_n)$ |	function call
		`sizeof`(t)	size of type t

Figure 3.7 Syntax of a subset of C language handled by our algorithm. We use the following notation: variable identifiers id, id_i are elements of a set of variables Var, function identifiers f and structure identifiers st_id are elements of a set of functions $Func$ and a set of structures $Struct$ respectively. Symbols n and ch belong to the set of rational numbers and character symbols respectively. All other symbols, for which no rule is defined, are terminals, and are highlighted with fixed-width font.

Definition. A definition d is either a single variable definition, or a sequence of definitions.

Type. Definitions use data types, where a type t can be integral, floating point, pointer and structure type.

Statement. The set of handled statements includes assignment, sequence, conditional and loop statements and function calls (to the latter corresponds the rule $s ::= e$).

Expression. The set of handled expressions includes variables *id*, numeric and character constants (n and $'ch'$ respectively), binary and unary operators ($\circ_2$ and $\circ_1$ respectively), array elements, function calls and `sizeof`-expressions.

In this framework we define intra-procedural analyses, and the syntax specified in Fig. 3.7 does not allow to introduce user-defined functions in programs. Therefore, we restrict function calls to library function calls. C expression `sizeof(`t`)` evaluates to the number of bytes needed to store type t.

Pre-Processing of C Code

To handle C programs, our algorithm performs a pre-processing step, during which the algorithm makes the following transformations of the code:

Loops. Our algorithm re-writes `for` loops and `do-while` loops to `while` loops.

Example 3.2.1. Our algorithm rewrites the `for` loop in Fig. 3.8a to the `while` loop in Fig. 3.8b, with the initialisation part `i=0` prepended to the `while` loop.

Variable definition with initialisation. Our algorithm re-writes the initialisation part in variable definitions to an assignment statement.

Example 3.2.2. The algorithm rewrites the variable definition `int x=0;` to the sequence `int x; x=0;`.

Assignment statements. Our algorithm re-writes increment, decrement and compound assignment statements to assignment statements.

Example 3.2.3. Consider again the code in Fig. 3.8a. The increment statement `i++` in the loop is re-written to the statement `i=i+1`, as shown in Fig. 3.8b.

Implicit conversion to boolean. Our algorithm re-writes every integer-valued expression `expr` implicitly converted to a boolean-valued expression in a branch or loop condition to the expression `expr!=0`.

Example 3.2.4. Our algrorithm re-writes the branch condition `nondetermined` in Fig. 3.3, line 9 to `nondetermined!=0`.

Datalog Relations

Our method encodes the control-flow graph of a C program as a Datalog program by translating every node and edge in the control-flow graph of the C program to one or more facts in the logic program. In Table 3.4 we give the details of the translation: the second column of the table shows a syntactic construct, and the third column lists the corresponding logic facts.

Table 3.4 Translation of C constructs to Datalog.

C declaration		**Translation to logic program**
C construct	**Syntactic rule**, $d ::=$	

Variable definition	$t\ id$	`var(`$node_d$`)`
		`name(`$node_d, id$`)`
		`type(`$node_d, t$`)`

(a) Definitions

C statement		**Translation to logic program**
C construct	**Syntactic rule**, $s ::=$	

Assignment statement	$e_1 = e_2$	`assignment_stmt(`$node_s$`)`
		`lhs_expr(`$node_s, node_{e1}$`)`
		`rhs_expr(`$node_s, node_{e2}$`)`

Sequence statement	$s_1; s_2$	`sequence_stmt(`$node_s$`)`
		`stmt1(`$node_s, node_{s1}$`)`
		`stmt2(`$node_s, node_{s2}$`)`

Conditional statement	`if(`e_1`) {`s_1`} else {`s_2`}`	`if_stmt(`$node_s$`)`
		`condition(`$node_s, node_{e1}$`)`
		`then_stmt(`$node_s, node_{s1}$`)`
		`else_stmt(`$node_s, node_{s2}$`)`

Loop statement	`while(`e_1`) {`s_1`}`	`while_stmt(`$node_s$`)`
		`condition(`$node_s, node_{e1}$`)`
		`body(`$node_s, node_{s1}$`)`

(b) Statements

C expression		**Translation to logic program**
C construct	**Syntactic rule**, $e ::=$	

Constant literal	n	`const_literal(`$node_e$`)`
		`val(`$node_e, n$`)`
		`type(`$node_e, type_e$`)`

Character literal	$'ch'$	`char_literal(`$node_e$`)`
		`val(`$node_e, 'ch'$`)`
		`type(`$node_e$`, int)`

Binary operation	$e_1\ \circ_2\ e_2$	`bop_expr(`$node_e$`)`
		`opcode(`$node_e, \circ_2$`)`
		`lhs_expr(`$node_e, node_{e1}$`)`
		`rhs_expr(`$node_e, node_{e2}$`)`

Unary operation	$\circ_1\ e_1$	`uop_expr(`$node_e$`)`
		`opcode(`$node_e, \circ_1$`)`
		`sub_expr(`$node_e, node_{e1}$`)`

Array element	$e_1[e_2]$	`array_expr(`$node_e$`)`
		`arrayptr_expr(`$node_e, node_{e1}$`)`
		`arrayind_expr(`$node_e, node_{e2}$`)`

Function call	$f(e_1, \ldots, e_n)$	`call_expr(`$node_e$`)`
		`function(`$node_e, node_f$`)`
		`param(`$node_e$`,`i`,`$node_{e_i}$`)`

Size of type	`sizeof(`t`)`	`sizeof_expr(`$node_e$`)`

(c) Expressions

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

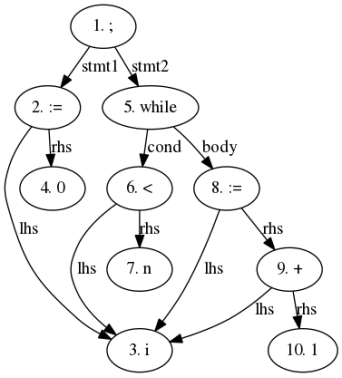
(a) Source code

```
1 i=0;
2 while (i<n) {
3   i = i+1;
4 }
```

(b) Preprocessed code.

```
1 var(3).                 17 const_literal(10).
2 name(3,"i").            18 val(10,"1").
3 type(3,int).            19 type(10,int).
4 const_literal(4).       20 bop_expr(9).
5 val(4,"0").             21 opcode(9,"+").
6 type(4,int).            22 lhs_expr(9,3).
7 assignment_stmt(2).     23 rhs_expr(9,10).
8 lhs_expr(2,3).          24 assignment_stmt(8).
9 rhs_expr(2,4).          25 lhs_expr(8,3).
10 var(7).                26 rhs_expr(8,9).
11 name(7,"n").           27 while_stmt(5).
12 type(7,int).           28 condition(5,6).
13 bop_expr(6).           29 body(5,8).
14 opcode(6,"<").         30 sequence_stmt(1).
15 lhs_expr(6,3).         31 stmt1(1,2).
16 rhs_expr(6,7).         32 stmt2(1,5).
```

(c) Datalog program



(d) Control flow graph

Figure 3.8 Translation of C code to a Datalog program.

In particular, we denote a node corresponding to a definition d as $\texttt{node}_d$, and nodes corresponding to an expression e or a statement s as $\texttt{node}_e$ and $\texttt{node}_s$ respectively. Similarly, we denote the types of respective constructs with $type_d$, $type_e$ and $type_s$. We will explain the meaning of each logic relation from the third column one by one in this chapter, as the relations appear in examples.

Example 3.2.5. In Fig.3.8c we give a translation to Datalog of the code in Fig. 3.8a. We show the corresponding control-flow graph (CFG) in Fig. 3.8d: we show identifiers and codes of operations inside nodes, and the relations between the nodes – on the edges of the CFG.

We now explain the relations which appear in the logic program. Here and further in this chapter we will shorten names of some relations when depicting a control-flow graph, as shown in Table 3.5.

Variable definition. The node `3` corresponds to the variable `i` and is encoded with the relations `var(3)`, `name(3,"i")` and `type(3,int)` at lines 1–3 of Fig. 3.8c;

In particular, the relation $\texttt{var(node}_\texttt{v}\texttt{)}$ encodes that the node $\texttt{node}_\texttt{v}$ corresponds to a variable. Next, the relation $\texttt{name(node}_\texttt{v}\texttt{,v)}$ encodes that `v` is the name of the variable (or function) at node $\texttt{node}_\texttt{v}$. Finally, the relation $\texttt{type(node}_\texttt{v}\texttt{,}t\texttt{)}$ encodes that the type of the variable, constant or function at the node $\texttt{node}_\texttt{v}$ is t.

Constant literal. The node `4` corresponds to the constant literal `0` and is encoded with the relations `const_literal(4)`, `val(4,"0")` and `type(4,int)` in lines 4–6 of Fig. 3.8c

In particular, the relation `const_literal(node_c)` encodes that the node $\texttt{node}_c$ corresponds to a constant literal `c`, and the relation `val(node_c,c)` encodes that `c` is the value of the constant at the node $\texttt{node}_c$.

Assignment statement. The node `2` corresponds to the assignment statement `i=0;` and is encoded with the relations `assignment_stmt(2)`, `lhs_expr(2,3)` and `rhs_expr(2,4)` at lines 7–9.

In particular, the relation `assignment_stmt(`n`)` encodes that the node n corresponds to an assignment statement. The relations `lhs_expr(`n_1`, `n_2`)` and `rhs_expr(`n_1`, `n_3`)` denote that the left- and right-hand side expressions of the statement or the expressions at the node n_1 are the expression at the nodes n_2 and n_3 respectively.

Binary operation. The node `6` corresponds to the binary operation `i<n`, encoded with the relations `bop_expr(6)`, `opcode(6,"<")`, `lhs_expr(6,3)` and `rhs_expr(6,7)` at lines 14–16.

In particular, the relation `bop_expr(`n`)` denotes that the node n corresponds to a binary operation. The relation `opcode(`n`,`*str*`)` denotes that the string *str* is the code of the operation at the node n. The relations `lhs_expr(`n_1`, `n_2`)` and `rhs_expr(`n_1`, `n_3`)` have the same meaning as for an assignment statement.

Loop statement. The node `5` corresponds to the loop statement. Note that our algorithm re-writes `for`-statements with `while`-statements, and therefore the statement `for(i=0;i<n;i++);` is re-written to a sequence of statements `i=0; while(i<n){i++;}`. The `while`-statement at the node `5` is encoded with the relations `while_stmt(5)`, `condition(5,6)`, `body(5,8)` at lines 27–29.

In particular, the relation `while_stmt(`n`)` encodes that the node n corresponds to a `while` statement. Next, the relation `condition(`n_1`, `n_2`)` encodes that the condition of the loop (or branch) statement at the node n_1 is the expression at the node n_2. Finally, the relation `body(`n_1`, `n_2`)` encodes that the body of the loop statement at the node n_1 is the statement at node n_2.

Sequence statement. The node `1` corresponds to the sequence statement obtained after re-writing the `for`-loop, as discussed above. The node `1` is encoded with the relations `sequence_stmt(1)`, `stmt1(1,2)` and `stmt2(1,3)` at lines 30–32.

In particular, the relation `sequence_stmt(`n`)` encodes that the node n corresponds to a sequence statement. The relations `stmt1(`n_1`,`n_2`)` and `stmt2(`n_1`, `n_2`)` encode that the first and second statements of the sequence statement at the node n_1 are the statements at the nodes n_2 and n_3 respectively. ▲

Table 3.5 Simplifications in our notation in control-flow-graphs.

Relation in logic program	Notation in CFG
assigment_statement	:=
sequence_stmt	;
while_stmt	while
if_stmt	if
condition	cond
then_stmt	then
else_stmt	else
return_stmt	return
lhs_expr	lhs
rhs_expr	rhs
arrayptr_expr	arrptr
arrayind_expr	arrind
call_expr	call
macro_call_expr	macro_call
sizeof_expr	sizeof
function	func
param0 /.../paramn	par0/.../parn
PTR_PLUS_INT	ptr+
PTR_MINUS_INT	ptr-
BIT_AND	&
BIT_OR	|

3.3 Definition of Roles

We now give the definition of the roles introduced in Section 3.1. Specifically, in Sections 3.3.1, 3.3.2 and 3.3.3 we define the roles introduced in Sections 3.1.1, 3.1.2 and 3.1.3 respectively. We split each of the three sets of roles into two subsets – the roles defined *without negation* and the roles defined *with negation*. Intuitively, the roles without negation are defined using patterns in which a variable *must* be used at least once, so that the variable is assigned the respective role, and the roles with negation – using patterns in which a variable *must not* be used.

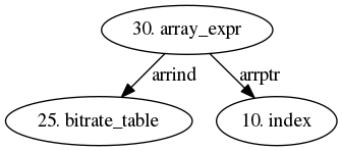
For the sake of readability, we put the definition of some auxiliary relations in Section 8.A on page 163.

3.3.1 Domain-Independent Roles

Below we give the definition of the domain-independent set of roles with and without negation.

Roles without Negation

Array index. Variable `X` is *array index* (denoted as `array_index(X)`, line 1), if at least one of the following conditions holds:



(a) Control flow graph

```
1 var(25).                    7 type(10,int).
2 type(25,26).                8 name(10,"index").
3 pointer_type(26).           9 array_expr(30).
4 name(25,                   10 arrayind_expr(30,25).
5   "bitrate_table").        11 arrayptr_expr(30,10).
6 var(10).
```

(b) Datalog program

Figure 3.9 Example for *array index*. The array expression `bitrate_table[index]` in line 14 of code in Fig. 3.1a on page 44 translated to Datalog.

- There exists an array subscript expression `Expr` with index `X`, which we denote with relation `arrayind_expr(Expr,X)` (line 1 in the listing below).
- `X` is assigned a variable `Y` which has the role *array index* (line 2).

```
1 array_index(X) :- var(X), array_expr(Expr), arrayind_expr(Expr,X).
2 array_index(X) :- var(X), assigned(X,Y), array_index(Y).
```

Here, the relations **`var`**, **`array_expr`** and **`arrayind_expr`** are generated by our algorithm during the translation of a C program to Datalog. We highlight such expressions with bold. Recall that the relation **`var`**`(X)` encodes the set of program variables. Next, the relation **`array_expr`**`(Expr)` denotes that `Expr` is an array subscription operation, and the relation **`array_ind`**`(Expr,X)` denotes that `X` is subscript of the array subscription expression `Expr`.

The rule at line 2 computes the transitive closure of the set of variables assigned the role *array index*. Specifically, the relation `assigned(X,Expr)`, which we define in Section 8.A, lines 2–3, denotes that the variable `X` is assigned the expression `Expr`. We note that similar rules for computing the transitive closure exist for all roles (the rules for *loop iterator* and *loop bound* are slightly different and we discuss them separately). For the sake of brevity, we will not comment on the rule for computing the transitive closure for the remaining roles.

Example 3.3.1. We illustrate the role *array index* on the example in Fig. 3.9. The figure shows the CFG of the expression `bitrate_table[index]` in Fig. 3.1a on page 44, line 14. The logic program, corresponding to the CFG in Fig. 3.9a, is shown in Fig. 3.9b.

We clarify the remaining relations in the logic program. Recall that the relation **`type`**`(Expr,Type)` denotes that the type of the expression `Expr` is `Type` and the relation **`name`**`(X,NameStr)` denotes that the identifier of the variable (or function) `X` is `NameStr`. Next, the relation **`pointer_type`**`( Type)` denotes that `Type` is a pointer type. Finally, the relation **`array_ptr`**`(Expr,PtrExpr)` denotes that `PtrExpr` is the subscripted array in the expression `Expr`.

For this program, the evaluation of the rule in line 1 infers that the variable `index` is *array index*, encoded as `array_index(node`$_{index}$`)`. ▲

Branch condition. Variable `X` is *branch condition* (denoted as `branch_cond(X)`, lines 3–6 in the listing below) if there exists an if statement `IfStmt` with condition `Cond`, with `X` being a *literal* of `Cond` (i.e. `X` is the smallest logical sub-expression of `Cond`):

```
3 branch_cond(X) :- var(X), if_stmt(IfStmt), condition(IfStmt,Cond),
4   literal(Cond,Lit), bop_expr(Lit), opcode(Lit,Opcode),
5   eq_opcode(Opcode), operand(Lit,X), operand(Lit,Const),
6   const_literal(Const), val(Const,"0").
7
8 branch_cond(X) :- var(X), assigned(X,Y), branch_cond(Y).
```

Recall that the relation **`const_literal`**`(Expr)` denotes that `Expr` is a constant number, and the relation **`val`**`(Expr,ValStr)` denotes that `ValStr` is the string representation of the value of the constant `Expr`.

Recall also that the relation **`bop_expr`**`(Expr)` denotes that the expression `Expr` is a binary operator. The relation **`opcode`**`(Expr,Opcode)` denotes that `Opcode` is the code of the binary operator `Expr`, e.g. ">", "==", "!=", etc. For the convenience of the reader, we have encoded all supported binary operation codes `Opcode` with the relation `bin_opcode(Opcode)` in Section 8.A, lines 243–262. The relation `eq_opcode(Opcode)` is defined in Section 8.A, lines 215–216 and includes the codes of the equality and dis-equality operations. The relation `operand(Expr, SubExpr)` denotes that `SubExpr` is (the left- or right-hand-side, or the only) sub-expression of expression `Expr`.

The relation **`if_stmt`**`(IfStmt)` denotes that `IfStmt` is an `if` statement, and the relation **`condition`**`(IfStmt,Cond)` denotes that the expression `Cond` is the condition of a (branching or loop) statement `IfStmt`.

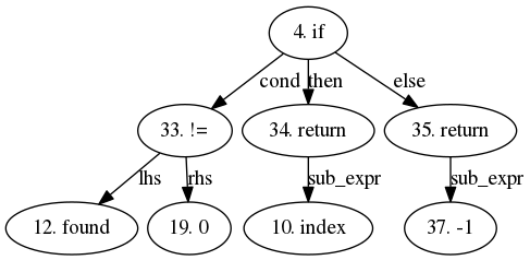
The relation `literal(Cond,X)` is defined in Section 8.A, line 27 and denotes that the expression `X` is a literal of the boolean-valued expression `Cond`.

Example 3.3.2. Consider the Fig. 3.10a which shows the CFG of the statement `if (found) return index; else return -1;` in Fig. 3.1a on page 44, lines 20–23. The corresponding logic program is shown in Fig. 3.10b.

The relation **`then_stmt`**`(IfStmt,ThenStmt)` in the logic program denotes that `ThenStmt` is the then-branch of the conditional statement `IfStmt`, and the relation **`else_stmt`**`(IfStmt,ElseStmt)` denotes that `ElseStmt` is the else-branch of the conditional statement `IfStmt`.

The relations **`lhs_expr`**`(Expr,LhsExpr)` and **`rhs_expr`**`(Expr,RhsExpr)` denote that `LhsExpr` and `RhsExpr` are the left- and right-hand-side operands of the binary operator (or of an assignment statement) `Expr`.

The relation **`return_stmt`**`(RetStmt)` denotes that `RetStmt` is a `return` statement, and the relation **`sub_expr`**`(Expr,SubExpr)` denotes that `SubExpr` is the (only) sub-statement of a statement or expression `Expr`.



(a) Control flow graph

```
1 if_stmt(4).
2 condition(4,33).
3 then_stmt(4,34).
4 else_stmt(4,35).
5 var(10).
6 name(10,"index").
7 type(10,int).
8 var(12).
9 name(12,"found").
10 type(12,int).
11 bop_expr(33).
12 lhs_expr(33,12).
13 rhs_expr(33,19).
14 opcode(33,"!=").
15 const_literal(19).
16 val(19,"0").
17 type(19,int).
18 return_stmt(34).
19 sub_expr(34,10).
20 return_stmt(35).
21 sub_expr(35,37).
22 const_literal(37).
23 type(37,int).
24 val(37,"-1").
```

(b) Datalog program

Figure 3.10 Example for *branch condition*. The statement `if (found) return index; else return -1;` in lines 20–23 of code in Fig. 3.1a on page 44 translated to Datalog.

For this example, the evaluation of the rule defining the relation `literal` (see Section 8.A, line 27) infers the fact `literal(33)`, where the node `33` corresponds to the expression `found!=0`. Then, the evaluation of the relation `operand` (Section 8.A, lines 10–11) infers the facts `operand(33,`$node_{found}$`)` and `operand(33,19)`, where the node `19` corresponds to the constant literal `0`. The definition of the relation `eq_opcode` (Section 8.A, lines 215–216) includes the fact `eq_opcode("!=")`.

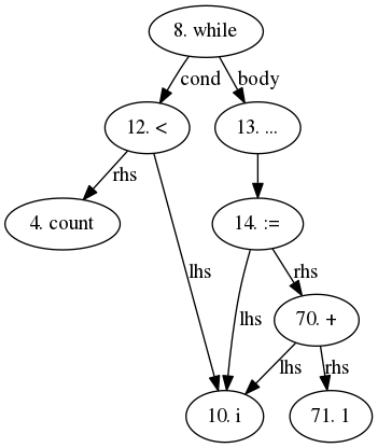
Finally, the evaluation of the rule in lines 3–6 infers that the variable `found` has the role *branch condition*, encoded as `branch_cond(`$node_{found}$`)`. ▲

Loop iterator. Variable `X` is *loop iterator* of loop `WhileStmt` (denoted as `loop_it(X, WhileStmt)`, lines 9–12 in the listing below) if the following conditions hold:

- There is a loop statement `WhileStmt` and assignment statement `AsgStmt` s.t. `AsgStmt` is a sub-statement of `WhileStmt`;
- In `AsgStmt` the variable `X` is assigned;
- The condition `Cnd` of `WhileStmt` is a binary comparison operator of which `X` is an operand.

```
9 loop_it(X,WhileStmt) :- var(X), while_stmt(WhileStmt),
10   assignment_stmt(AsgStmt),sub_stmt(WhileStmt,AsgStmt),
11   lhs_expr(AsgStmt,X), condition(WhileStmt,Cnd),bop_expr(Cnd),
12   opcode(Cnd,Opcode), compar_opcode(Opcode),operand(Cnd,X).
13
14 loop_it(X,WhileStmt) :- var(X),loop_it(Y,WhileStmt),
15   assignment_stmt(AsgStmt),sub_stmt(WhileStmt,AsgStmt),
16   lhs_expr(AsgStmt,X), rhs_expr(AsgStmt,Y).
```

Recall that the relation **`while_stmt`**`(WhileStmt)` denotes that `WhileStmt`



(a) Control flow graph

```
1 var(4).
2 type(4,int).
3 name(4,"count").
4 while_stmt(8).
5 condition(8,12).
6 body(8,13).
7 var(10).
8 type(10,int).
9 name(10,"i").
10 bop_expr(12).
11 lhs_expr(12,10).
12 rhs_expr(12,4).
13 opcode(12,"<").
14 assignment_stmt(14).
15 lhs_expr(14,10).
16 rhs_expr(14,70).
17 bop_expr(70).
18 lhs_expr(70,10).
19 rhs_expr(70,71).
20 opcode(70,"+").
21 const_literal(71).
22 type(71,int).
23 val(71,"1").
```

(b) Datalog program

Figure 3.11 Example for *loop iterator*, *loop bound*, *linear* and *counter*. The statement `for (i=0; i<count; ++i) {...}` in lines 15–25 of code in Fig. 3.1b on page 44 translated to Datalog.

is a `while` statement and that our translation algorithm rewrites every loop to a `while` statement. The relation **`assignment_stmt`**`(AsgStmt)` denotes that `AsgStmt` is an assignment statement.

The relation `sub_stmt(WhileStmt,AsgStmt)`, which we define in Section 8.A, lines 59–74, denotes that in the CFG the node corresponding to `AsgStmt` is a child node of the node corresponding to `WhileStmt`. The relation `compar_opcode(Opcode)`, which we define in Section 8.A, line 201–208 and 215–216, denotes that `Opcode` is a binary relational operation, e.g. $>$, $==$, etc.

Note that the relation for encoding *loop iterator* is binary, and takes the loop `WhileStmt` as a parameter. We need the parameter for the following reasons:

1. The rule in lines 14–16 computes the transitive closure of *loop iterators* only inside respective loop statements. Specifically, the rule infers that `X` is a *loop iterator* of loop `WhileStmt` if there is a loop iterator `Y` of loop `WhileStmt` and there is an assignment statement `AsgStmt` which is a sub-statement of `WhileStmt` and in which variable `X` is assigned `Y`. The relation `rhs_expr(`$\texttt{Expr}_1$`,`$\texttt{Expr}_2$`)` denotes that $\texttt{Expr}_2$ is the right-hand side expression of expression or statement $\texttt{Expr}_1$.
2. The parameter `WhileStmt` is used in the definition of the role *loop bound*, which we discuss later in this Section.

Example 3.3.3. Consider the example in Fig. 3.11a, which shows the CFG of the statement `for (i=0; i<count; ++i) {...}` from the code in Fig. 3.1b on page 44, line 15. We omit the body of the loop and denote the omitted body with dots. The corresponding logic program is shown in Fig. 3.11b.

The definition of the relation `compar_opcode` (see Section 8.A on page 163, lines 201–202) includes the fact `compar_opcode("<")`. The evaluation of the rule defining the relation `operand` (see also Section 8.A, lines 10–11) infers the facts `operand(12,node`$_i$`)` and `operand(12,node`$_{count}$`)`, where node `12` corresponds to the comparison operation in the loop condition. The evaluation of the rule defining the relation `sub_stmt` (in Section 8.A, lines 59–74) infers the fact `sub_stmt(8,14)`, where the nodes `8` and `14` correspond to the `while` statement and the assignment to the variable `i` in line 15 respectively.

Finally, the evaluation of the rule in lines 9–12 infers that the variable `i` is *loop iterator*, encoded as `loop_it(node`$_i$`)`. ▲

Loop bound. Variable `X` is *loop bound* of loop `WhileStmt` (denoted as `loop_bnd(X, WhileStmt)`, lines 17–19 in the listing below) if the following conditions hold:

- There is a `while` statement `WhileStmt`, the condition `Cnd` of which is a binary relational operation;
- The operands of `Cnd` are `X` and a variable `Y`;
- `Y` is a *loop iterator* of loop `WhileStmt`.

```
17 loop_bnd(X,WhileStmt) :- var(X), while_stmt(WhileStmt), condition(
     WhileStmt,Cnd),
18   bop_expr(Cnd), opcode(Cnd,Opcode), compar_opcode(Opcode),
19   operand(Cnd,X), operand(Cnd,Y), X!=Y, loop_it(Y,WhileStmt).
20
21 loop_bnd(X,WhileStmt) :- var(X), loop_bnd(Y,WhileStmt),
22   assignment_stmt(AsgnStmt), sub_stmt(WhileStmt,AsgnStmt),
23   lhs_expr(AsgnStmt,X), rhs_expr(AsgnStmt,Y).
```

The transitive closure of the set of *loop bound* variables (lines 21–23) is computed analogously to *loop iterator* variables.

Example 3.3.4. Consider again the CFG and logic program in Fig. 3.11. The rule in lines 17–19 infers that the variable `count` is *loop bound*, encoded as `loop_bound(node`$_{count}$`)`. ▲

Arithmetic. Variable `X` is *arithmetic* (denoted as `used_in_arithm(X)`, lines 24–25 and 27–28), if there is a binary or unary arithmetic operation `Expr` (e.g. +, –, >, etc), of which `X` is an operand.

```
24 used_in_arithm(X) :- var(X), bop_expr(Expr), opcode(Expr,Opcode),
25   arithm_opcode(Opcode), operand(Expr,X).
26
27 used_in_arithm(X) :- var(X), uop_expr(Expr), opcode(Expr,Opcode),
28   arithm_opcode(Opcode), operand(Expr).
29
30 used_in_arithm(X) :- var(X), assigned(X,Y), used_in_arithm(Y).
```

Specifically, the term `arithm_opcode(Opcode)`, which we define in Section 8.A, lines 218–222, denotes that the `Opcode` is a code of an arithmetic operation.

The term **`uop_expr`**`(Expr)` denotes that `Expr` is an unary operation. Again, for the convenience of the reader we encode the codes `Opcode` of all supported unary operators with the relation `un_opcode(Opcode)` in Section 8.A, lines 264–269.

Example 3.3.5. Consider again the CFG and logic program in Fig. 3.11.

The definition of the relation `arithm_opcode` includes the fact `arithm_opcode("+")`, and the evaluation of the rule defining the relation `operand` (see Section 8.A, lines 10–11) infers the fact `operand(70,node`$_i$`)`, where node `70` corresponds to the addition operation `i+1`.

Then, the evaluation of the rule in lines 24–25 infers that the variable `i` is *arithmetic*, encoded as `used_in_arithm(node`$_i$`)`. ▲

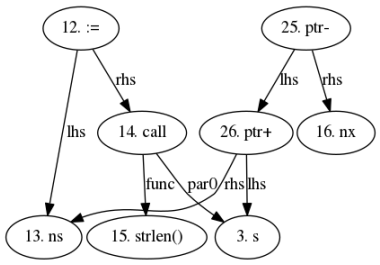
Pointer offset. Variable `X` is *pointer offset* (denoted as `offset(X)`) if at least one of the following conditions holds:

- There is a binary operator `Expr`, the code of which is a pointer addition operation + (denoted as `"PTR_PLUS_INT"`), and an operand of which is `X`, i.e. *offset* is added to a pointer (lines 31–32 in the listing below);
- There is a binary operator `Expr`, the code of which is a pointer subtraction operation – (denoted as `PTR_MINUS_INT`), and the right-hand side operand of which is `X`, i.e. the *offset* is subtracted from a pointer (lines 34–35).

```
31 offset(X) :- var(X), bop_expr(Expr), opcode(Expr,"PTR_PLUS_INT"),
32   operand(Expr,X).
33
34 offset(X) :- var(X), bop_expr(Expr), opcode(Expr,"PTR_MINUS_INT"),
35   rhs_expr(Expr,X).
36
37 offset(X) :- var(X), assigned(X,Y), offset(Y).
```

Example 3.3.6. For example, in Fig. 3.12a we show the CFG of the the statement `ns=strlen(s);` in line 5 and the expression `s+ns-nx` in line 11 of the code in Fig. 3.2a on page 47. The translation these constructs to Datalog is shown in Fig. 3.12b.

In particular, the relation **`func_decl`**`(Func)` denotes that `Func` is a function declaration. The term **`call_expr`**`(CallExpr)` denotes that `CallExpr` is a call expression, the relation **`function`**`(CallExpr,Func)` denotes that in the expression `CallExpr` the function `Func` is called, and the relation **`param`**`(CallExpr, I,ParExpr)` denotes that the `I`-th parameter of the call expression `CallExpr` is `ParExpr`.



(a) Control flow graph

```
1 var(3).
2 type(3,ptr).
3 name(3,"s").
4 var(13).
5 type(13,int).
6 name(13,"ns").
7 assignment_stmt(12).
8 lhs_expr(12,13).
9 rhs_expr(12,14).
10 func_decl(15).
11 name(15,"strlen").
12 call_expr(14).
13 function(14,15).
14 param(14,0,3).
15 var(16).
16 type(16,int).
17 name(16,"nx").
18 bop_expr(25).
19 lhs_expr(25,26).
20 rhs_expr(25,16).
21 opcode(25,
22   "PTR_MINUS_INT").
23 bop_expr(26).
24 lhs_expr(26,3).
25 rhs_expr(26,13).
26 opcode(26,
27   "PTR_PLUS_INT").
```

(b) Datalog program

Figure 3.12 Example for *input* and *offset*. Statement `ns=strlen(s)` in line 5 and the expression `s+ns-nx` in line 11 of code in Fig. 3.2a on page 47 translated to Datalog.

In this example, the evaluation of the rule in lines 31–32 results in the inference of the fact `offset(node`$_{nx}$`)`. Similarly, the evaluation of the rule in lines 34–35 infers that the variable `ns` is *offset*, encoded as `offset(node`$_{ns}$`)`. ▲

Input. Variable `X` is *input* (denoted as `input(X)`, line 38 in the listing below) if `X` is assigned the result of a call to an externally defined function `Func`.

```
38 input(X) :- var(X), assigned_call(X,Func), ext_func(Func).
39 input(X) :- var(X), assigned(X,Y), input(Y).
```

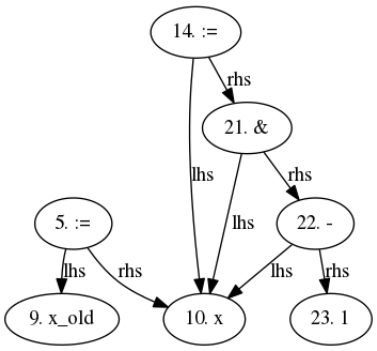
In particular, the relation `assigned_call(X,Func)`, which we define in Section 8.A, lines 24–25, denotes that `X` is assigned a call expression, in which the function `Func` is called. The relation `ext_func(Func)`, which we define in Section 8.A on page 163, line 89, denotes that `Func` is an externally defined function.

Example 3.3.7. Consider again the example in Fig. 3.12a.

The evaluation of the rules defining the relations `ext_func` and `assigned_call` infers the facts `ext_func(node`$_{strlen}$`)` and `assigned_call(node`$_{ns}$`, node`$_{strlen}$`)` respectively. The evaluation of the rule in line 38 infers that the variable `ns` is *input*, encoded as `input(node`$_{ns}$`)`. ▲

Bitvector. Variable `X` is *bitvector* (denoted as `bitvector(X)`), if at least one of the following conditions holds:

– There exists a binary operator `Expr` which is a bitwise operation, of which `X` is an operand (lines 40–41); Specifically, the relation `bit_opcode(Opcode)`,



(a) Control flow graph

```
1 var(9).
2 type(9,int).
3 name(9,"x_old").
4 var(10).
5 type(10,int).
6 name(10,"x").
7 assignment_stmt(5).
8 lhs_expr(5,9).
9 rhs_expr(5,10).
10 assignment_stmt(14).
11 lhs_expr(14,10).
12 rhs_expr(14,21).
13 bop_expr(21).
14 lhs_expr(21,10).
15 rhs_expr(21,22).
16 opcode(21,"BIT_AND").
17 bop_expr(22).
18 lhs_expr(22,10).
19 rhs_expr(22,23).
20 opcode(22,"-").
21 const_literal(23).
22 type(23,int).
23 val(23,"1").
```

(b) Datalog program

Figure 3.13 Example for *bitvector*. Lines 5 and 9 of code in Fig. 1.1a on page 7 translated to Datalog.

which we define in Section 8.A, lines 195–198, denotes that `Opcode` is the code of a bit operation, e.g. bit-and, bit-or, etc.

– There exists an unary operator `Expr` which is a bitwise operation and takes `X` as an operand (lines 43–44).

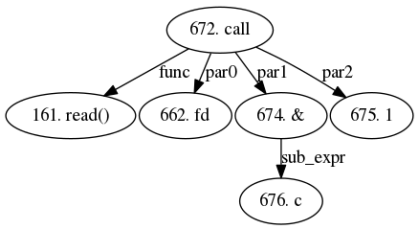
```
40 bitvector(X):- var(X), bop_expr(Expr), opcode(Expr,Opcode),
41   bit_opcode(Opcode), operand(Expr,X).
42
43 bitvector(X) :- var(X), uop_expr(Expr), opcode(Expr, Opcode),
44   bit_opcode(Opcode), operand(Expr,X).
45
46 bitvector(X) :- var(X), assigned(X,Y), bitvector(Y).
```

Example 3.3.8. For example, in Fig. 3.13a is shown the CFG of the statements `x_old=x;` and `x=x&(x-1);` in Fig. 1.1a on page 7, lines 5 and 9 respectively. The corresponding logic program is shown in Fig. 3.13b.

The definition of the relation `bit_opcode` includes the fact `bit_opcode("BIT_AND")`. Then, the evaluation of the rule in lines 40–41 infers that the variable `X` is *bitvector*, encoded as `bitvector(node`$_x$`)`. ▲

Example 3.3.9. We now illustrate the rule for computing the transitive closure on example of the role *bitvector*. The evaluation of the rule defining the relation `assigned` (see Section 8.A on page 163, lines 2–3) infers the fact `assigned(node`$_{x\_old}$`,node`$_x$`)`. Then, the evaluation of rule in line 46 infers that the variable `x_old` is *bitvector*, encoded as `bitvector(node`$_{x\_old}$`)`. ▲

File descriptor. Variable `X` is *file descriptor* (denoted as `file_descr(X)`), if at least one of the following conditions holds:



(a) Control flow graph

```
1 func_decl(161).
2 type(161,int).
3 name(161,"read").
4 var(662).
5 type(662,int).
6 name(662,"fd").
7 call_expr(672).
8 function(672,161).
9 param(672,0,662).
10 param(672,1,674).
11 const_literal(675).
12 type(675,int).
13 val(675,"1").
14 param(672,2,675).
15 var(676).
16 type(676,int).
17 name(676,"c").
18 uop_expr(674).
19 sub_expr(674,676).
20 opcode(674,"ADDR_OF").
```

(b) Datalog program

Figure 3.14 Example for *file descriptor* and *unresolved*. The function call in line 7 of code in Fig. 1.1b on page 7 translated to Datalog.

– `X` is passed as the `I`-th parameter to a function `Func`, s.t. `Func` is a library function which takes a file descriptor as input, e.g. `read()` , `write()`, etc.) (lines 47–48 in the listing below).

  We denote such library functions with the relation `file_use_func(FuncName,I)`, which we define in Section 8.A, lines 118–129. Here, `FuncName` is the name of the library function, and `I` is the number of the parameter which corresponds to a file descriptor.

  The relation `act_arg(Func,I,Expr)`, which we define in Section 8.A, lines 17–21, denotes there is a call to a function (or to a macro) `Func` and the `I`-th parameter of `CallExpr` is `Expr`.

– `X` is assigned the result of a call to a function `Func` which returns a file descriptor, e.g. `open()`, `creat()`, `dup()`, etc., (lines 50–51).

  We denote such library functions with the relation `file_def_func(FuncName,fres)`, where `FuncName` is the name of the library function, and the constant term `fres` denotes that the variable with the respective role (in this case, a file descriptor) is returned as output of the function.

```
47 file_descr(X) :- var(X), act_arg(Func,I,X), name(Func,FuncName),
48   file_use_func(FuncName,I).
49
50 file_descr(X) :- var(X), assigned_call(X,Func), name(Func,FuncName),
51   file_def_func(FuncName,fres).
52
53 file_descr(X) :- var(X), assigned(X,Y), file_descr(Y).
```

Example 3.3.10. In Fig. 3.14a is shown the CFG of the expression `read(fd, &c, 1)` from the code in Fig. 1.1b on page 7, line 7. The corresponding logic program is shown in Fig. 3.14.

The relation **`sizeof_expr`**`(Expr)` denotes that `Expr` is a C `sizeof` expression.

In this example, the definition of the relation `file_use_func` includes the fact `file_use_func(read,0)`, and the evaluation of the rule defining the relation `act_arg` (see Section 8.A, lines 17–21) infers the fact `act_arg(node`$_{read}$`,0,node`$_{fd}$`)`. Finally, the evaluation of the rule in lines 47–48 infers that the variable `fd` is *file descriptor*, encoded as `file_descr(node`$_{fd}$`)`. ▲

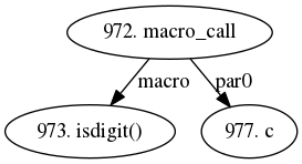
Character. Variable `X` is *character* (denoted as `char(X)`) if at least one of the following conditions holds:

- `X` is assigned a character literal, e.g. 'A' (line 54 in the listing below).
  Specifically, the relation **`char_literal`**`(CharLit)` denotes that `CharLit` is a character literal.
- `X` is passed as the `I`-th argument to a library function `Func` which manipulates characters and takes a character as the `I`-th parameter, e.g. `putc()`, `isalpha()`, etc. (lines 56–57).
  We denote such functions with the relation `char_use_func(FuncName, I)`, which we define in Section 8.A on page 163, lines 94–108. The parameters of the relation have the same meaning as in the relation `file_use_func` (see the role *file descriptor* above).
- `X` is assigned the result of a call to a library function which returns a character, e.g. `getc()`, `getchar()`, etc. (lines 59–60).
  We denote such functions with term `char_def_func(FuncName, fres)`, which we define in Section 8.A, lines 111–116. The parameters of the relation have the same meaning as in the relation `file_def_func`.

```
54 char(X) :- var(X), assigned(X,CharLit), char_literal(CharLit).
55
56 char(X) :- var(X), act_arg(Func,I,X), name(Func,FuncName),
57   char_use_func(FuncName,I).
58
59 char(X) :- var(X), assigned_call(X,Func), name(Func,FuncName),
60   char_def_func(FuncName,fres).
61
62 char(X) :- var(X), assigned(X,Y), char(Y).
```

Example 3.3.11. Consider the example in Fig. 3.15a, which shows the CFG of the expression `isdigit(c)` of the code in Fig. 1.1b, line 8.

Note that the library function `isdigit()`, as most functions from the standard library which manipulate characters, is implemented as a macro. Here, the fact **`macro_call_expr`**`(972)` denotes that the node 972 is a call to a macro. The fact **`macro`**`(972,973)` denotes that the node `973` is the corresponding macro definition (the name of which is `isdigit()`, as denoted by the fact `name(973, "isdigit")`). Finally, the fact **`param`**`(972,0,`$node_c$`)` denotes that the first parameter to the macro call is the variable `c`.



(a) Control flow graph

```
1 macro_call_expr(972).
2 macro(972,973).
3 param(972,0,977).
4 macro_decl(973).
5 name(973,"isdigit").
6 var(977).
7 type(977,int).
8 name(977,"c").
```

(b) Datalog program

Figure 3.15 Example for *character*. The macro call `isdigit(c)` in line 8 of code in Fig. 1.1b on page 7 translated to Datalog.

In this example, the evaluation of the rule defining the relation `act_arg()` (see the definition in Section 8.A, lines 17–21) infers the fact `act_arg(973, 0,`$node_c$`)`. The definition of the relation `char_use_func` includes the fact `char_use_func("isdigit,0")`. Finally, the evaluation of the rule in lines 56–57 infers that `c` is a *character*, encoded as `char(`$node_c$`)`. ▲

Allocation size. Variable `X` is *allocation size* (denoted as `alloc_size(X)`) if at least one of the following cases holds:

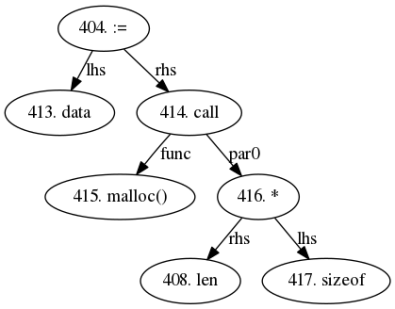
- `X` is passed to a function `Func` which is a memory allocation function, denoted as `alloc_arg(X)` (line 63 in the listing below).

  Specifically, the relation `alloc_arg` in lines 71–72 includes the expressions passed to a memory allocation function (e.g. `malloc()`, `calloc()`, etc.) as a parameter which corresponds to the size of the allocated memory in the respective function.

  The relation `dyn_mem_size_func(FuncName,I)`, which we define in Section 1.1.2, lines 179–186 denotes that `FuncName` is the name of the library functions which allocates memory and takes the size of the memory to be allocated as `I`-th parameter.

- A binary operator `Expr` is passed to a memory allocation function, and `Expr` is a multiplication of `X` and a `sizeof()` expression `SizeOfExpr` (lines 65–67).

```
63 alloc_size(X) :- var(X), alloc_arg(X).
64
65 alloc_size(X) :- alloc_arg(Expr), bop_expr(Expr), opcode(Expr,"*"),
66   operand(Expr,X),var(X), operand(Expr,SizeOfExpr),
67   sizeof_expr(SizeOfExpr).
68
69 alloc_size(X) :- var(X), assigned(X,Y), alloc_size(Y).
70
71 alloc_arg(X) :- act_arg(Func,I,X), name(Func,FuncName),
72   dyn_mem_size_func(FuncName,I).
```



(a) Control flow graph

```
1 var(408).
2 type(408,int).
3 name(408,"len").
4 var(413).
5 type(413,ptr).
6 name(413,"data").
7 assignment_stmt(404).
8 lhs_expr(404,413).
9 rhs_expr(404,414).
10 call_expr(414).
11 function(414,415).
12 param(414,0,416).
13 func_decl(415).
14 type(415,ptr).
15 name(415,"malloc").
16 bop_expr(416).
17 lhs_expr(416,417).
18 rhs_expr(416,408).
19 opcode(416,"*").
20 sizeof_expr(417).
```

(b) Datalog program

Figure 3.16 Example for *allocation size*. The statement `data=malloc( sizeof(int)*len);` in lines 15–16 of code in Fig. 3.4a on page 51, translated to Datalog.

Example 3.3.12. In Fig. 3.16a is shown the CFG of the statement `data=malloc( sizeof(int)*len);` from the code in Fig. 3.4a on page 51, lines 15–16. The corresponding logic program is shown in Fig. 3.16b.

The definition of the relation `dyn_mem_size_func` includes the fact `dyn_mem_size_func(malloc,0)`, and the evaluation of the rule in lines 71–72 infers the fact `alloc_arg(node`$_{len}$`)`.

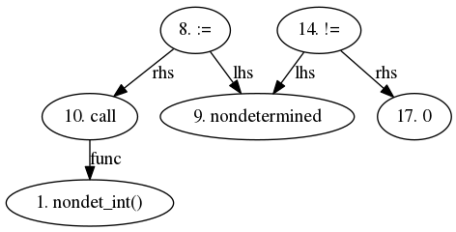
The evaluation of the rule defining the relation `operand` (see Section 8.A, lines 10–11) infers the facts `operand(416,node`$_{len}$`)` and `operand(416,417)`, where the node `416` corresponds to the multiplication operation `sizeof(int)*len`, and the node `417` corresponds to the expression `sizeof(int)`.

Finally, the evaluation of the rule in lines 65–67 infers that the variable `len` is *allocation size*, encoded as `alloc_size(node`$_{len}$`)`. ▲

Unresolved. Variable `X` is *unresolved* (denoted as `unresolved(X)`) if at least one of the following conditions holds:

- The variable `X` is assigned an expression `Expr` which is included in the relation `unresolved_expr` (line 73 in the listing below).
- `X` is passed by value to an external function `Func`. Specifically, there is an expression `Expr` passed as an argument to a call to al function `Func`, s.t. `Expr` is an address-of operation (encoded with the code `"ADDR_OF"`), of which `X` is the operand (lines 75– 76). Recall that the fact `ext_func(Func)` encodes that `Func` is an external function.

```
73 unresolved(X) :- var(X), assigned(X, Expr), unresolved_expr(Expr).
74
75 unresolved(X) :- var(X), act_arg(Func,I, Expr), ext_func(Func),
76   uop_expr(Expr), opcode(Expr,"ADDR_OF"), operand(Expr,X).
77
```



(a) Control flow graph

```
1 var(1).
2 type(1,int).
3 name(1,
4   "nondet_int").
5 func_decl(1).
6 var(9).
7 type(9,int).
8 name(9,
9   "nondetermined").
10 assignment_stmt(8).
11 lhs_expr(8,9).
12 rhs_expr(8,10).
13 call_expr(10).
14 function(10,1).
15 const_literal(17).
16 type(17,int).
17 name(17,"0").
18 bop_expr(14).
19 lhs_expr(14,9).
20 rhs_expr(14,17).
21 opcode(14,"!=").
```

(b) Datalog program

Figure 3.17 Example for *unresolved*. The statement `nondetermined=nondet_int();` and the condition expression `nondetermined` in lines 6 and 9 respectively of code in Fig. 3.3 on page 49, translated to Datalog.

```
78 unresolved(X) :- var(X), assigned(X,Y), unresolved(Y).
```

An expression `Expr` is included in the relation `unresolved_expr` if `Expr` is a call to an external function (line 79 in the listing below).

```
79 unresolved_expr(Expr) :- called(Func,Expr), ext_func(Func).
```

In particular, the relation `called(Func,CallExpr)` which we define in Section 8.A, line 14 denotes that `CallExpr` is a call expression to the function `Func`.

Example 3.3.13. Consider the example in Fig. 3.17a, which shows the CFG of the statement `nondetermined=nondet_int();` and the condition expression `nondetermined` in lines 6 and 9 respectively of code in Fig. 3.3 on page 49. The corresponding logic program is shown in Fig. 3.17b.

The evaluation of the rule defining the relation `ext_func` (see Section 8.A on page 163, line 89) infers the fact $\texttt{ext\_func}(\texttt{node}_{\textit{nondet\_int}})$. Then, the evaluation of the rule in line 79 infers the fact `unresolved_expr(10)`, where the node `10` corresponds to the call of the function `nondet_int()`. Finally, the evaluation of the rule in line 73 infers that the variable `nondetermined` is *unresolved*, encoded as $\texttt{unresolved}(\texttt{node}_{\textit{nondetermined}})$. ▲

Roles with Negation

Boolean. Variable `X` is *boolean* (denoted as `bool(X)`, line 80 in the listing below) if `X` is not a non-boolean variable, denoted with relation `non_bool_var(X)`:

```
80 bool(X) :- var(X), not non_bool_var(X).
```

Variable `X` is non-boolean if `X` is:

1. assigned a non-boolean variable (line 83), or
2. assigned an expression `Expr` which is not boolean (denoted as `not bool_expr(Expr)`, line 86; we define and explain this relation below), or
3. used in a binary or unary operation which is neither a logical operation (encoded as `not logical_opcode(Opcode)`) nor equality comparison (encoded as `not eq_opcode(Opcode)`) (lines 89–90, 93–94 respectively), or
4. compared for equality to a non-boolean variable (lines 97–98) or
5. compared for equality to an expression which is not boolean (lines 101–102).

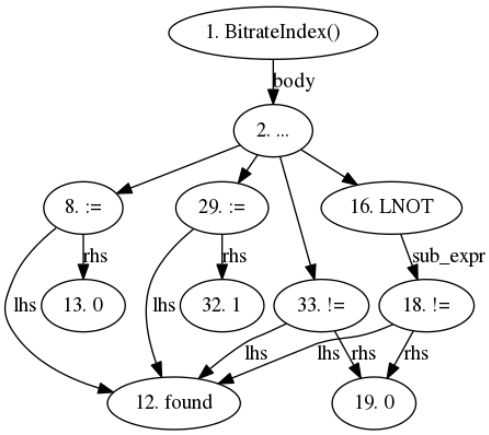
```
81 % Case split for non-boolean variables
82 %1. X is assigned a non-boolean variable
83 non_bool_var(X) :- assigned(X, Y), var(Y), non_bool_var(Y).
84
85 %2. X is assigned a non-boolean expression
86 non_bool_var(X) :- assigned(X, Expr), not bool_expr(Expr).
87
88 %3. X is used in a non-boolean binary operator
89 non_bool_var(X) :- bop_expr(Expr), opcode(Expr,Opcode),
90   not logical_opcode(Opcode), not eq_opcode(Opcode), operand(Expr,X).

91
92 %4. X is used in a non-boolean unary operator
93 non_bool_var(X) :- uop_expr(Expr), opcode(Expr,Opcode),
94   not logical_opcode(Opcode), operand(Expr,X).
95
96 %5. X is compared to a non-boolean variable
97 non_bool_var(X) :- bop_expr(Expr), opcode(Expr,Opcode), eq_opcode(
       Opcode),
98   operand(Expr,X), operand(Expr,Y), var(Y), non_bool_var(Y).
99
100 %6. X is compared to a non-boolean expression
101 non_bool_var(X) :- bop_expr(Expr), opcode(Expr,Opcode), eq_opcode(
       Opcode),
102   operand(Expr,X), operand(Expr,NonBoolExpr), not bool_expr(
         NonBoolExpr).
```

The relations `logical_opcode`, `eq_opcode` and `bool_res_opcode` are defined in Section 8.A on page 163, lines 233–235, 215–216 and 238–240 respectively and include the codes of logical operations, (dis-)equality operations and the operations which return a boolean value respectively.

A boolean expression `Expr` (denoted as `bool_expr(Expr)`, lines 105–114 in the listing below) is one of the following:

- A constant literal `0` or `1` (lines 105–106),
- A boolean-valued binary or unary operation, e.g. ">", "!=", logical-or, etc. (lines 109–114).



(a) Control flow graph

```
1 func_decl(1).
2 name(1,
3   "BitrateIndex").
4 body(1,2).
5 var(12).
6 type(12,int).
7 name(12,"found").
8 assignment_stmt(8).
9 lhs_expr(8,12).
10 rhs_expr(8,13).
11 const_literal(13).
12 val(13,"0").
13 type(13,int).
14 uop_expr(16).
15 opcode(16,"LNOT").
16 sub_expr(16,18).
17 bop_expr(18).
18 lhs_expr(18,12).
19 rhs_expr(18,19).
20 opcode(18,"!=").
21 const_literal(19).
22 val(19,"0").
23 type(19,int).
24 assignment_stmt(29).
25 lhs_expr(29,12).
26 rhs_expr(29,32).
27 const_literal(32).
28 val(32,"1").
29 type(32,int).
30 bop_expr(33).
31 lhs_expr(33,12).
32 rhs_expr(33,19).
33 opcode(33,"!=").
```

(b) Datalog program

Figure 3.18 Example for *boolean*. The statements `found=0;` and `found=1;` and expressions `!(found!=0)` and `found!=0` in lines 11, 15, 13 and 20 of the code in Fig. 3.1a on page 44 translated to Datalog.

```
103 % Case split for boolean expressions
104 %1. zero or unit constant
105 bool_expr(Expr) :- const_literal(Expr), val(Expr,"0").
106 bool_expr(Expr) :- const_literal(Expr), val(Expr,"1").
107
108 %2. boolean-valued binary operator
109 bool_expr(Expr) :- bop_expr(Expr), opcode(Expr,Opcode),
110   bool_res_opcode(Opcode).
111
112 %3. boolean-valued unary operator
113 bool_expr(Expr) :- uop_expr(Expr), opcode(Expr,Opcode),
114   bool_res_opcode(Opcode).
```

Example 3.3.14. In Fig. 3.18a is shown the CFG of the statements `found=0;` and `found=1;` and expressions `!(found!=0)` and `found!=0` in lines 11, 15, 13 and 20 of the code in Fig. 3.1a on page 44. We omit the statements of the function `BitrateIndex()` which do not use the variable `found`, and denote the ommited part of the code with dots in node `2`. The corresponding logic program is shown in Fig. 3.18b.

In this example, the evaluation of the rules in lines 105 infers the fact `bool_expr(13)`, where the node `13` corresponds to the constant literal `0`. Similarly, the facts `bool_expr(19)` and `bool_expr(32)` are inferred.

Then, the evaluation of the rules in lines 83–102 infers that the transitive closure of the relation `non_bool_var` does not contain the fact `non_bool_var(`$node_{found}$`)`.

Finally, the evaluation of the rule in line 80 infers that `found` is *boolean*, encoded with the fact `bool(`$node_{found}$`)`. ▲

Constant assigned. Variable `X` is *constant-assigned* (denoted as `const_assign(X)`, line 115 in the listing below), if `X` is not non-constant assigned (denoted as `non_const_assign_var(X)`):

```
115 const_assign(X) :- var(X), not non_const_assign_var(X).
```

A variable `X` is non-constant assigned (line 116) if `X` is assigned an expression which is not a constant expression:

```
116 non_const_assign_var(X) :- assigned(X,Expr), not const_expr(Expr).
```

A constant expression `Expr` (denoted as `const_expr(Expr)`) is either a constant literal (line 118), or a binary or unary operation taking constant expressions as operands (lines 120–121 and 123 respectively):

```
117 % constant expressions
118 const_expr(Expr) :- const_literal(Expr).
119
120 const_expr(Expr) :- bop_expr(Expr), lhs_expr(Expr,Op1),
121   const_expr(Op1), rhs_expr(Expr,Op2), const_expr(Op2).
122
123 const_expr(Expr) :- uop_expr(Expr), operand(Expr,Op), const_expr(Op).
```

Example 3.3.15. Consider again the example in Fig. 3.18.

The evaluation of the rule in line 118 infers the facts `const_expr(13)`, `const_expr(32)` and `const_expr(19)` (recall that the nodes `13`, `32` and `19` correspond to the constant literals `0`, `1` and `0`).

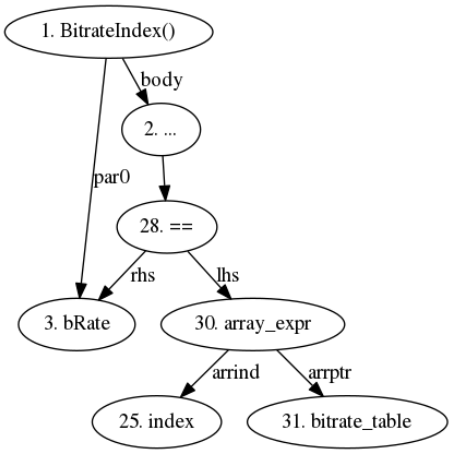
Then, the evaluation of the rule in line 116 infers that the transitive closure of the relation `non_const_assign_var` does not include the fact `non_const_assign_var(`$node_{found}$`)`.

Finally, the evaluation of the rule in line 115 infers that the variable `found` is *constant assigned*, encoded as `const_assign(`$node_{found}$`)`. ▲

Enumeration. Variable `X` is enumeration (denoted as `enum(X)`, line 124) if `X` is *constant assigned* and `X` is not compared to non-constant expressions (encoded as `not non_const_compar(X)`):

```
124 enum(X) :- const_assign(X), not non_const_compar(X).
```

Specificallly, the relation `non_const_compar` (lines 125–127) includes a variable `X` if `X` is compared to an expression `Expr` which is not a constant expression:



(a) Control flow graph

```
1 func_decl(1).
2 name(1,
3   "BitrateIndex").
4 body(1,2).
5 param(1,0,3).
6 var(3).
7 type(3,int).
8 name(3,"bRate").
9 var(25).
10 type(25,int).
11 name(25,"index").
12 bop_expr(28).
13 opcode(28,"==").
14 lhs_expr(28,30).
15 rhs_expr(28,3).
16 array_expr(30).
17 arrayind_expr(30,25).
18 arrayptr_expr(30,31).
19 var(31).
20 type(31,32).
21 pointer_type(32).
22 name(31,
23   "bitrate_table").
```

(b) Datalog program

Figure 3.19 Example for *syntactic constant*. The expression `bitrate_table[index]==bRate` in lines 14 of code in Fig. 3.1a on page 44 translated to Datalog.

```
125 non_const_compar(X) :- bop_expr(Expr), opcode(Expr,Opcode),
126   compar_opcode(Opcode), operand(Expr,X), operand(Expr,Op), X!=Op,
127   not const_expr(Op).
```

Example 3.3.16. Consider again the example in Fig. 3.18. We have already demonstrated that for this example the fact `const_assign(`$node_{found}$`)` is inferred.

Then, the evaluation of the rule in lines 125–127 infers that the transitive closure of the relation `non_const_compar` does not include the fact `non_const_compar(`$node_{found}$`)`.

Finally, the evaluation of the rule in line 124 infers that the variable `found` is *enumeration*, encoded as `enum(`$node_{found}$`)`. ▲

Syntactic constant. Variable `X` is *syntactic constant* (denoted as `synt_const(X)`, line 128) if `X` is not modified (encoded as `not modified(X)`):

```
128 synt_const(X) :- var(X), not modified(X).
```

Variable `X` is modified, if either `X` is directly assigned some expression (line 129) or there is an address-of operation of which `X` is an operand (line 130):

```
129 modified(X) :- assigned(X,Expr).
130 modified(X) :- uop_expr(Expr), opcode(Expr,"ADDR_OF"), operand(Op, X).
```

Example 3.3.17. Consider the example in Fig. 3.19a, which shows the CFG of the expression `bitrate_table[index]==bRate` in lines 14 of code in Fig. 3.1a on page 44. This expression is the only expression in the function `BitrateIndex()` which uses the variable `bRate`. We omit the remaining code and denote it with dots. The corresponding logic program is shown in Fig. 3.19b.

In this example, the evaluation of the rules in lines 129–130 infers that the transitive closure of the relation `modified` does not include the fact `modified(`$node_{bRate}$`)`. Then, the evaluation of the rule in line 128 infers that the variable `bRate` is *syntactic constant*, encoded as `synt_const(`$node_{bRate}$`)`. ▲

Counter. Variable `X` is *counter* (denoted as `counter(X)`, line 131 in the listing below), if `X` is not a non-counter variable (denoted as `non_counter(X)`):

```
131 counter(X) :- var(X), not non_counter(X).
```

A variable `X` is non-counter (lines 132–133) if `X` is assigned an expression which is neither a constant expression (denoted as `const_expr(Expr)`, see definition in lines 118–123), nor a counter expression (denoted as `cnt_expr(Expr,X)`):

```
132 non_counter(X) :- assigned(X,Expr), not const_expr(Expr),
133   not cnt_expr(Expr,X).
```

A counter expression is one of the following:

- A binary addition operation, an operand of which is `X` (line 134);
- A binary subtraction operation, the left-hand side operand of which is `X` (line 135).

```
134 cnt_expr(Expr,X) :- bop_expr(Expr), opcode(Expr,"+"), operand(Expr,X).
135 cnt_expr(Expr,X) :- bop_expr(Expr), opcode(Expr,"-"),lhs_expr(Expr,X).
```

Example 3.3.18. Consider again the example in Fig. 3.11 on page 72.

The evaluation of the rule in lines 134 infers the fact `cnt_expr(70,`$node_i$`)`, and the evaluation of the rule in lines 132–133 infers that the transitive closure of the relation `non_counter` does not include the fact `non_counter(`$node_i$`)`. Finally, the evaluation of the rule in line 131 infers that the variable `i` is *counter*, encoded as `counter(`$node_i$`)`. ▲

Linear. Variable `X` is *linear* (denoted as `linear(X)`, line 136) if `X` is not a non-linear variable:

```
136 linear(X) :- var(X), not non_lin_var(X).
```

Variable `X` is non-linear, denoted with term `non_lin_var(X)` if `X` is

1. assigned a non-linear variable (line 139), or
2. assigned the result of a non-linear-shape expression `Expr` (line 142), denoted as `not linear_shape(Expr)`, or
3. assigned the result of a linear-shape expression `Expr` which uses a variable `Y` as an operand (denoted as `linear_expr_subvar(Expr,Y)`), s.t. `Y` is non-linear (lines 146–147).

```
137 % Case split for non-linear variables
138 %1. X is assigned a non-linear variable
139 non_lin_var(X) :- assigned(X,Y), var(Y), non_lin_var(Y).
140
141 %2. X is assigned the result of a non-linear operation
142 non_lin_var(X) :- assigned(X,Expr), not linear_shape(Expr).
143
144 %3. X is assigned the result of a linear operation
145 %   in which non-linear variables are used
146 non_lin_var(X) :- assigned(X,Expr), linear_shape(Expr),
147   linear_expr_subvar(Expr,Y), non_lin_var(Y).
```

The expression `Expr` is linear-shape, denoted as `linear_shape( Expr)` if `Expr` is

1. a constant expression (line 150), or
2. a variable (line 153), or
3. an addition or subtraction operation (denoted as `add_opcode(Opcode)`), the operands of which are linear expressions (lines 156–158), or
4. a multiplication operation, one operand of which is a constant literal, and the other one is a linear-shape expression (lines 162–164), or
5. an unary plus or unary minus operation (denoted as `uadd_opcode(Opcode)`), the operand of which is a linear expression (lines 167–168):

```
148 % Recursive definition of a linear-shape expression
149 %1. Constant
150 linear_shape(Expr) :- const_expr(Expr).
151
152 %2. Variable
153 linear_shape(X) :- var(X).
154
155 %3. Binary plus or minus, the operands of which are linear operations
156 linear_shape(Expr) :- bop_expr(Expr), opcode(Expr, Opcode),
157   add_opcode(Opcode), lhs_expr(Expr,LhsExpr), linear_shape(LhsExpr),
158   rhs_expr(Expr,RhsExpr), linear_shape(RhsExpr).
159
160 %4. Multiplication, one operand of which is a constant, and the other
      one
161 % is a linear operation
162 linear_shape(Expr) :- bop_expr(Expr), opcode(Expr, Opcode),
```

```
163   Opcode="*", operand(Expr,Op1), const_literal(Op1),
164   operand(Expr,Op2), Op1!=Op2, linear_shape(Op2).
165
166 %5. Unary plus or minus, the operand of which is a linear operation
167 linear_shape(Expr) :- uop_expr(Expr), opcode(Expr, Opcode),
168   uadd_opcode(Opcode), operand(Expr,Op), linear_shape(Op).
```

To check that a variable `X` is used in a linear expression `Expr`, we use a recursively defined binary relation `linear_expr_subvar`. Specifically, variable `X` is used in a linear expression `Expr` if one of the following cases holds:

- `Expr` is `X` (line 171);
- `Expr` is a binary or unary operator, which has an operand `Op` and variable `X` is used in `Op` (lines 174–175 and 178–179 respectively).

```
169 % Recursive definition of a linear operation Expr with variable X
170 %1. Variable (Expr=X)
171 linear_expr_subvar(X,X) :- var(X).
172
173 %2. A binary linear operation with operand Op which contains X
174 linear_expr_subvar(Expr,X) :- linear_shape(Expr), bop_expr(Expr),
175   operand(Expr,Op), linear_expr_subvar(Op,X).
176
177 %3. An unary linear operation with operand Op which contains X
178 linear_expr_subvar(Expr,X) :- linear_shape(Expr), uop_expr(Expr),
179   operand(Expr,Op), linear_expr_subvar(Op,X).
```

Example 3.3.19. Consider again the example in Fig. 3.11 on page 72.

The evaluation of the rule in line 171 infers the fact `linear_expr_subvar(`$\texttt{node}_i$`,`$\texttt{node}_i$`)`, and the evaluation of the rule in lines 174–175 infers the fact `linear_expr_subvar(70,`$\texttt{node}_i$`)` (recall that the node `70` corresponds to the addition operation `i+1`).

Then, the evaluation of the rule in line 118 infers the fact `const_expr(71)`, where the node `71` corresponds to the constant literal `1`).

Next, the evaluation of the rules in lines 150 and 153 infers the facts `linear_shape(71)` and `linear_shape(`$\texttt{node}_i$`)` respectively. Using these two facts, the evaluation of the rule in lines 162–164 infers the fact `linear_shape(70)`, and the evaluation of the rules in lines 139–147 infers that the transitive closure of the relation `non_lin_var` does not include the fact `non_lin_var(`$\texttt{node}_i$`)`. Finally, the evaluation of the rule in line 136 infers that the variable `i` is *linear*, encoded as `linear(`$\texttt{node}_i$`)`. ▲

3.3.2 Definition of Roles for a Portfolio Solver for Software Verification

We now give the definitions of the roles for the portfolio solver. We note that all the roles for the portfolio solver are defined *without negation*.

Roles without Negation

Thread descriptor. Variable `X` is *thread descriptor*, denoted as `thread_descr(X)`, if one of the following conditions hold:

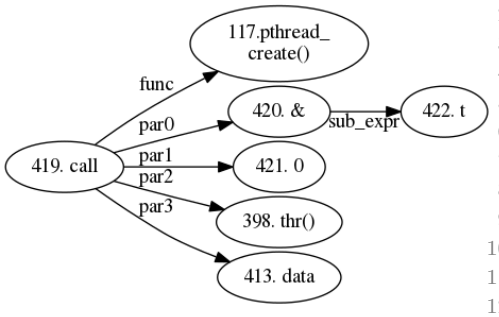
- `X` is passed by address as `I`-th argument to a function which initialises its `I`-th argument with a thread descriptor (denoted as `thread_descr_def_func(FuncName,I)`, lines 180–182 in the listing below), e.g. the function `pthread_create()`;
- `X` is assigned the result of a function which returns a thread descriptor (denoted as `thread_descr_def_ func(FuncName,fres)`, lines 184–185), e.g. the function `pthread_self()`;
- `X` is passed as `I`-th argument to a function which takes a thread descriptor as `I`-th argument (denoted as `thread_descr_use_func(FuncName,I)`, lines 187–188), e.g. the function `pthread_join()`.

```
180 thread_descr(X) :- var(X), act_arg(Func,I,Expr), name(Func,FuncName),
181   thread_descr_def_func(FuncName,I), uop_expr(Expr),
182   opcode(Expr,"ADDR_OF"), operand(Expr,X).
183
184 thread_descr(X) :- var(X), assigned_call(X,Func), name(Func,FuncName),
185   thread_descr_def_func(FuncName,fres).
186
187 thread_descr(X) :- var(X), act_arg(Func,I,X), name(Func,FuncName),
188   thread_descr_use_func(FuncName,I).
189
190 thread_descr(X) :- var(X), assigned(X,Y), thread_descr(Y).
```

We define the relations `thread_descr_def_func` and `thread_descr_use_func` in Section 8.A on page 163 in lines 150–151 and 141–147 respectively.

Example 3.3.20. Consider the example in Fig. 3.20a, which shows the CFG of the expression `pthread_create(&t,0,thr,data)` in Fig. 3.4a on page 51, line 19. The corresponding logic program is given in Fig. 3.20b.

The definition of the relation `thread_descr_def_func` includes the fact `thread_descr_def_func("pthread_create",0)`. The evaluation of the rule defining the relation `act_arg` (see Section 8.A, lines 17–21) infers the fact `act_arg(`$\mathtt{node}_{\mathit{pthread\_create}}$`,0,420)`, where the node `420` corresponds to the address-of operator `&t`. Then, the evaluation of the rule defining the relation `operand` (in Section 8.A, lines 10–11) infers the fact `operand(420,`$\mathtt{node}_t$`)`.

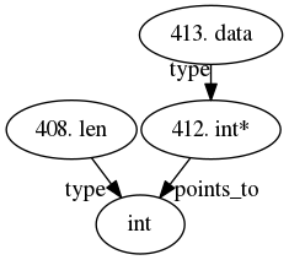


(a) Control flow graph

```
1 func_decl(117).
2 type(117,int).
3 name(117,
4  "pthread_create").
5 func_decl(398).
6 type(398,ptr).
7 name(398,"thr").
8 var(413).
9 type(413,ptr).
10 name(413,"data").
11 call_expr(419).
12 function(419,117).
13 param(419,0,420).
14 param(419,1,421).
15 param(419,2,398).
16 param(419,3,413).
17 const_literal(421).
18 type(421,int).
19 val(421,"0").
20 var(422).
21 type(422,int).
22 name(422,"t").
23 uop_expr(420).
24 sub_expr(420,422).
25 opcode(420,"ADDR_OF").
```

(b) Datalog program

Figure 3.20 Example for *thread descriptor*. The expression `pthread_create(&t,0,thr, data)` in line 19 of code in Fig. 3.4a on page 51 translated to Datalog.



(a) Graphical representation of the logic relations.

```
1 pointer_type(412).
2 points_to_type(412,int).
3 var(413).
4 type(413,412).
5 name(413,"data").
6 var(408).
7 type(408,int).
8 name(408,"len").
```

(b) Logic program.

Figure 3.21 Example for *scalar* and *pointer to scalar*. The definitions `int len` and `int* data` in lines 10 and 15 of code in Fig. 3.4a on page 51 translated to Datalog.

Finally, using these facts, the evaluation of the rule in lines 180–182 infers that the variable `t` is *thread descriptor*, encoded as `thread_descr(node`$_t$`)`. ▲

Integral. Variable `X` is *integral* (denoted as `scalar_int(X)`, line 191 in the listing below) if `X` is of integral type.

```
191 scalar_int(X) :- var(X), type(X,Type), integral_type(Type).
```

Specifically, the relation `integral_type(Type)`, defined in Section 8.A, lines 275–276, encodes that the type `Type` is an integral type.

Example 3.3.21. Consider the example in Fig. 3.21b, which shows the translation to a logic program of variable definitions `int len` and `int* data` in Fig. 3.4a on page 51, lines 10 and 15 respectively.

We show a graphical representation of the logic relations in Fig. 3.21a. Specifically, we depict the relations **var**, **pointer_type** and the constant term `int` as nodes, and the relations **type** and **points_to_type** as edges.

For this example, the evaluation of the rule in line 191 infers that the variable `len` is *integral*, encoded as `scalar_int(node`$_{len}$`)`. ▲

Scalar. Variable `X` is *scalar* (denoted as `scalar(X)`, line 192 in the listing below) if `X` is of scalar (i.e. integral or floating-point) type.

```
192 scalar(X) :- var(X), type(X,Type), scalar_type(Type).
```

Specifically, the relation `scalar_type(Type)`, which we define in Section 8.A, lines 282–283, encodes that `Type` is an integer or floating-point type.

Example 3.3.22. For the example in Fig. 3.21, the evaluation of the rule in line 192 infers that the variable `len` is *scalar*, encoded as `scalar(node`$_{len}$`)`. ▲

Pointer to scalar. Variable `X` is *pointer to scalar* (denoted as `ptr_scalar(X)`, lines 193–194 in the listing below) if the type of `X` is pointer to a scalar type.

```
193 ptr_scalar(X) :- var(X), type(X,PtrType), pointer_type(PtrType),
194   points_to_type(PtrType,ScalarType), scalar_type(ScalarType).
```

Specifically, the term **pointer_type**`(PtrType)` denotes that `PtrType` is a pointer type and the term **points_to_type**`(PtrType,ScalarType)` denotes that the pointer type `PtrType` points to the type `ScalarType`.

Example 3.3.23. For the program in Fig. 3.21, the evaluation of the rule in lines 193–194 infers that the variable `data` is *scalar pointer*, encoded as `ptr_scalar(node`$_{data}$`)`. ▲

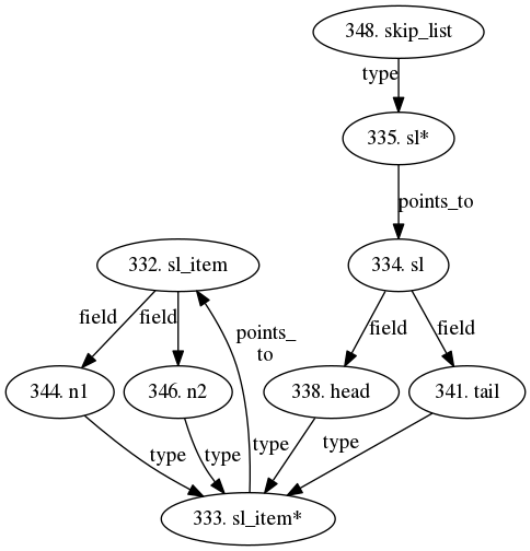
Pointer to structure. Variable `X` is *pointer to structure* (denoted as `ptr_struct(X)`, lines 195–196 in the listing below) if the type of `X` is a pointer to structure.

```
195 ptr_struct(X) :- var(X), type(X,PtrType), pointer_type(PtrType),
196   points_to_type(PtrType,StructType), structure_type(StructType).
```

Specifically, the term **structure_type**`(StructType)` encodes that `StructType` is a structure type.

Example 3.3.24. Consider the example in Fig. 3.22a, which shows the CFG of the definitions of the structures `sl_item` and `sl` and the variable `skip_list` in Fig. 3.4b on page 51. The corresponding logic program is given in Fig. 3.22b.

For this program, the evaluation of the rule in lines 195–196 infers that the variable `skip_list` and structure fields `sl_item.n1`, `sl_item.n2`, `sl.head` and



(a) Graphical representation of the logic relations.

```
1 structure_type(332).
2 name(332,"sl_item").
3 pointer_type(333).
4 points_to_type(333,332).
5 structure_type(334).
6 name(334,"sl").
7 pointer_type(335).
8 points_to_type(335,334).
9 var(338).
10 field(334,338).
11 type(338,333)
12 name(338,"head").
13 var(341).
14 field(334,341).
15 type(341,333).
16 name(341,"tail").
17 var(344).
18 field(332,344).
19 type(344,333).
20 name(344,"n1").
21 var(346).
22 field(332,346).
23 type(346,333).
24 name(346,"n2").
25 var(348).
26 type(348,335).
27 name(348,"skip_list").
```

(b) Logic program.

Figure 3.22 Example for *pointer to structure*, *pointer to non-flat structure*, *linked list*, *multi-linked list*. The definitions of the structures `sl_item` and `sl` and of the variable `skip_list` of the code in Fig. 3.4b on page 51 translated to Datalog.

`sl.tail` have role *pointer to structure*, encoded as `ptr_struct(`node_{skip_list}`)`, `ptr_struct(`node_{n1}`)`, `ptr_struct(`node_{n2}`)`, `ptr_struct(`node_{head}`)` and `ptr_struct(`node_{tail}`)` respectively. ▲

Pointer to non-flat structure. Variable `X` is *pointer to non-flat structure* (denoted as `ptr_non_flat_struct(X)`, lines 197–200 in the listing below) if the type of `X` is pointer to type `StructType`, s.t. `StructType` is a structure with at least one pointer field.

```
197 ptr_non_flat_struct(X):- var(X), type(X,PtrType),
198   pointer_type(PtrType), points_to_type(PtrType,StructType),
199   structure_type(StructType), field(StructType,Field),
200   type(Field,FieldPtrType), pointer_type(FieldPtrType).
```

Specifically, the term **field**(StructType,Field) encodes that structure StructType has a field Field.

Example 3.3.25. Consider again the example in Fig. 3.22.

For this program, the evaluation of the rule in lines 197–200 infers that the variable skip_list and structure fields sl_item.n1, sl_item.n2, sl.head and sl.tail have role *pointer to non-flat structure*, encoded as ptr_non_flat_struct(node$_{skip_list}$), ptr_non_flat_struct(node$_{n1}$), ptr_non_flat_struct(node$_{n2}$), ptr_non_flat_struct(node$_{head}$) and ptr_non_flat_struct(node$_{tail}$) respectively. ▲

Linked list. Variable X is *linked list* (denoted as linked_list(X), lines 201–203 in the listing below) if the type of X is pointer to a recursively defined structure.

```
201 linked_list(X) :- var(X), type(X,PtrType), pointer_type(PtrType),
202   points_to_type(PtrType,StructType), structure_type(StructType),
203   field(StructType,Field), type(Field,PtrType).
```

Example 3.3.26. Consider the program in Fig. 3.22.

For this program, the evaluation of the rule in lines 197–200 infers that the structure fields sl_item.n1, sl_item.n2, sl.head and sl.tail have role *linked list*, encoded as linked_list(node$_{n1}$), linked_list(node$_{n2}$), linked_list(node$_{head}$) and linked_list(node$_{tail}$) respectively. ▲

Multi-linked list. Variable X is *multi-linked list* (denoted as multi_linked_list(X), lines 204–207 in the listing below) if the type PtrType of X is pointer to a structure type StructType, s.t. StructType has at least two fields of type pointing to PtrType.

```
204 multi_linked_list(X):- var(X), type(X,PtrType), pointer_type(PtrType),
205   points_to_type(PtrType,StructType), structure_type(StructType),
206   field(StructType,Field1), type(Field1,PtrType),
207   field(StructType,Field2), type(Field2,PtrType), Field1!=Field2.
```

Example 3.3.27. Consider the program in Fig. 3.22.

For this program, the evaluation of the rule in lines 204–207 infers that the structure fields sl_item.n1, sl_item.n2, sl.head and sl.tail have role *multi-linked list*, encoded as multi_linked_list(node$_{n1}$), multi_linked_list(node$_{n2}$), multi_linked_list(node$_{head}$) and multi_linked_list(node$_{tail}$) respectively. ▲

Pointer. Variable X is *pointer* (denoted as ptr(X), line 208 in the listing below) if the type PtrType of X is pointer type.

```
208 ptr(X) :- var(X), type(X,PtrType), pointer_type(PtrType).
```

Example 3.3.28. Consider the program in Fig. 3.22.

For this program, the evaluation of the rule in lines 195–196 infers that the variable `skip_list` and the structure fields `sl_item.n1`, `sl_item.n2`, `sl.head` and `sl.tail` have role *pointer*, encoded as `ptr(node`$_{skip\_list}$`)`, `ptr(node`$_{n1}$`)`, `ptr(node`$_{n2}$`)`, `ptr(node`$_{head}$`)` and `ptr(node`$_{tail}$`)` respectively. ▲

Dynamic memory pointer. Variable `X` is *dynamic memory pointer* (denoted as `dyn_mem_ptr(X)`), if at least one of the following conditions holds:

- `X` is assigned the result of a call to a memory allocation function `Func`, e.g. `malloc()`, `calloc()` etc (lines 209–210 in the listing below).

  We denote such functions with the relation `dyn_mem_def_func(FuncName, fres)`, which we define in Section 8.A, lines 167–174;
- `X` is passed as `I`-th parameter to a standard memory manipulation function which takes as `I`-th parameter a pointer to dynamically allocated memory, e.g. memory deallocation function `free()` (lines 212–213 in the listing below).

  We denote such functions with the relation `dyn_mem_use_func(FuncName,I)`, which we define in Section 8.A, lines 155–164.

```
209 dyn_mem_ptr(X) :- var(X), assigned_call(X,Func), name(Func,FuncName),
210   dyn_mem_def_func(FuncName,fres).
211
212 dyn_mem_ptr(X) :- var(X), act_arg(Func,I,X), name(Func,FuncName),
213   dyn_mem_use_func(FuncName,I).
214
215 dyn_mem_ptr(X) :- var(X), assigned(X,Y), dyn_mem_ptr(Y).
```

Example 3.3.29. Consider the program in Fig. 3.16 on page 80.

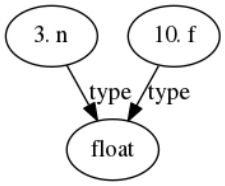
For this program, the evaluation of the rule defining the relation `assigned_call` in Section. 8.A on page 163, lines 24–25 infers the fact `assigned_call(node`$_{data}$`, node`$_{malloc}$`)`. Then, the definition of the relation `dyn_mem_def_func` contains the fact
`dyn_mem_def_func("malloc",fres).`

Finally, the evaluation of the rule in lines 209–210 infers that the variable `data` is *dynamic memory pointer*, encoded as `dyn_mem_ptr(node`$_{data}$`)`. ▲

Floating point. Variable `X` is *floating point* (denoted as `scalar_float(X)`, line 216 in the listing below) if the type `Type` of `X` is a floating-point type.

```
216 scalar_float(X) :- var(X), type(X,Type), floating_point_type(Type).
```

Specifically, the relation `floating_point_type(Type)`, which we define in Section 8.A, line 279, encodes that the type `Type` is a floating-point type.



(a) Graphical representation of the logic relations.

```
1 var(3).
2 type(3,float).
3 name(3,"n_2_10").
4 var(10).
5 type(10,float).
6 name(10,"f").
```

(b) Datalog program

Figure 3.23 Example for *floating point*. The variable definitions `float n` and `float f` in lines 1 and 3 respectively of code in Fig. 3.5b on page 54 translated to Datalog.

Example 3.3.30. Consider example in Fig. 3.23a, which shows the graphical representation of the logic relation of the variable definitions `float n` and `float f` in lines 1 and 3 respectively of code in Fig. 3.5b on page 54.

The definition of the relation `floating_point` includes the fact `floating_point_type(float)`. Then, for this program the evaluation of the rule in line 216 infers that the variables `n` and `f` have the role *floating point*, encoded with the facts `scalar_float(node`$_n$`)` and `scalar_float(node`$_f$`)` respectively. ▲

Recursive function result. We will define the role *recursive function result* in Section 3.4.4, after we describe an extension of our framework to inter-procedural analysis.

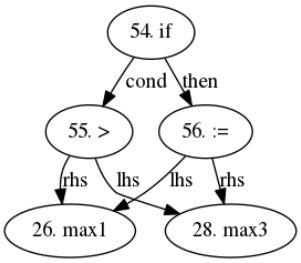
3.3.3 Definition of Roles for Heuristics in Predicate Abstraction

Finally, we give the definitions of the roles for heuristics in predicate abstraction. We again split the definitions to two groups depending on whether a role is defined using negation.

Roles without Negation

Extremum. Variable `X` is *extremum* (denoted as `extremum(X)`), if the following conditions hold:

- There is an if statement `IfStmt`, the condition `Cond` of which is the operator greater-than > or less-than < (encoded with the relation `strict_rel_opcode(Opcode)`, which we define in Section 8.A on page 163, lines 211–212);
- An operand of `IfStmt` is a variable `Y`;
- There is an assignment statement `AsgStmt` which is a sub-statement of `IfStmt` and `X` is assigned `Y` in `AsgStmt` (lines 217–220);



(a) Control flow graph

```
1 var(26).
2 type(26,int).
3 name(26,"max1").
4 var(28).
5 type(28,int).
6 name(28,"max3").
7 if_stmt(54).
8 condition(54,55).
9 then_stmt(54,56).
10 bop_expr(55).
11 lhs_expr(55,28).
12 rhs_expr(55,26).
13 opcode(55,">").
14 assignment_stmt(56).
15 lhs_expr(56,26).
16 rhs_expr(56,28).
```

(b) Datalog program

Figure 3.24 Example for *extremum*. Statement `if (max3>max1) max1=max3;` of code in Fig. 3.6a on page 57, line 17 translated to Datalog.

```
217 extremum(X) :- if_stmt(IfStmt), condition(IfStmt,Cond),
218   bop_expr(Cond), opcode(Cond,Opcode), strict_rel_opcode(Opcode),
219   var(Y), operand(Cond,Y), assignment_stmt(AsgStmt),
220   sub_stmt(IfStmt,AsgStmt), lhs_expr(AsgStmt,X), rhs_expr(AsgStmt,Y).
221
222 extremum(X) :- var(X), assigned(X,Y), extremum(Y).
```

We define the role `extremum` without parameters (which could be, e.g., a pair of values of which an extremum is taken), since in the heuristic for Eldarica in which we use this role, we do not need this information (see Section 5.2).

Example 3.3.31. Consider the example in Fig. 3.24a, which shows the CFG of the statement `if (max3>max1) max1=max3;` of code in Fig. 3.6a, line 17. The corresponding logic program is given in Fig. 3.24b.

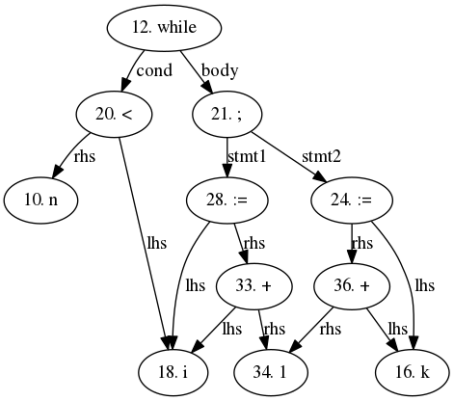
The evaluation of the rule defining the relation `operand` (see Section 8.A on page 163, lines 10–11) infers the fact `operand(55,`$\texttt{node}_{max1}$`)`, where the node `55` corresponds to the comparison operation `max1>max3`. Then, the evaluation of the rule defining the relation `sub_stmt` (see Section 8.A, lines 59–74) infers the fact `sub_stmt(54,56)`, where the node `54` corresponds to the `if` statement, and the node `56` corresponds to the assignment statement `max1=max3`.

Using these facts, the evaluation of the rule in lines 217–220 infers that the variable `max3` is *extremum*, encoded as `extremum(`$\texttt{node}_{max3}$`)`. ▲

Local counter. Similarly to *loop counter* and *loop bound*, role *local counter* is defined in the scope of a loop `WhileStmt`.

Variable `X` is *local counter* in `WhileStmt` (denoted as `local_cnt(X,WhileStmt)`) if at least one of the following conditions holds:

- There is a loop statement `WhileStmt` and an assignment statement `AsgStmt`, s.t. `AsgStmt` is a sub-statement of `WhileStmt`; and in `AsgStmt` `X` is assigned an expression `Expr`, and `Expr` is the sum or the difference of `X` and some other expression.



(a) Control flow graph

```
1 var(10).
2 type(10,int).
3 name(10,"n").
4 var(18).
5 type(18,int).
6 name(18,"i").
7 var(16).
8 type(16,int).
9 name(16,"k").
10 while_stmt(12).
11 condition(12,20).
12 body(12,21).
13 bop_expr(20).
14 lhs_expr(20,18).
15 rhs_expr(20,10).
16 opcode(20,"<").
17 sequence_stmt(21).
18 stmt1(21,28).
19 stmt2(21,24).
20 assignment_stmt(28).
21 lhs_expr(28,18).
22 rhs_expr(28,33).
23 const_literal(34).
24 type(34,int).
25 val(34,"1").
26 bop_expr(33).
27 lhs_expr(33,18).
28 rhs_expr(33,34).
29 opcode(33,"+").
30 assignment_stmt(24).
31 lhs_expr(24,16).
32 rhs_expr(24,36).
33 bop_expr(36).
34 lhs_expr(36,16).
35 rhs_expr(36,34).
36 opcode(36,"+").
```

(b) Datalog program

Figure 3.25 Example for *local counter*. The statement `for (k=0, i=0; i<n; i++, k++);` of code in Fig. 1.2, line 8, translated to Datalog.

The last fact is encoded with the term `cnt_expr(Expr,X)`, recall the definition of the relation `cnt_expr` in Section 3.3.1 on page 86, lines 134–135.

– There is an assignment statement `AsgStmt` which is a sub-statement of the loop statement `WhileStmt`, s.t. in `AsgStmt` the variable `X` is assigned a variable `Y` which is a *local counter* in `WhileStmt`.

```
223 local_cnt(X,WhileStmt) :- var(X), while_stmt(WhileStmt),
224   assignment_stmt(AsgStmt), sub_stmt(WhileStmt,AsgStmt),
225   lhs_expr(AsgStmt,X), rhs_expr(AsgStmt,Expr), cnt_expr(Expr,X).
226
227 local_cnt(X,WhileStmt) :- var(X), assignment_stmt(AsgStmt),
228   sub_stmt(WhileStmt,AsgStmt), lhs_expr(AsgStmt,X),
229   rhs_expr(AsgStmt,Y), local_cnt(Y,WhileStmt).
```

The parameter `WhileStmt` of the relation `local_cnt` is needed for two reasons. First, we use the parameter in the rule in lines 227–229 to compute the transitive closure of *local counters*, similarly to `loop_counter` and `loop_bound`. Second, we use the parameter in a heuristic for ELDARICA to define a predicate template for two *local counters* located in the same loop (see Section 5.2).

Example 3.3.32. Consider the example in Fig. 3.25a, which shows the CFG of the statement `for (k=0, i=0; i<n; i++, k++);` of code in Fig. 1.2, line 8. In particular, the statement is rewritten by our algorithm to the sequence of statements `k=0; i=0; while(i<n) {i=i+1; k=k+1;}`. The corresponding logic program is shown in Fig. 3.25b.

For this program, the evaluation of the rule defining the relation `sub_stmt` in Section 8.A, lines 59–59 infers the fact `sub_stmt(12, 28)`, where the node `12` corresponds to the loop statement, and the node `28` – to the assignment statement `i=i+1`.

Then, the evaluation of the rule defining the relation `cnt_expr` infers the fact `cnt_expr(33,node`$_i$`)`, where the node `33` corresponds to the expression `i+1`.

Finally, the evaluation of the rule in lines 223–225 infers that the variable `i` is *local counter*, encoded as `local_cnt(node`$_i$`)`.

Using similar reasoning, the evaluation of the rule in lines 223–225 infers that the variable `k` is *local counter*, encoded as `local_cnt(node`$_k$`)`. ▲

Parity variable. Variable `X` is *parity variable* with constant literal parameter `Num` (denoted as `parity(X,Num)`) if at least one of the following conditions holds:

- There is there is a binary remainder operation `Expr`, s.t. `X` is an operand of `Expr`, and the other operand of `Expr` is `Num` (lines 230–231 in the listing below);

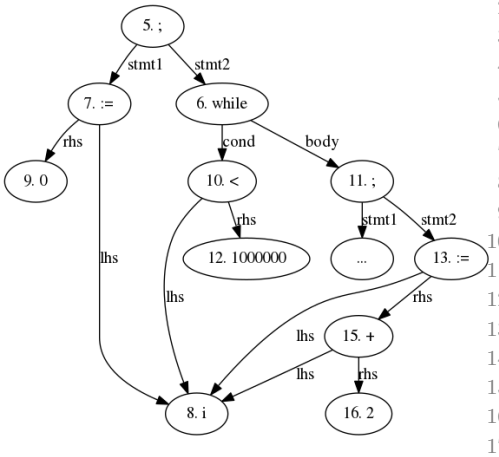
  Specifically, the operation code `Opcode` of the binary remainder operation is encoded as `opcode(Opcode,"REM")`.
- There is a `while` statement `WhileStmt` and an assignment statement `AsgStmt`, s.t. in `AsgStmt` the variable `X` is assigned an expression `Expr`, and `Expr` is either a sum or a difference of `X` and `Num`, encoded as `inc_expr(Expr, X,Num)` (lines 233–235).

```
230 parity(X,Num) :- var(X), const_literal(Num), bop_expr(Expr),
231   opcode(Expr,"REM"), operand(Expr,X), operand(Expr,Num).
232
233 parity(X,Num) :- var(X), const_literal(Num), while_stmt(WhileStmt),
234   assignment_stmt(AsgStmt), sub_stmt(WhileStmt,AsgStmt),
235   lhs_expr(AsgStmt,X), rhs_expr(AsgStmt,Expr), inc_expr(Expr,X,Num).
236
237 parity(X, Num) :- var(X), assigned(X,Y), parity(Y,Num).
```

The term `inc_expr(Expr,X,IncExpr)` is derived if one of the following cases holds:

- The expression `Expr` is a binary addition operation, of which the variable `X` and the expression `IncExpr` are distinct operands (lines 238–239);
- `Expr` is a binary subtraction operation, of which the variable `X` is the left-hand side expression, and the expression `IncExpr` is the right-hand side expression (lines 241–242).

```
238 inc_expr(Expr,X,IncExpr) :- bop_expr(Expr), opcode(Expr,"+"),
239   operand(Expr,X), operand(Expr,IncExpr), X!=IncExpr.
```



(a) Control flow graph

```
1 sequence_stmt(5).
2 stmt1(5,7).
3 stmt2(5,6).
4 var(8).
5 type(8,int).
6 name(8,"i").
7 const_literal(9).
8 type(9,int).
9 val(9,"0").
10 assignment_stmt(7).
11 lhs_expr(7,8).
12 rhs_expr(7,9).
13 while_stmt(6).
14 condition(6,10).
15 body(6,11).
16 const_literal(12).
17 type(12,int).
18 val(12,"1000000").
19 bop_expr(10).
20 lhs_expr(10,8).
21 rhs_expr(10,12).
22 opcode(10,"<").
23 sequence_stmt(11).
24 stmt1(11,14).
25 stmt2(11,13).
26 assignment_stmt(13).
27 lhs_expr(13,8).
28 rhs_expr(13,15).
29 const_literal(16).
30 type(16,int).
31 val(16,"2").
32 bop_expr(15).
33 lhs_expr(15,8).
34 rhs_expr(15,16).
35 opcode(15,"+").
```

(b) Datalog program

Figure 3.26 Example for *parity*. The statement `for (i=0; i<1000000; i+=2);` of code in Fig. 3.6c on page 57, line 5, translated to Datalog.

```
240
241 inc_expr(Expr,X,IncExpr) :- bop_expr(Expr), opcode(Expr,"-"),
242   lhs_expr(Expr,X), rhs_expr(Expr,IncExpr).
```

Example 3.3.33. Consider the example in Fig. 3.26a, which shows the CFG of the statement `for (i=0; i<1000000; i+=2);` of code in Fig. 3.6c on page 57, line 5. The corresponding logic program is given in Fig. 3.26b.

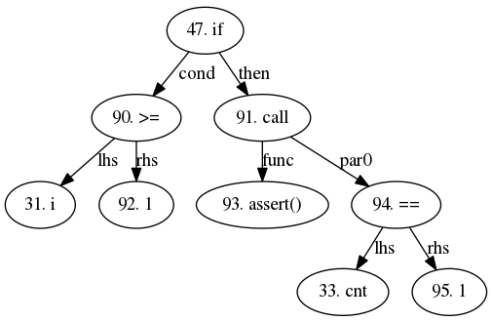
For this program, the evaluation of the rule defining the relation `sub_stmt` (Section 8.A, lines 59–59 infers the fact `sub_stmt(6, 13)`, where the node `6` corresponds to the loop statement, and the node `13` – to the assignment statement `i=i+2`.

Then, the evaluation of the rule defining the relation `sub_stmt` (Section 8.A, lines 59–59 infers the fact `sub_stmt(6, 13)`, where the node `6` corresponds to the loop statement, and the node `13` – to the assignment statement `i=i+2`.

Finally, using these facts, the evaluation of the rule in lines 233–235 infers that the variable `i` is *parity variable* with parameter `2`, encoded as `parity(node`$_i$`,16)`. ▲

Assertion parameter. Variable `X` is *assertion parameter* of expression `Expr` (denoted as `assert_param(X,Expr)`, lines 243–244) if there is an assertion condition `Expr`, encoded as `assert_cond(Expr)`, which has a literal `Lit` (denoted as `literal(Expr,Lit)`, s.t. `X` is atom of `Lit` (denoted as `atom(Expr,X)`).

```
243 assert_param(X,Expr) :- var(X), assert_cond(Expr),
```



(a) Control flow graph

```
1 var(31).
2 type(31,int).
3 name(31,"i").
4 var(33).
5 type(33,int).
6 name(33,"cnt").
7 if_stmt(47).
8 condition(47,90).
9 then_stmt(47,91).
10 const_literal(92)
11 type(92,int).
12 val(92,"1").
13 bop_expr(90).
14 lhs_expr(90,31).
15 rhs_expr(90,92).
16 opcode(90,">=").
17 func_decl(93).
18 type(93,int).
19 name(93,"assert").
20 call_expr(91).
21 function(91,93).
22 param(91,0,94).
23 const_literal(95).
24 type(95,int).
25 val(95,"1").
26 bop_expr(94).
27 lhs_expr(94,33).
28 rhs_expr(94,95).
29 opcode(94,"==").
```

(b) Datalog program

Figure 3.27 Example for *assertion condition*. The statement `if (i>=1) assert(cnt==1);` of code in Fig. 3.6a on page 57, line 27 translated to Datalog.

```
244   literal(Expr,Lit), atom(Lit,X).
245
246 assert_param(X,Expr) :- var(X), assigned(X,Y), assert_param(Y,Expr).
```

An atom of an expression `Expr`, which we define with the relation `atom` in Section 8.A on page 163, lines 46–47), is an arithmetic operand of `Expr` which is not an arithmetic operation itself.

An expression `Expr` is assertion condition, encoded as `assert_cond(Expr)`, if at least one of the following conditions holds:

- The expression `Expr` is passed as `I`-th argument to an assertion function, e.g. `assert()` (lines 247–248 in the listing below).

  The relation `assert_func(FuncName,I)` encodes that `FuncName` is an assertion function. We define the relation in Section 8.A on page 163, line 189).

- There is a call expression `CallExpr`, in which an assertion function `Func` is called, and there is an `if` statement `IfStmt` of which `CallExpr` is a sub-statement, and the condition of `IfStmt` is `Expr` (lines 250–252).

```
247 assert_cond(Expr) :- act_arg(Func,I,Expr), name(Func,FuncName),
248   assert_func(FuncName,I).
249
250 assert_cond(Expr) :- called(Func,CallExpr), name(Func,FuncName),
251   assert_func(FuncName,I), if_stmt(IfStmt), sub_stmt(IfStmt,CallExpr),
252   condition(IfStmt,Expr).
```

Example 3.3.34. Consider the example in Fig. 3.27a, which shows the CFG of the statement `if (i>=1) assert( cnt==1);` of code in Fig. 3.6a on page 57, line 27. The corresponding logic program is given in Fig. 3.27b.

The evaluation of the rule in lines 247–248 infers the fact `assert_cond(94)`, where the node `94` corresponds to the expression `cnt==1`. Then, the evaluation of the rule defining the relation `atom` (see Section 8.A, lines 46–47) infers the facts `atom(94,`$\texttt{node}_{cnt}$`)` and `atom(94,95)`, where the node `95` corresponds to a constant literal `1`. Using these facts, the evaluation of the rule in lines 243–244 infers that the variable `cnt` is *assertion parameter* of expression `cnt==1`, encoded as `assert_param(`$\texttt{node}_{cnt}$`,94)`.

Similarly, the evaluation of the rule in lines 250–252 infers the fact `assert_cond(90)`, where the node `90` corresponds to the expression `i>=1`. The evaluation of the rule defining the relation `atom` infers the facts `atom(90,`$\texttt{node}_i$`)` and `atom(90,92)`, where the node `92` corresponds to a constant literal `1`. Finally, given these two facts, the evaluation of the rule in lines 243–244 infers that the variable `i` is *assertion parameter* of expression `i>=1`, encoded as `assert_param(`$\texttt{node}_i$`, 90)`. ▲

Roles with Negation

Dynamic enumeration. Variable `X` is *dynamic enumeration* with parameter variable `Y` (denoted as `dyn_enum(X, Y)`, lines 253–254) if the following conditions hold:

- `X` is not in the relation `not_dyn_enum(X)` and
- `X` is transitively assigned a variable `Y` (encoded as `assigned_tc( X,Y)`), s.t. `Y` is *input*.

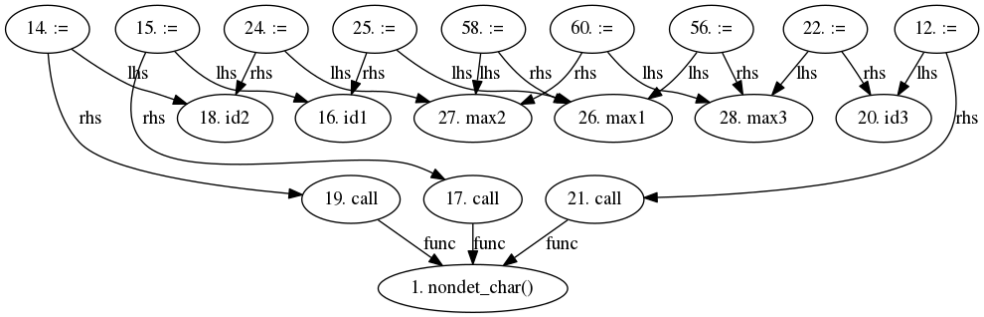
```
253 dyn_enum(X,Y) :- var(X), not non_dyn_enum(X), assigned_tc(X,Y),
254   input(Y).
```

The relation `assigned_tc(X,Expr)` is defined in Section 8.A on page 163, lines 6–7 and computes the transitive closure of expressions `Expr` assigned to the variable `X`.

A variable `X` is included in the relation `non_dyn_enum` (lines 255) if `X` is transitively assigned an expression `Expr` which is not in the relation `dyn_enum_expr(Expr)`.

```
255 non_dyn_enum(X) :- assigned_tc(X,Expr), not dyn_enum_expr(Expr)
```

An expresion `Expr` is in the relation `dyn_enum_expr` if `Expr` is one of the following:
1) a variable (line 256),
2) a constant literal (line 257),
3) an input variable (line 258).



(a) Control flow graph

```
1 func_decl(1).
2 name(1,"nondet_char").
3 var(16).
4 name(16,"id1").
5 assignment_stmt(15).
6 lhs_expr(15,16).
7 rhs_expr(15,17).
8 call_expr(17).
9 function(17,1).
10 var(18).
11 name(18,"id2").
12 assignment_stmt(14).
13 lhs_expr(14,18).
14 rhs_expr(14,19).
15 call_expr(19).
16 function(19,1).
17 var(20).
18 name(20,"id3").
19 assignment_stmt(12).
20 lhs_expr(12,20).
21 rhs_expr(12,21).
22 call_expr(21).
23 function(21,1).
24 var(26).
25 name(26,"max1").
26 assignment_stmt(25).
27 lhs_expr(25,26).
28 rhs_expr(25,16).
29 var(27).
30 name(27,"max2").
31 assignment_stmt(24).
32 lhs_expr(24,27).
33 rhs_expr(24,18).
34 var(28).
35 name(28,"max3").
36 assignment_stmt(22).
37 lhs_expr(22,28).
38 rhs_expr(22,20).
39 assignment_stmt(56).
40 lhs_expr(56,26).
41 rhs_expr(56,28).
42 assignment_stmt(58).
43 lhs_expr(58,27).
44 rhs_expr(58,26).
45 assignment_stmt(60).
46 lhs_expr(60,28).
47 rhs_expr(60,27).
```

(b) Datalog program

Figure 3.28 Example for *dynamic enumeration*. The statements `id1=nondet_char();`, `id2=nondet_char();`, `id3=nondet_char();` of code in Fig. 3.6a on page 57, lines 4–6, the statements `max1=id1;`, `max2=id2;`, `max3=id3;` of same program, line 10, and the statements `max1=max3;`, `max2=max1;` and `max3=max2;` of same program, lines 17–19 respectively, translated to Datalog.

```
256 dyn_enum_expr(Expr) :- var(Expr).
257 dyn_enum_expr(Expr) :- const_literal(Expr).
258 dyn_enum_expr(Expr) :- call_expr(Expr).
```

Example 3.3.35. Consider the example in Fig. 3.28a, which shows the CFG of the following statements:

- The statements `id1=nondet_char();`, `id2=nondet_char();`, `id3=nondet_char();` of code in Fig. 3.6a on page 57, lines 4–6;
- The statements `max1=id1;`, `max2=id2;`, `max3=id3;` of same program in line 10;
- The statements `max1=max3;`, `max2=max1;` and `max3=max2;` of same program in lines 17–19.

The corresponding logic program is given in Fig. 3.28a. We omit the information about types to keep the CFG readable.

First, the evaluation of the rule defining the relation `assigned_tc()` in Section 8.A, lines 6–7 infers for the variable `max1` the facts `assigned_tc(node`$_{max1}$`, node`$_{id1}$`)`, `assigned_tc(node`$_{max1}$`,node`$_{id2}$`)`, `assigned_tc(node`$_{max1}$`, node`$_{id3}$`)`, `assigned_tc(node`$_{max1}$`,node`$_{max1}$`)`, `assigned_tc(node`$_{max1}$`, node`$_{max2}$`)`, `assigned_tc(node`$_{max1}$`,node`$_{max3}$`)`.

Next, the evaluation of the rule in line 256 infers the facts `dyn_enum_expr(node`$_{id1}$`)`, `dyn_enum_expr(node`$_{id2}$`)`, `dyn_enum_expr(node`$_{id3}$`)`, `dyn_enum_expr(node`$_{max1}$`)`, `dyn_enum_expr(node`$_{max2}$`)`, `dyn_enum_expr(node`$_{max3}$`)`.

Using these facts, the evaluation of the rule in line 255 infers that the transitive closure of the relation `non_dyn_enum()` does not include the fact `non_dyn_enum(max1)`.

In addition, using the reasoning explained in Section 3.3.1, the evaluation of the rule in line 38 on page 75 defining the role *input* infers that the variables `id1`, `id2` and `id3` are *inputs*, encoded as `input(node`$_{id1}$`)`, `input(node`$_{id2}$`)` and `input(node`$_{id3}$`)`.

Finally, the evaluation of the rule in lines 253–254 infers that the variable `max1` is *dynamic enumeration* with parameters `id1`, `id2` and `id3`, encoded with the facts `dyn_enum(node`$_{max1}$`, node`$_{id1}$`)`, `dyn_enum(node`$_{max1}$`, node`$_{id2}$`)` and `dyn_enum(node`$_{max1}$`, node`$_{id3}$`)`.

Using a similar reasoning, the evaluation of the rule in lines 253–254 infers that the variables `max2` and `max3` have the role *dynamic enumeration* with parameters `id1`, `id2` and `id3`, encoded with the facts `dyn_enum(node`$_{max2}$`, node`$_{id1}$`)`, `dyn_enum(node`$_{max2}$`, node`$_{id2}$`)` and `dyn_enum(node`$_{max2}$`, node`$_{id3}$`)` for `max2` and the facts `dyn_enum(node`$_{max3}$`, node`$_{id1}$`)`, `dyn_enum(node`$_{max3}$`, node`$_{id2}$`)` and `dyn_enum(node`$_{max3}$`, node`$_{id3}$`)` for `max3`.

3.4 Extension to Inter-Procedural Analysis

In this section we extend the analyses presented in Sections 3.3.1, 3.3.2 and 3.3.3 to inter-procedural analysis.

3.4.1 Syntax of C Language

We first extend the syntax of the subset of C language handled by our algorithm from Fig. 3.7, with new rules shown in Fig. 3.29. In particular, we add a rule for a function definition and a `return` statement.

	Grammar rules	*Explanation of the rules*
	Definitions	
d ::=	$\ldots$ |	rules for definitions from Fig. 3.7.
	$t\ f(t_1\ id_1, \ldots, t_n\ id_n)\ \{s\}$ |	function definition
	Statements	
s ::=	$\ldots$ |	rules for statements from Fig. 3.7
	return e	`return` statement

Figure 3.29 Extention to the syntax of the subset C language defined in Fig. 3.7 with functions.. Recall that the terminals id_i and f are variable and function identifiers respectively, and the non-terminals t and e correspond to a data type and an expression respectively

Table 3.6 Translation of the C constructs from Fig. 3.29 to Datalog.

C declaration		**Translation to logic program**
C construct	**Syntactic rule**, d ::=	

Function definitions	$t_0\ f(t_1\ p_1, \ldots, t_n\ p_n)\ \{s\}$	`func_decl(`$node_f$`)`
		`name(`$node_f, f$`)`
		`type(`$node_f, t_0$`)`
		`body(`$node_f, node_b$`)`
		
		`param(`$node_f$`,0,`$node_{p_1}$`)`
		`type(`$node_{p_1}$`,`t_1`)`
		...
		`param(`$node_f$`,n-1,`p_n`)`
		`type(`$node_{p_n}$`,`t_n`)`
		
		`res_node(`$node_f$`,`$node_{f\_res}$`)`
		`var(`$node_{f\_res}$`)`
		`type(`$node_{f\_res}, t_0$`)`

(a) Definitions

The node $node_{f\_res}$ corresponds to a pseudo-variable for the result of the function f.

C statement		**Translation to logic program**
C construct	**Syntactic rule**, s ::=	

`return` statement	*return e*	`return_stmt(`$node_s$`)`
		`sub_expr(`$node_s, node_e$`)`
		`function(`$node_s, node_f$`)`

(b) Statements.

The node $node_f$ corresponds to the function f hosting the `return` statement.

3.4.2 Datalog Relations in Program Translation

Next, we show in Table 3.6 how the two new grammar constructs from Fig. 3.29 are translated to Datalog.

For the convenience of the reader we remind the meaning of the following relations:

- The relation `func_decl(`$node_f$`)` denotes that the node $node_f$ corresponds to a

function.

- The relations `name(`$node_f$`, f)` and `type(`$node_f$`, `t_0`)` denote that the name and the type of the function (or variable) at the node $node_f$ are `f` and t_0 respectively.
- The relation `param(`$node_f$`,i-1,`$node_{p_i}$`)` denotes that the variable at the node $node_{p_i}$ is a (formal) parameter of the function at the node $node_f$. Before we have already used the relation `param` for the actual parameters of a call expression.

For each function f we generate a node for a pseudo-variable $node_{f\_res}$ which holds a summary of the return values of the function. We call this summary the *result* of the function.[15]

We introduce the following new relations:

- The relation `body(`$node_f$`, `$node_b$`)` in Table 3.6a denotes that the statement corresponding to the node $node_b$ is the body of the function corresponding to the node $node_f$.
- The relation `res_node(`$node_f$`,`$node_{f\_res}$`)` in Table 3.6a denotes that the pseudo variable at the node $node_{f\_res}$ holds the *result* of the function at the node $node_f$.
- The relation `return_stmt(`$node_s$`)` in Table 3.6b denotes that the node $node_s$ corresponds to a `return` statement.

  We use the relation `sub_expr(`$node_s$`,`$node_e$`)` to denote that the expression at the node $node_e$ is the returned expression, i.e. the sub-expression of the `return` statement at the node $node_s$. Further, we use the relation `function(`$node_s$`, `$node_f$`)` to denote that the `return` statement at the node $node_s$ corresponds to the function at the node $node_f$, i.e. is a sub-statement of the body of the function `f`.

3.4.3 Datalog Rules for Inter-Procedural Analysis

Finally, we add the following Datalog rules to extend the analyses for roles to inter-procedural analyses:

a) Data-flow from a returned expression to the result of a function.

The rule in lines 259–261 in the listing below generates for each return statement `RetStmt` in function `Func` the fact `assigned(FuncRes,RetExpr)`, where the node `FuncRes` corresponds to the pseudo-variable for the result of the function `Func`, and the expression `RetExpr` corresponds to the expression returned in the statement `RetStmt`.

[15] We introduce the pseudo-variable node for demonstration purposes. In our implementation, we re-use instead the node corresponding to the function f.

```
259 assigned(FuncRes,RetExpr) :- return_stmt(RetStmt),
260   sub_expr(RetStmt,RetExpr), function(RetStmt,Func),
261   res_node(Func,FuncRes).
```

b) Data-flow from the actual to the formal parameters.

The rule in lines 262–263 generates for each actual parameter `ActParam` in a call expression `CallExpr` the fact `assigned(FormParam,ActParam)`, where `FormParam` is the formal parameter in the function definition of the function `Func` at the same position `I` as the actual parameter `ActParam`.

```
262 assigned(FormParam,ActParam) :- called(Func,CallExpr),
263   param(CallExpr,I,ActParam), param(Func,I,FormParam).
```

c) Data-flow from the result of a function to a variable assigned a call expression.

The rule in line 264 generates for each assignment statement, in which a variable `X` is assigned a call to a function `Func`, the fact `assigned(X,FuncRes)`, where the node `FuncRes` corresponds to the pseudo-variable for the result of the function `Func`.

```
264 assigned(X, FuncRes) :- assigned_call(X,Func), res_node(Func,FuncRes).
```

Example 3.4.1. Consider the example in Fig. 3.30a, which shows the CFG of the statements `return n+1;`, `return ackermann(m-1,1);` and `return ackermann(m-1, ackermann(m,n-1));` of the function `ackermann()` of the code in Fig. 3.5a on page 54, lines 4, 5 and 6 respectively. The corresponding logic program is shown in Fig. 3.30b.

We explain how interprocedural analysis works on example of the role *linear*. We do this in two steps. In Step 1 we show how the rules for the interprocedural analysis in lines 259–263 are applied to the program. In Step 2 we show how the facts inferred in Step 1 are used to infer that the result of the function `ackermann()` is *linear*.

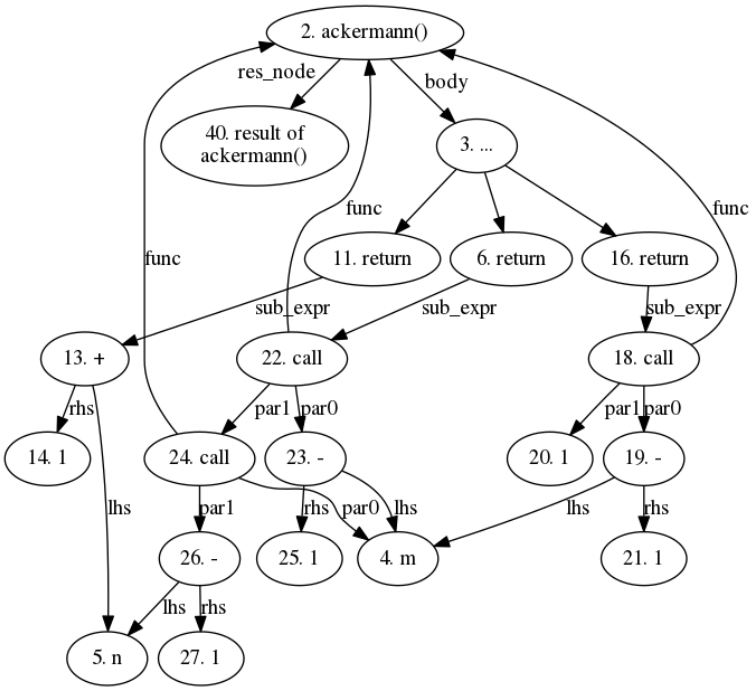
Step 1. Applying the interprocedural rules (lines 259–263).

a) Data-flow from a returned expression to the result of a function.

Note that the node `40` is allocated for a pseudo-variable for the result of the function `ackermann()`. This is encoded with the fact `res_node(2,40)` in line 5 of the Fig. 3.30. We denote the node `40` with $\texttt{node}_{ackermann\_res}$.

The evaluation of the rule in lines 259–261, applied to the statement `return n+1;` (corresponding to the node `11`), infers the fact `assigned(` $\texttt{node}_{ackermann\_res}$`,13)`, where the node `13` corresponds to the expression `n+1`.

In a similar way, for the statements `return ackermann(m-1,ackermann(`



(a) Control flow graph

```
1 type(2,int).
2 name(2,"ackermann").
3 func_decl(2).
4 body(2,3).
5 res_node(2,40).
6 var(40).
7 type(40,int).
8 var(4).
9 type(4,int).
10 name(4,"m").
11 param(2,0,4).
12 var(5).
13 type(5,int).
14 name(5,"n").
15 param(2,1,5).
16 return_stmt(11).
17 function(11,2).
18 sub_expr(11,13).
19 const_literal(14).
20 type(14,int).
21 name(14,"1").
22 bop_expr(13).
23 lhs_expr(13,5).
24 rhs_expr(13,14).
25 opcode(13,"+").
26 return_stmt(16).
27 function(16,2).
28 sub_expr(16,18).
29 call_expr(18).
30 function(18,2).
31 param(18,0,19).
32 const_literal(20).
33 type(20,int).
34 name(20,"1").
35 param(18,1,20).
36 const_literal(21).
37 type(21,int).
38 name(21,"1").
39 bop_expr(19).
40 lhs_expr(19,4).
41 rhs_expr(19,21).
42 opcode(19,"-").
43 return_stmt(6).
44 function(6,2).
45 sub_expr(6,22).
46 call_expr(22).
47 function(22,2).
48 param(22,0,23).
49 param(22,1,24).
50 const_literal(25).
51 type(25,int).
52 name(25,"1").
53 bop_expr(23).
54 lhs_expr(23,4).
55 rhs_expr(23,25).
56 opcode(23,"-").
57 call_expr(24).
58 function(24,2).
59 param(24,0,4).
60 param(24,1,26).
61 const_literal(27).
62 type(27,int).
63 name(27,"1").
64 bop_expr(26).
65 lhs_expr(26,5).
66 rhs_expr(26,27).
67 opcode(26,"-").
```

(b) Datalog program

Figure 3.30 Interprocedural analysis on example of the role *linear*. The statements `return n+1;`, `return ackermann(m-1,1);` and `return ackermann(m-1, ackermann(m, n-1));` of the function `ackermann()` of the code in Fig. 3.5a on page 54, lines 4, 5 and 6 respectively, translated to Datalog.

`m,n-1));` and `return ackermann(m-1,1);` corresponding to the nodes `6` and `16` respectively, the evaluation of the rule in lines 259–261 infers the facts `assigned(node`$_{ackermann\_res}$`,22)` and `assigned(node`$_{ackermann\_res}$`,18)`, where the node `22` corresponds to the expression `ackermann(m-1, ackermann(m,n-1))` and the node `18` – to the expression `ackermann(m-1, 1)`.

b) Data-flow from the actual to the formal parameters.

Note that the variables `m` and `n` are formal parameters of the function `ackermann()`. This is encoded with the facts `param(node`$_{ackermann}$`,0, node`$_m$`)` and `param(node`$_{ackermann}$`,1,node`$_n$`)` in lines 11 and 15 respectively of the Fig. 3.30.

First, for the expression `ackermann(m,n-1)`, corresponding to the node `24`, the evaluation of the rules in lines 262–263 infers the facts `assigned(node`$_m$`, node`$_m$`)` and `assigned(node`$_n$`,26)`, where the node `26` corresponds to the expression `n-1`.

Next, in a similar way, for the expression `ackermann(m-1, ackermann(m, n-1))` the evaluation of the rules in lines 262–263 infers the facts `assigned(node`$_m$`,23)` and `assigned(node`$_n$`,24)`, where the node `23` corresponds to the expression `m-1` and the node `24` to the expression `ackermann(m, n-1))`.

Finally, for the expression `ackermann(m-1,1)`, corresponding to the node `18`, the evaluation of the rules in lines 262–263 infers the facts `assigned(node`$_m$`,19)` and `assigned(node`$_n$`,20)`, where the node `19` corresponds to the expression `m-1` and the node `20` – to the constant literal `1`.

c) Data-flow from the result of a function to a variable assigned a call expression.

Now, recall that the facts `assigned(node`$_{ackermann\_res}$`,18)` and `assigned(node`$_{ackermann\_res}$`,22)`, were inferred in Step 1a), where the node `22` corresponds to the expression `ackermann(m-1,ackermann(m,n-1))` and the node `18` – to the expression `ackermann(m-1,1)`. To these facts now the rule in line 264 is applied. The evaluation infers two identical trivial facts `assigned(node`$_{ackermann\_res}$`, node`$_{ackermann\_res}$`)`.

Step 2. Applying the rules for the role *linear*.

We now use the facts inferred in Step 1 to show that the result of the function `ackermann` is *linear*. Recall that we defined the role *linear* in Section 3.3.1, pages 86–88, lines 136–179 .

First, the evaluation of the rule for the relation `linear_shape` in lines 156–158 on page 87 infers the facts `linear_shape(13)`, `linear_shape(19)`, `linear_shape(23)` and `linear_shape(26)`, where the node `13` corresponds to the expression `n+1`, the nodes `19` and `23` – to the expression `m-1` and the node

`26` – to the expression `n-1`.

Second, the evaluation of the rule for the relation `lin_expr_subvar` in lines 174–175 on page 88 infers the facts `linear_expr_subvar(13,node`$_n$`)`, `linear_expr_subvar(19,node`$_m$`)`, `linear_expr_subvar(23,node`$_m$`)` and `linear_expr_subvar(26,node`$_n$`)`.

Third, the evaluation of the rules for the relation `non_lin_var` in lines 139–147 on pages 87–87 infers that the transitive closure of the relation `non_lin_var` does not include the variables `n`, `m` and `node`$_{ackermann\_res}$.

Finally, the evaluation of the rule for the relation `linear` in line 136 on page 86 infers that the result of the function `ackermann()` is *linear*, encoded with the fact `linear(node`$_{ackermann\_res}$`)`.

3.4.4 Additional Roles for a Porfolio Solver

Now, having defined the framework for inter-procedural analysis, we can introduce the role *recursive function result* for a portfolio solver.

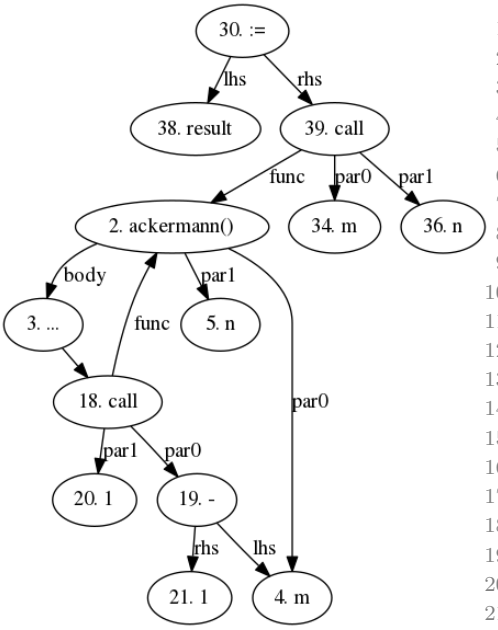
Recursive function result. Variable `X` is *recursive function result* (denoted as `recursive_func_res(X)`, lines 265–266 in the listing below) if `X` is assigned the result of a call to a function `Func`, s.t. `Func` is recursively defined, i.e. the definition `Body` of `Func` (denoted as `body(Func,Body)`) contains a call to the function `Func`:

```
265 recursive_func_res(X) :- var(X), assigned_call(X,Func),
266   body(Func,Body), sub_stmt(Body,CallExpr), called(Func,CallExpr).
267
268 recursive_func_res(X) :- var(X), assigned(X,Y), recursive_func_res(Y).
```

Example 3.4.2. Consider the example in Fig. 3.31a, which shows the CFG of the expression `ackermann(m-1,1)` and statement `result=ackermann(m,n);` in lines 5 and 11 respectively of code in Fig. 3.5a on page 54. The corresponding logic program is shown in Fig. 3.31b.

For this program, the evaluation of the rule defining the relation `assigned_call` in Section. 8.A on page 163, lines 24–25 infers the fact `assigned_call(node`$_{result}$`,` `node`$_{ackermann}$`)`. Then, the evaluation of the rule defining the relation `sub_stmt` (see Section 8.A, lines 59–74) infers the fact `sub_stmt(3,18)`.

Finally, the evaluation of the rule in lines 265–266 infers that `result` is *recursive function result*, encoded with the fact `recursive_func_res(node`$_{result}$`)`. ▲



(a) Control flow graph

```
1 type(2,int).
2 name(2,"ackermann").
3 func_decl(2).
4 body(2,3).
5 var(4).
6 type(4,int).
7 name(4,"m").
8 param(2,0,4).
9 var(5).
10 type(5,int).
11 name(5,"n").
12 param(2,1,5).
13 call_expr(18).
14 function(18,2).
15 param(18,0,19).
16 const_literal(20).
17 type(20,int).
18 name(20,"1").
19 param(18,1,20).
20 const_literal(21).
21 type(21,int).
22 name(21,"1").
23 bop_expr(19).
24 lhs_expr(19,4).
25 rhs_expr(19,21).
26 var(38).
27 type(38,int).
28 name(38,"result").
29 assignment_stmt(30).
30 lhs_expr(30,38).
31 rhs_expr(30,39).
32 var(34).
33 type(34,int).
34 name(34,"m").
35 var(36).
36 type(36,int).
37 name(36,"n").
38 call_expr(39).
39 function(39,2).
40 param(39,0,34).
41 param(39,1,36).
```

(b) Datalog program

Figure 3.31 Example for *recursive function result*. The expression `ackermann(m-1,1)` and statement `result=ackermann(m,n);` in lines 5 and 11 respectively of code in Fig. 3.5a on page 54 translated to Datalog.

3.5 Implementation

We have implemented a tool for the specification of variable roles as and inference of roles for variables in C programs.[16]

In our implementation the roles are inferred for variables and structure fields of scalar data type. In addition, several roles are inferred for pointers, specifically the roles for the portfolio solver, which we described in Section 3.1.2.

As an engine for evaluating Datalog programs, our tool uses an answer set solver Clingo [GKKS14]. We chose the clingo solver, since our initial definitions of the variable roles *linear* and *dynamic enumeration* included non-stratified negation, not supported by standard Datalog (and supported by answer set solvers). Further, we re-formulated the roles so that only stratified negation is used. Though, we did not change to a new solver which not only saved the implementation effort but would also in principle allow to extend our set of roles with new recursively defined roles which use non-stratified negation.

[16] `https://github.com/YuliaDemyanova/variable_roles`. Accessed 23 January 2018.

3.6 Trade-Off for Pointer Analysis

As a trade-off between soundness and performance, our implementation does not include pointer analysis (though we could extend our implementation with e.g. the implementation of pointer analysis in Datalog [BS09]). As a result, the relation `assigned` which we define in Section 8.A, lines 2–3, does not take into account indirect variable accesses. Instead we extend the role *unresolved* to capture indirect accesses to variables.

We now describe how we extend the role *unresolved* to capture indirect variable accesses.

Unresolved. Variable `X` is *unresolved* (denoted as `unresolved(X)`) if either `X` satisfies the definition of *unresolved* in Section 3.3.1, page 80, lines 73–78, or the following condition holds:

- There is an address-of operation (encoded with the code `"ADDR_OF"`), of which `X` is the operand (lines 269– 270).

```
269 unresolved(X) :- var(X), uop_expr(Expr), opcode(Expr,"ADDR_OF"),
270   operand(Expr,X).
```

We also extend the relation `unresolved_expr` as follows. An expression `Expr` is included in the relation `unresolved_expr` if either `Expr` satisfies the conditions for the definition of `unresolved_expr` in Section 3.3.1, page 81, line 79, or `Expr` is one of the following:

- an array subscript expression (line 271),
- a pointer dereference operation, encoded with the operation code `"DEREF"` (line 272 in the listing below).

```
271 unresolved_expr(Expr) :- array_expr(Expr).
272 unresolved_expr(Expr) :- uop_expr(Expr), opcode(Expr,"DEREF").
```

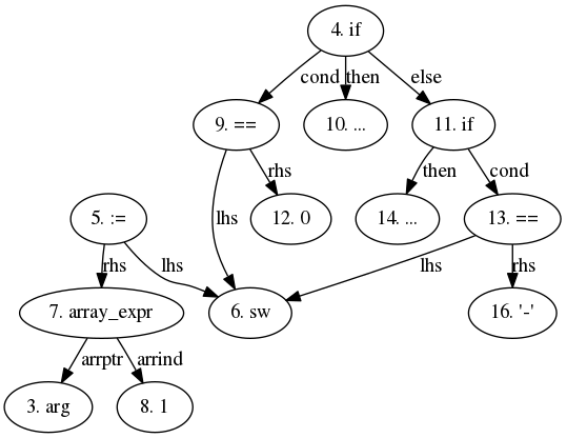
Example 3.6.1. Consider the example in Fig. 3.32a, taken from a cBench benchmark[17] with some modifications. We omit some irrelevant code and denote the omitted code with dots. The function `swproc()` stores the second element of the array `arg` in the variable `sw` and does a case split on this value. In particular, the variable `sw` is compared to the zero symbol (which typically indicates the end of a string), and to several other character constant literals. The CFG of the program and the corresponding logic program are shown in Figs. 3.32b and 3.32c respectively.

The evaluation of the rule in line 271 infers the fact `unresolved_expr(7)`, where the node `7` corresponds to the array expression `arg[1]`. Then, given the fact

[17] `office_ghostscript/src/imainarg.c`.

```
1 int swproc(const char* arg)
2 {
3   char sw = arg[1];
4
5   if (sw==0) {...}
6   else if (sw=='-') {...}
7   else if (sw=='A') {...}
8    ...
9 }
```

(a) Datalog program



(b) Control flow graph

```
1 var(3).
2 type(3,ptr).
3 name(3,"arg").
4 var(6).
5 type(6,int).
6 name(6,"sw").
7 assignment_stmt(5).
8 lhs_expr(5,6).
9 rhs_expr(5,7).
10 const_literal(8).
11 type(8,int).
12 name(8,"1").
13 array_expr(7).
14 arrayptr_expr(7,3).
15 arrayind_expr(7,8).
16 if_stmt(4).
17 condition(4,9).
18 then_stmt(4,10).
19 else_stmt(4,11).
20 const_literal(12).
21 type(12,int).
22 name(12,"0").
23 bop_expr(9).
24 lhs_expr(9,6).
25 rhs_expr(9,12).
26 opcode(9,"==").
27 if_stmt(11).
28 condition(11,13).
29 then_stmt(11,14).
30 const_literal(16).
31 type(16,int).
32 name(16,"'-'").
33 bop_expr(13).
34 lhs_expr(13,6).
35 rhs_expr(13,16).
36 opcode(13,"==").
```

(c) Datalog program

Figure 3.32 Example for (the extended version of) *unresolved*. The statement `sw=arg[1];` and the statements `if (sw==0){...} else if (sw =='-'){...}` of code in Fig. 3.32a, lines 3 and 5–6 respectively, translated to Datalog.

`assigned(node`$_{sw}$`,7)` the evaluation of the rule in line 73 on page 80 infers that the variable `sw` is *unresolved*, encoded as `unresolved(node`$_{sw}$`)`. ▲

CHAPTER 4

Empirical Software Metrics for Benchmarking of Verification Tools

In this chapter, we explore the application of variable roles in software verification. In particular, we devise program metrics based on variable roles and we use these metrics to solve Task 2 formulated in Section 1.5, namely *Portfolio Solver for Software Verification (SV)*. We motivated this task in Section 1.4.1. To make the task practical, we solve it in the context of the Software Verification Competition SV-COMP. We gave a brief introduction in SV-COMP in the Section 1.1.1.

Recall that the benchmarks of the SV-COMP competition are manually partitioned into *categories*, by characteristic features such as usage of bitvectors, concurrent programs, Linux device drivers, etc. As a first step to solve the Task 2 (*Portfolio Solver for SV*), we devised an algorithm which, given an SV-COMP benchmark, predicts its category [DVZ13]. The algorithm is based on a machine learning technique called *support vector machines*, and for making a prediction uses metrics devised from domain-independent variable roles (see Section 3.3.1 for the definition of roles). In our experiments we showed that these metrics are sufficient to classify SV-COMP benchmarks to the competition categories. Note that our choice of the domain-independent variable roles was based on examples from cbench rather than SV-COMP.

Further, we tried to build a portfolio solver for software verification using role-based metrics. We elaborated our algorithm from [DVZ13] so that given a verification task the algorithm predicts a software verification tool which would most efficiently solve the task. In particular, our goal was to build a solver which would beat the participants of the SV-COMP competition with a clear margin. We started with the metrics based on domain-independent roles and our preliminary experiments showed that in a number of

Table 4.1 Sources of complexity for 4 tools participating in SV-COMP'15, marked with + / – / n/a when supported/not supported/no information is available. Extracted from the competition report [Bey15] and tool papers [CKL04, DPV13].

Source of complexity	CBMC	Predator	CPAchecker	SMACK	**Corresponding feature**	
					Name	**Ref. to definition**
pointers	+	+	+	+	*pointer*	Sec. 3.3.1
arrays	+	–	n/a	+	*array index*	
non-static pointer offsets	–	+	n/a	n/a	*pointer offset*	
non-static size of heap-allocated memory	+	+	n/a	n/a	*allocation size*	
bit operations	+	–	+	–	*bitvector*	
external functions	+	–	n/a	n/a	*input*	
dynamic data structures	n/a	+	n/a	+	*linked list*	Sec. 3.3.2
integer variables	+	+	+	+	*integral*	
multi-threading	+	–	–	–	*thread descriptor*	
unbounded loops	–	n/a	n/a	–	$\mathcal{L}^{\text{SB}}, \mathcal{L}^{\text{ST}}, \mathcal{L}^{\text{simple}}, \mathcal{L}^{\text{hard}}$	Sec. 4.1.2
pointers to functions	+	n/a	n/a	n/a	$m_{\text{fpcalls}}, m_{\text{fpargs}}$	Sec. 4.1.3
recursion	–	–	–	+	m_{reccalls}	
big CFG ($\geq$ 100 KLOC)	+	n/a	n/a	+	$m_{\text{cfgblocks}}$, m_{maxindeg}	

cases the algorithm makes non-optimal predictions. Therefore, we iteratively improved the solver by adding new variable roles which captured the incorrectly predicted cases. In this way, we devised an extended set of roles for portfolio solver, which we defined in Section 3.3.2.

We describe the metrics for portfolio solver algorithm in Section 4.1. In addition to the role-based metrics, our algorithm uses metrics based on patterns of loops in C programs [PVZ15] and indicators of control-flow complexity. In Section 4.2 we give a description of our portfolio solver algorithm and precede it with an introduction to the machine learning techniques used in the portfolio solver. We describe the evaluation of our portfolio construction on the data from SV-COMP in Section 4.3.

4.1 Source Code Metrics for Software Verification

To choose the software metrics describing SV-COMP benchmarks, we consider the various techniques used in software verification along with their target domains, our intuition as programmers, as well as the tool developer reports in their competition contributions. Table 4.1 exemplarily summarizes these reports for tools CBMC, Predator, CPAchecker and SMACK: The first column gives obstacles the tools' authors identified,

the following columns show whether the feature is supported by respective tool, and the last two columns reference the corresponding metrics, which we introduce below in this section. The obtained metrics are naturally understood in three dimensions that we motivate informally first:

1. *Program Variables.* Does the program deal with machine or unbounded integers? Are the ints used as indices, bit-masks or in arithmetic? Dynamic data structures? Arrays? Interval analysis or predicate abstraction?
2. *Program Loops.* Reducible loops or goto programs? FOR-loops or ranking functions? Widening, loop acceleration, termination analysis, or loop unrolling?
3. *Control Flow.* Recursion? Function pointers? Multithreading? Simulink-style code or complex branching?

Our hypothesis is that precise metrics along these dimensions allow us to predict tool performance. The challenge lies in identifying metrics which are predictive enough to understand the relationship between tools and benchmarks, but also simple enough to be used in a preprocessing and classification step. In Sections 4.1.1, 4.1.2 and 4.1.3 we introduce program features along the three dimensions – *program variables*, *program loops* and *control flow* – and describe how to derive corresponding metrics using simple data-flow analyses.[1]

4.1.1 Variable Role Based Metrics

The first set of features we consider are variable roles. We introduce the variable roles for portfolio solver and describe our criteria for choosing the roles in Section 3.1.2. We give the definitions of the roles in Section 3.3.2. Note that the set of variable roles for portfolio solver extend the set of domain-independent variable roles which we describe in Sections 3.1.1 and 3.3.1. We now define the metrics based on the variable roles.

Definition 1 (Variable role based metrics). For a given benchmark file f, we compute the mapping $Res^R : Roles \to 2^{Vars}$ from variable roles to sets of program variables of f. We derive role metrics m_R that represent the relative occurrence of each variable role $R \in Roles$:

$$m_R = \frac{|Res^R|}{|Vars|} \qquad R \in Roles \tag{4.1}$$

4.1.2 Loop Pattern Based Metrics

The second set of program features we consider is a *classification of loops* in the program under verification, as introduced in [PVZ15]. Although undecidable in general, the ability to reason about bounds or termination of loops is highly useful for software verification:

[1] We stress that the classification of the loops and the identification of control flow-based features as well as the implementation of respective algorithms are not the contributions of this dissertation.

Table 4.2 List of loop patterns with informal descriptions.

Loop pattern	**Empirical hardness**	**Informal definition**
Syntactically bounded loops $\mathcal{L}^{\text{bounded}}$	easy	The number of executions of the loop body is bounded (considers outer control flow).
FOR loops $\mathcal{L}^{\text{FOR}}$	intermediate	The loop terminates whenever control flow enters it (disregards outer control flow).
Generalized FOR loops $\mathcal{L}^{\text{FOR(*)}}$	advanced	A heuristic derived from FOR loops by weakening the termination criteria. A good heuristic for termination.
Hard loops $\mathcal{L}^{\text{hard}}$	hard	Any loop that is not classified as generalized FOR loop.

For example, it allows a tool to assert the (un)reachability of program locations after the loop, and to compute unrolling factors and soundness limits in the case of bounded model checking.

Criteria for choosing loop patterns. We consider 4 heuristics for loop termination defined in [PVZ15]. In Table 4.2 we list the heuristics along with their informal definitions. In particular, we list the heuristics in the order of weakening constraints, i.e. $\mathcal{L}^{\text{bounded}} \subseteq \mathcal{L}^{\text{FOR}} \subseteq \mathcal{L}^{\text{FOR(*)}} \subseteq \mathcal{L}^{\text{hard}}$, where $\mathcal{L}^{H}$ denotes the set of loops matching the heuristic H:

First, the restricted set of the *FOR* loops corresponds to a frequently used programming pattern. Then, the *syntactically bounded* and *generalized FOR* loops are obtained by strengthening and weakening respectively the constrains of the *FOR* loop. Finally, the *hard* loops are defined so that the classification of the loops is complete.

We give a more detailed definition of the heuristics in Appendix, Section 8.C. Below we define the metrics based on loop patterns.

Definition 2 (Loop pattern based metrics). For a given benchmark file f, we compute $\mathcal{L}^{\text{bounded}}$, $\mathcal{L}^{\text{FOR}}$, $\mathcal{L}^{\text{FOR}(*)}$, $\mathcal{L}^{\text{hard}}$, and the set of all loops *Loops*. We derive loop metrics m_H that represent the relative occurrence of each loop heuristic H:

$$m_H = \frac{|\mathcal{L}^H|}{|Loops|} \qquad H \in \{\text{bounded}, \text{FOR}, \text{FOR}(*), \text{hard}\} \tag{4.2}$$

4.1.3 Control Flow Based Metrics

Complex control flow poses another challenge for program analysis. To measure the complexity of control flow, we introduce four additional metrics:

- For *intraprocedural control flow*, we count the following numbers:

  - The number of *basic blocks* in the control flow graph (CFG) $m_{\text{cfgblocks}}$, where a basic block is the longest possible sequence of code without branches;
  - The maximum number of edges entering any basic block in the CFG m_{maxindeg}.

- To represent *indirect function calls*, we measure the following numbers:
  - The ratio m_{fpcalls} of call expressions taking a function pointer as argument (over all call expressions);
  - The ratio m_{fpargs} of parameters to call expressions that have a function pointer type (over all parameters to call expressions).

Criteria for choosing control flow features. We identify the features $m_{\text{cfgblocks}}$ and m_{maxindeg} to capture the peculiarities of the category ECA. The term ECA stands for event-condition action and the benchmarks of the category ECA represent automatically generated code with lots of simple `if-then-else` branches. We have already justified the choice of the metrics capturing indirect function calls in Section 4.1.1.

4.1.4 Usefulness of Our Features for Selecting a Verification Tool

In our experiments (see Section 4.3), we will demonstrate that a portfolio built on top of these metrics performs well as a tool selector. In this section, we already give two reasons why we believe these metrics have predictive power in the software verification domain in the first place.

Tool developer reports. The developer reports in the competition report for SV-COMP'15 [Bey15], as well as tool papers (e.g. [CKL04, DPV13], for a full list of tool papers see the competition report), give evidence for the relevance of our features for selecting verification tools: They mention language constructs, which – depending on whether they are fully, partially, or not modeled by a tool – constitute its strengths or weaknesses. We give a short survey of such language constructs in Table 4.1 and relate them to our features. For example, Predator is specifically built to deal with dynamic data structures (variable role *linked list*) and pointer offsets (*offset*), and CPAchecker does not model multi-threading (*thread descriptor*) or support recursion (*recursive function result*). For CBMC, unbounded loops (various loop patterns $\mathcal{L}^{\text{H}}$) are an obstacle.

Preliminary experiments. In addition, in previous work we have successfully used variable roles and loop patterns to deduce properties of verification tasks:

- In [DVZ13], we use *variable roles* to predict – for a given verification task – its category in SV-COMP'13.
- In [PVZ15], *loop patterns* are shown to be good heuristics for identifying bounded loops.

These give further evidence for our claim that the features described above are useful in predicting properties of verification tasks.

4.2 A Portfolio Solver for Software Verification

4.2.1 Preliminaries on Machine Learning

In this section we introduce standard terminology from the machine learning community (see for example [Bis06]).

Supervised Machine Learning

In supervised machine learning problems, we learn a *model* $M : \mathbb{R}^n \to \mathbb{R}$. The $\mathbf{x}_i \in \mathbb{R}^n$ are called *feature vectors*, measuring some property of the object they describe. The $y_i \in \mathbb{R}$ are called *labels*.

We learn model M by considering a set of labeled examples $X||\mathbf{y} = \{(\mathbf{x}_i, y_i)\}_{i=1}^N$. M is then used to predict the label of previously unseen inputs $\mathbf{x}' \notin X$.

We distinguish two kinds of supervised machine learning problems:

- *Classification* considers labels from a finite set $y \in \{1, \ldots, C\}$. For $C = 2$, we call the problem *binary classification*, for $C > 2$ we speak of *multi-class classification*.
- *Regression* considers labels from the real numbers $y \in \mathbb{R}$.

Support Vector Machines

A *support vector machine* (SVM) [BGV92, CV95] is a binary classification algorithm that finds a hyperplane $\mathbf{w} \cdot \mathbf{x} + b = 0$ separating data points with different labels. We first assume that such a hyperplane exists, i.e. that the data is *linearly separable*:

Also called a *maximum margin classifier*, SVM learns a hyperplane that maximizes the gap $||\mathbf{w}||^{-1}$ (*margin*) between the hyperplane and the nearest data points with different labels. Maximizing the margin is formulated as

$$\text{minimize } ||\mathbf{w}|| \text{ subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 \text{ for } i = 1, \ldots, N \tag{4.3}$$

which is usually encoded as the following quadratic programming problem:

$$\text{maximize } \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i,j=1}^{N} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j \text{ subject to } \alpha_i \geq 0 \text{ and } \sum_{i=1}^{N} \alpha_i y_i = 0. \tag{4.4}$$

After computing the separating hyperplane on a set of labeled examples, a previously unseen feature vector $\mathbf{x}'$ is classified using function

$$M(\mathbf{x}') = \operatorname{sgn}\left(\mathbf{w} \cdot \mathbf{x}' + b\right). \tag{4.5}$$

Thus M predicts the class of $\mathbf{x}'$ by computing on which side of the hyperplane it falls.

If the data is not linearly separable, e.g. due to outliers or noisy measurements, there are two orthogonal approaches that we both make use of in our portfolio solver:

Soft-margin SVM. Soft-margin SVM allows some data points to be misclassified while learning the hyperplane. For this, we associate a slack variable $\xi_i \geq 0$ with each data point $\mathbf{x}_i$, where

$$\xi_i = \begin{cases} \text{the distance from the hyperplane} & \text{if } \mathbf{x}_i \text{ is misclassified} \\ 0 & \text{otherwise} \end{cases}.$$

We thus replace Equation 4.3 with the following equation:

$$\text{minimize } ||\mathbf{w}|| + C\sum_{i=1}^{N} \xi_i \text{ subject to } y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i \text{ for } i = 1, \ldots, N \tag{4.6}$$

and substitute $0 \leq \alpha_i \leq C$ for the constraint $\alpha_i \geq 0$ in Equation 4.4. Parameter $C > 0$ controls the trade-off between allowing misclassification and maximizing the margin.

Kernel transformations. Another, orthogonal approach to data that is not linearly separable *in the input space*, is to transform it to a higher-dimensional *feature space* $\mathbb{H}$ obtained by a transformation $\phi : \mathbb{R}^n \to \mathbb{H}$. For example, 2-class data not linearly separable by $\mathbb{R}^2$ can be linearly separated in $\mathbb{R}^3$ if ϕ pushes points of class 1 above, and points of class 2 below some plane.

The quadratic programming formulation of SVM allows for an efficient implementation of this transformation: We define a *kernel function* $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$ instead of explicitly giving ϕ, and replace the dot product in Equation 4.4 with $K(\mathbf{x}_i, \mathbf{x}_j)$. An example of a non-linear kernel function is the *radial basis function* (RBF): $K(\mathbf{x}_i, \mathbf{x}_j) = \exp(-\gamma||\mathbf{x}_i - \mathbf{x}_j||^2),\ \gamma > 0$.

For classifying unseen feature vectors $\mathbf{x}'$, we replace Equation 4.5 with

$$M(\mathbf{x}') = \text{sgn}\left(\mathbf{w} \cdot \phi(\mathbf{x}') + b\right) \text{ where } \mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \phi(\mathbf{x}_i). \tag{4.7}$$

Probabilistic Classification

Probabilistic classification is a generalization of the classification algorithm, which searches for a function $M : \mathbb{R}^n \to \Pr(\mathbf{y})$, where $\Pr(\mathbf{y})$ is the set of all probability distributions over $\mathbf{y}$. $M(\mathbf{x}')$ then gives the probability $\mathrm{p}(y_i \mid \mathbf{x}', X||\mathbf{y})$, i.e. the probability that $\mathbf{x}'$ actually has label y_i given the model trained on $X||\mathbf{y}$. There is a standard algorithm for estimating class probabilities for SVM [WLW04].

Creating and Evaluating a Model

The labeled set $X||\mathbf{y}$ used for creating (training) model M is called *training set*, and the set X' used for evaluating the model is called *test set*. To avoid overly optimistic evaluation of the model, it is common to require that the training and test sets are disjoint: $X \cap X' = \emptyset$. A model which produces accurate results with respect to $||\mathbf{w}||$ for the training set, but results in a high error for previously unseen feature vectors $\mathbf{x}' \notin X$, is said to *overfit*.

Data Imbalances

The training set $X||\mathbf{y}$ is said to be *imbalanced* when it exhibits an unequal distribution between its classes: $\exists y_i, y_j \in \mathbf{y}\ .\ \text{num}(y_i)/\text{num}(y_j) \sim 100$, where $\text{num}(y) = |\{\mathbf{x}_i \in X \mid y_i = y\}|$, i.e. imbalances of the order 100:1 and higher. Data imbalances significantly compromise the performance of most standard learning algorithms [HG09].

A common solution for the imbalanced data problem is to use a *weighting function* $\text{Weight} : X \to \mathbb{R}$ [HD05]. *SVM with weights* is a generalization of SVM, where we

$$\text{minimize } ||\mathbf{w}|| + C \sum_{i=1}^{N} \text{Weight}(\mathbf{x}_i)\xi_i. \tag{4.8}$$

Weight is usually chosen empirically.

Multi-Class Classification

SVM is by nature a binary classification algorithm. To tackle multi-class problems, we reduce an n-class classification problem to n binary classification problems: *One-vs.-rest* classification creates one model M_i per class i, with the labeling function

$$M_i(\mathbf{x}) = \begin{cases} 1 & \text{if } M(\mathbf{x}) = i \\ -1 & \text{otherwise} \end{cases}$$

and the predicted value is calculated as $M(\mathbf{x}) = \mathit{choose}\,\{i \mid M_i(\mathbf{x}) = 1\}$, where a suitable operator *choose* is used to select a single class from multiple predicted classes.

4.2.2 The Competition on Software Verification SV-COMP

In this section we give the definitions which formalise the competition's setup and which we later use in the description of our portfolio solver (Section 4.2.3). Detailed information about the competition is available on its website.[2]

SV-COMP maintains a repository of verification tasks, on which the competition's participants are tested:

[2] `https://sv-comp.sosy-lab.org`. Accessed 23 January 2018.

Definition 3 (Verification task.). We denote the set of all considered verification tasks as *Tasks*. A verification task $v \in \mathit{Tasks}$ is described by a triple $v = (f, p, \mathit{type})$ of a C source file f, verification property p and property type *type*. For SV-COMP'14 and '15, *type* is either a label reachability check or a memory safety check (comprising checks for freedom of unsafe deallocations, unsafe pointer dereferences, and memory leaks). SV-COMP'16 adds the property types overflow and termination.

For each verification task, its designers define the expected answer, i.e. if property p holds on f:

Definition 4 (Expected answer.). Function ExpAns : $\mathit{Tasks} \to \{\text{true}, \text{false}\}$ gives the *expected answer* for task v, i.e. $\text{ExpAns}(v) = \text{true}$ if and only if property p holds on f.

Furthermore, SV-COMP partitions the verification tasks *Tasks* into *categories*, a manual grouping by characteristic features such as usage of bitvectors, concurrent programs, linux device drivers, etc.

Definition 5 (Competition category.). Let *Categories* be the set of competition categories. Let Cat : $\mathit{Tasks} \to \mathit{Categories}$ define a partitioning of *Tasks*, i.e. $\text{Cat}(v)$ denotes the category of verification task v.

Finally, SV-COMP assigns a *score* to each tool's result and computes weighted *category scores*. For example, the *Overall* SV-COMP score considers a meta category of all verification tasks, with each constituent category score normalized by the number of tasks in it. We describe and compare the scoring policies of recent competitions in Section 4.3.1. In addition, medals are awarded to the three best tools in each category. In case multiple tools have equal scores, they are ranked by runtime for awarding medals.

Definition 6 (Score, category score, *Overall* score.). Let $\mathit{score}_{t,v}$ denote the score of tool $t \in \mathit{Tools}$ on verification task $v \in \mathit{Tasks}$ calculated according to the rules of the respective edition of SV-COMP. Let $\text{cat\_score}(t, c)$ denote the score of tool t on the tasks in category $c \in \mathit{Categories}$ calculated according to the rules of the respective edition of SV-COMP.

4.2.3 Tool Selection as a Machine Learning Problem

In this section, we describe the setup of our portfolio solver $\mathcal{TP}$. We give formal definitions for modeling SV-COMP, describe the learning task as multi-class classification problem, discuss options for breaking ties between multiple tools predicted correct, present our weighting function to deal with data imbalances, and finally discuss implementation specifics.

Definitions

Definition 7 (Verification tool.). We model the constituent verification tools as set $\mathit{Tools} = \{1, 2, \ldots, |\mathit{Tools}|\}$ and identify each verification tool by a unique natural number $t \in \mathit{Tools}$.

Definition 8 (Tool run.). The result of a run of tool t on verification task $v = (f, p, \mathit{type})$ is a triple

$$\langle \mathit{ans}_{t,v}, \mathit{runtime}_{t,v}, \mathit{memory}_{t,v} \rangle$$

where $\mathit{ans}_{t,v} \in \{\text{true}, \text{false}, \text{unknown}\}$ is the tool's answer whether property p holds on file f, i.e.

$$\mathit{ans}_{t,v} = \begin{cases} \text{true} & \text{if } t \text{ claims } f \text{ satisfies } p, \\ \text{false} & \text{if } t \text{ claims } f \text{ does not satisfy } p \\ \text{unknown} & \text{if } t \text{ claims it cannot decide } p \text{ on } f, \\ & \text{or } t \text{ fails to decide } p \text{ on } f \text{ (e.g. tool crash, time-/mem-out)} \end{cases}$$

and $\mathit{runtime}_{t,v} \in \mathbb{R}$ (respectively $\mathit{memory}_{t,v} \in \mathbb{R}$) is the runtime (respectively memory usage) of tool t on task v in seconds (respectively megabytes).

Definition 9 (Virtual best solver.). The *virtual best solver* (VBS) is an oracle that selects for each verification task the tool which gives the correct answer in minimal time.

Machine Learning Data

We compute feature vectors from the metrics introduced in Section 4.1 and the results of SV-COMP as follows:

For verification task $v = (f, p, \mathit{type})$ we define feature vector

$$\begin{aligned} \mathbf{x}(v) = (& m_{\text{array index}}(v), \ldots, m_{\text{recusive function result}}(v), \\ & m_{\text{bounded}}(v), \ldots, m_{\text{hard}}(v), \\ & m_{\text{cfgblocks}}(v), \ldots, m_{\text{fpargs}}(v), \\ & \mathit{type}) \end{aligned}$$

where the $m_i(v)$ are our metrics from Section 4.1 computed on f and $\mathit{type} \in \{0, 1, 2, 3\}$ encodes if the property is reachability, memory safety, overflow, or termination.

We associate each feature vector $\mathbf{x}(v)$, with a label $t \in \mathit{Tools}$, such that t is the tool chosen by the virtual best solver for task v. In the following, we reduce the corresponding classification problem to $|\mathit{Tools}|$ independent classification problems.

Formulation of the Machine Learning Problem

For each tool $t \in \mathit{Tools}$, $\mathcal{TP}$ learns a model to predict whether tool t gives gives a correct or incorrect answer, or responds with „unknown“. Since the answer of a tool does not depend on the answers of other tools, $|\mathit{Tools}|$ independent models (i.e., one per tool) give more accurate results and prevent overfitting.

We define labeling function $L_t(v)$ for tool t and task v as follows:

$$L_t(v) = \begin{cases} 1 & \text{if } \mathit{ans}_{t,v} = \text{ExpAns}(v), \\ 2 & \text{if } \mathit{ans}_{t,v} = \text{unknown}, \\ 3 & \text{otherwise.} \end{cases}$$

I.e., $L_t(v) = 1$ if tool t gives the correct answer on v, $L_t(v) = 2$ if t answers unknown, and $L_t(v) = 3$ if t gives an incorrect answer. A tool can opt-out from a category, which we treat as if the tool had answered unknown for all of the category's verification tasks. Thus, for each tool t, we obtain training data $\{(\mathbf{x}(v), L_t(v))\}_{v \in \mathit{Tasks}}$ from which we construct model M_t.

Tool selection based on predicted answer correctness. Let operator *choose* : $(2^{\mathit{Tools}} \times \mathit{Tasks}) \to \mathit{Tools}$ select for a given task one tool from a set of tools $\mathit{TPredicted} \subseteq \mathit{Tools}$ (we give concrete definitions of *choose* below). Given $|\mathit{Tools}|$ predictions of the models $M_t, t \in \mathit{Tools}$ for a task v, the portfolio algorithm selects a single tool t^{best} as follows:

$$t^{\text{best}} = \begin{cases} \mathit{choose}(\text{TCorr}(v), v) & \text{if } \text{TCorr}(v) \neq \emptyset, \\ \mathit{choose}(\text{TUnk}(v), v) & \text{if } \text{TCorr}(v) = \emptyset \wedge \text{TUnk}(v) \neq \emptyset, \\ t^{\text{winner}} & \text{if } \text{TCorr}(v) = \emptyset \wedge \text{TUnk}(v) = \emptyset. \end{cases}$$

where $\text{TCorr}(v)$ and $\text{TUnk}(v)$ are the sets of tools predicted to give the correct answer and respond with „unknown“ on v, respectively:

$$\text{TCorr}(v) = \{t \in \mathit{Tools} \mid M_t(v) = 1\}$$
$$\text{TUnk}(v) = \{t \in \mathit{Tools} \mid M_t(v) = 2\}$$

and t^{winner} is the *Overall* winner of the competition, e.g. UltimateAutomizer in SV-COMP'16.

Choosing among Tools Predicted Correct

We now describe three alternative ways of implementing the operator *choose*:

1. **Time: $\mathcal{TP}^{\mathbf{time}}$.** We formulate $|\mathit{Tools}|$ additional regression problems: For each tool t, we use training data $\{(\mathbf{x}(v), \mathit{runtime}^{\text{norm}}_{t,v})\}_{v \in \mathit{Tasks}}$ to obtain a model $M^{\text{time}}_t(v)$ predicting runtime, where

$$\mathit{runtime}^{\text{norm}}_{t,v} = \text{norm}(\mathit{runtime}_{t,v}, \{\mathit{runtime}_{t',v'}\}_{t' \in \mathit{Tools}, v' \in \mathit{Tasks}})$$

and norm normalizes to the unit interval:

$$\text{norm}(x, X) = \frac{x - \min(X)}{\max(X) - \min(X)}.$$

The predicted value $M_t^{\text{time}}(v)$ is the predicted runtime of tool t on task v. We define

$$\mathit{choose}(\mathit{TPredicted}, v) = \underset{t \in \mathit{TPredicted}}{\arg\min} \; M_t^{\text{time}}(v).$$

2. **Memory: $\mathcal{TP}^{\mathbf{mem}}$.** Similar to $\mathcal{TP}^{\text{time}}$, we formulate $|\mathit{Tools}|$ additional regression problems: For each tool t, we use training data $\{(\mathbf{x}(v), \mathit{memory}_{t,v}^{\text{norm}})\}_{v \in \mathit{Tasks}}$ to obtain a model $M_t^{\text{mem}}(v)$ predicting runtime, where

$$\mathit{memory}_{t,v}^{\text{norm}} = \text{norm}(\mathit{memory}_{t,v}, \{\mathit{memory}_{t',v'}\}_{t' \in \mathit{Tools}, v' \in \mathit{Tasks}}).$$

We define

$$\mathit{choose}(\mathit{TPredicted}, v) = \underset{t \in \mathit{TPredicted}}{\arg\min} \; M_t^{\text{mem}}(v).$$

3. **Class probabilities: $\mathcal{TP}^{\mathbf{prob}}$.** We define the operator

$$\mathit{choose}(\mathit{TPredicted}, v) = \underset{t \in \mathit{TPredicted}}{\arg\max} \; P_{t,v}$$

where $P_{t,v}$ is the class probability estimate for $M_t(v) = 1$, i.e. the probability that tool t gives the expected answer on v.

In Table 4.3 we present preliminary experiments comparing the *choose* operators for category *Overall* in the setup of SV-COMP'14. We consider the following criteria: the percentage of correctly and incorrectly answered tasks, SV-COMP score, runtime, and the place in the competition.

Discussion. $\mathcal{TP}^{\text{mem}}$ and $\mathcal{TP}^{\text{time}}$ clearly optimize the overall memory usage and runtime, respectively. At the same time, they fall behind $\mathcal{TP}^{\text{prob}}$ with respect to the ratio of correct answers and SV-COMP score. Our focus here is on building a portfolio for SV-COMP, where tools are ranked by score. In the following we thus focus on the implementation of *choose* from $\mathcal{TP}^{\text{prob}}$ and refer to it as $\mathcal{TP}$.

Dealing with Data Imbalances

An analysis of the SV-COMP data shows that the labels $L_t(v)$ are highly imbalanced: For example, in SV-COMP'14 the label which corresponds to incorrect answers, $L_t(v) = 3$, occurs in less than 4% for every tool. The situation is similar for SV-COMP'15 and '16. We therefore use SVM with weights, in accordance with standard practice in machine learning.

Table 4.3 Comparison of formulations of $\mathcal{TP}$, using different implementations of operator *choose*. Runtime shown here is de-normalized from the predicted (normalized) value defined above.

Setting	Correct / Incorrect / Unknown answers, %	Score	Runtime (min)	Memory (GiB)	Place
$\mathcal{TP}^{\text{mem}}$	88/2/10	1047	2819	390.2	3
$\mathcal{TP}^{\text{time}}$	92/2/6	1244	920	508.4	1
$\mathcal{TP}^{\text{prob}}$	94/1/5	1443	2866	618.1	1

Given a task v and tool t, we calculate the weighting function Weight as follows:

$$\begin{aligned}\text{Weight}(v,t) = &\,\text{Potential}(v) \times \text{Criticality}(v) \times \\ &\text{Performance}(t, \text{Cat}(v)) \times \text{Speed}(t, \text{Cat}(v)).\end{aligned}$$

We briefly give informal descriptions of functions Potential, Criticality, Performance, Speed before defining them formally:

- **Potential**(v) describes how important predicting a correct tool for task v is, based on its score potential. E.g., safe tasks (ExpAns = true) have more points deducted for incorrect answers than unsafe (ExpAns = false) tasks, thus their score potential is higher.
- **Criticality**(v) captures how important predicting a correct tool is, based on how many tools give a correct answer. Intuitively, this captures how important an informed decision about task v, as opposed to a purely random guess, is.
- **Performance**(t, c) describes how well tool t does on category c compared to the category winner.
- **Speed**(t, c) describes how fast tool t solves tasks in category c compared to the fastest tool in the category.

Note that the weighting function uses the information about the correct answer and the category of a task, which during the competition is withheld from the competition. This is possible because we use the weighting function only for training our model, and we do not use this information for testing the model.

We now formally define the four functions:

$$\text{Potential}(v) = \text{score}_{\max}(v) - \text{score}_{\min}(v)$$

where $\text{score}_{\max}(v)$ and $\text{score}_{\min}(v)$ are the maximal and minimal possible scores for task v, respectively. For example, in the setup of SV-COMP'14, if v is safe, then $\text{score}_{\max}(v) = 2$ and $\text{score}_{\min}(v) = -8$.

$$\text{Criticality}(v) = |\{t \in \mathit{Tools} \mid \mathit{ans}_{t,v} = \text{ExpAns}(v)\}|^{-1}$$

is inversely proportional (subject to a constant factor) to the probability of randomly choosing a tool which gives the correct answer.

$$\text{Performance}(t, c) = \frac{\text{cat\_score}(t, c) - \text{cat\_score}_{\min}(c)}{\text{cat\_score}(t^{\mathit{cbest}}, c) - \text{cat\_score}_{\min}(c)}$$

is the ratio of SV-COMP scores of tool t and the category winner t^{cbest} on tasks from category c, where

$$\begin{aligned} t^{\mathit{cbest}} &= \operatorname*{arg\,max}_{t_i \in \mathit{Tools}} \text{cat\_score}(t_i, c) \\ \text{cat\_score}(t, c) &= \sum_{\{v \in \mathit{Tasks} \mid \text{Cat}(v) = c\}} \mathit{score}_{t,v} \\ \text{cat\_score}_{\min}(c) &= \sum_{\{v \in \mathit{Tasks} \mid \text{Cat}(v) = c\}} \text{score}_{\min}(v) \end{aligned}$$

and $\mathit{score}_{t,v}$ is the SV-COMP score of tool t on task v.

$$\text{Speed}(t, c) = \frac{\ln \text{rel\_time}(t, c)}{\ln \text{rel\_time}(t^{\mathit{cfst}}, c)}$$

is the ratio of orders of magnitude of normalized total runtime of tool t and of the fastest tool t^{cfst} in category c, where

$$\begin{aligned} \text{rel\_time}(t, c) &= \frac{\text{cat\_time}(t, c)}{\sum_{t_i \in \mathit{Tools}} \text{cat\_time}(t_i, c)} \\ t^{\mathit{cfst}} &= \operatorname*{arg\,min}_{t_i \in \mathit{Tools}} \text{cat\_time}(t_i, c) \\ \text{cat\_time}(t, c) &= \sum_{\{v \in \mathit{Tasks} \mid \text{Cat}(v) = c\}} \mathit{runtime}_{t,v}. \end{aligned}$$

Implementation of $\mathcal{TP}$

Finally, we discuss details of the implementation of $\mathcal{TP}$. We use the SVM machine learning algorithm with the RBF kernel and weights as implemented in the LIBSVM library [CL11]. To find optimal parameters C for soft-margin SVM and γ for the RBF kernel, we do exhaustive search on the grid, as described in [HCL+03].

4.2.4 Virtual Strategies

To estimate how optimally our portfolio solver chooses a tool, we also define two theoretic strategies T_{cat} and T_{vbs}, to which we will compare our portfolio solver in Section 4.3:

- Given a verification task v, T_{cat} selects the tool winning the corresponding competition category $\text{Cat}(v)$.

- T_{vbs} is the *virtual best solver* (VBS): the strategy selects for each verification task the tool which gives the correct answer in minimal time.

Neither T_{cat} nor T_{vbs} can be built in practice: For T_{cat}, we would need to know competition category $\text{Cat}(v)$ of verification task v, which is withheld from the competition participants (e.g., the category of a verification task could be predicted using our machine-learning algorithm for benchmark classification [DVZ13]). For T_{vbs}, we would need an oracle telling us the tool giving the correct answer in minimal time. Thus any practical approach must be a heuristic such as the portfolio described in this work.

4.3 Experimental Results

4.3.1 SV-COMP 2014 vs. 2015 vs. 2016

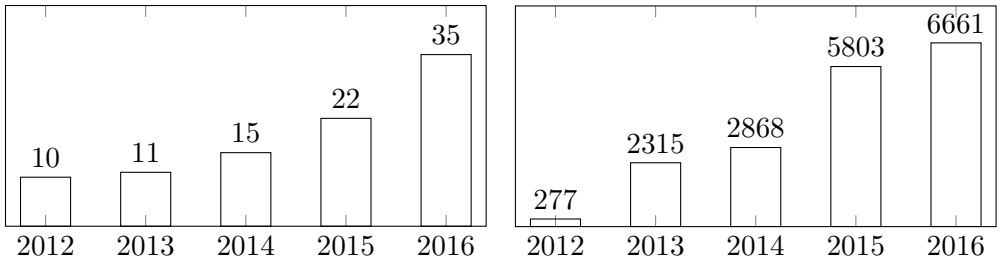
Candidate tools and verification tasks. Considering the number of participating tools, SV-COMP is a success story: Figure 4.1a shows the increase of participants over the years. Especially the steady increase in the last two years is a challenge for our portfolio, as the number of machine learning problems (cf. Section 4.2.3) increases. As Figure 4.1b shows, also the number of verification tasks used in the competition has increased steadily.

Scoring. As described in Section 4.2.2, SV-COMP provides two metrics for comparing tools: score and medal counts. As Table 4.1c shows, the scoring policy has constantly changed (the penalties for incorrect answers were increased). At least for 2015, this was decided by a close jury vote.[3] We are interested how stable the competition ranks are under different scoring policies. Table 4.4 gives the three top-scoring tools in *Overall* and their scores in SV-COMP, as well as the top-scorers of each year if the scoring policy of other years had been applied:

Clearly, the scoring policy has a major impact on the competition results: In the latest example of SV-COMP'16, UltimateAutomizer wins SV-COMP'16 with the original scoring policy applied, but is not even among the three top-scorers if the policies of 2015 or 2014 are applied.

Given that SV-COMP score and thus also medal counts are rather volatile, we introduce *decisiveness-reliability plots* (DR-plots) in the next section to complement our interpretation of the competition results.

[3]The transcript of the meeting was originally published at `http://sv-comp.sosy-lab.org/2015/Minutes-2014.txt` (accessed 6 February, 2015) and is no longer available. The archived version is available at `https://web.archive.org/web/20150413080431/http://sv-comp.sosy-lab.org/2015/Minutes-2014.txt` (accessed 23 January 2018).



(a) Number of participants in SV-COMP over the years.

(b) Number of verification tasks in SV-COMP over the years.

Tool reports	**Tool's answer**	**SV-COMP score**				
		2012	**2013**	**2014**	**2015**	**2016**
Unknown	n/a	0	0	0	0	0
Property does not hold	correct	+1	+1	+1	+1	+1
	incorrect	**-2**	**-4**	-4	**-6**	**-16**
Property holds	correct	+2	+2	+2	+2	+2
	incorrect	**-4**	**-8**	-8	**-12**	**-32**

(c) Scoring policies of SV-COMP 2014, 2015, and 2016. Changing scores are shown in bold.

Figure 4.1 SV-COMP over the years: number of participants, number of verification tasks, scoring policy.

Table 4.4 *Overall* competition ranks for SV-COMP'14–'16 under the scoring policies of SV-COMP'14–'16.

Year		**1st place (score)**	**2nd place (score)**	**3rd place (score)**
Competition	**Scoring**			
2014	*2014*	CBMC (3,501)	CPAchecker (2,987)	LLBMC (1,843)
	2015	CBMC (3,052)	CPAchecker (2,961)	LLBMC (1,788)
	2016	CPAchecker (2,828)	LLBMC (1,514)	UFO (1,249)
2015	*2014*	CPAchecker (5,038)	SMACK (3,487)	CBMC (3,473)
	2015	CPAchecker (4,889)	SMACK (3,168)	UAutomizer (2,301)
	2016	CPAchecker (4,146)	SMACK (1,573)	PredatorHQ (1,169)
2016	*2014*	CBMC (6,669)	CPA-Seq (5,357)	ESBMC (5,129)
	2015	CBMC (6,122)	CPA-Seq (5,263)	ESBMC (4,965)
	2016	UAutomizer (4,843)	CPA-Seq (4,794)	SMACK (4,223)

4.3.2 Decisiveness-Reliability Plots

To better understand the competition results, we create scatter plots where each data point $\mathbf{v} = (c, i)$ represents a tool that gives c% correct answers and i% incorrect answers. Figure 4.2 shows such plots based on the verification tasks in SV-COMP'14, '15, and '16. Each data point marked by an unfilled circle $\circ$ represents one competing tool. The rectilinear distance $c + i$ from the origin gives a tool's *decisiveness*, i.e. the farther from the origin, the fewer times a tool reports "unknown". The angle enclosed by the horizontal axis and $\mathbf{v}$ gives a tool's *(un)reliability*, i.e. the wider the angle, the more often the tool gives incorrect answers. Thus, we call such plots *decisiveness-reliability plots* (DR-plots).

Discussion. Figure 4.2 shows DR-plots for the verification tasks in SV-COMP'14–'16:

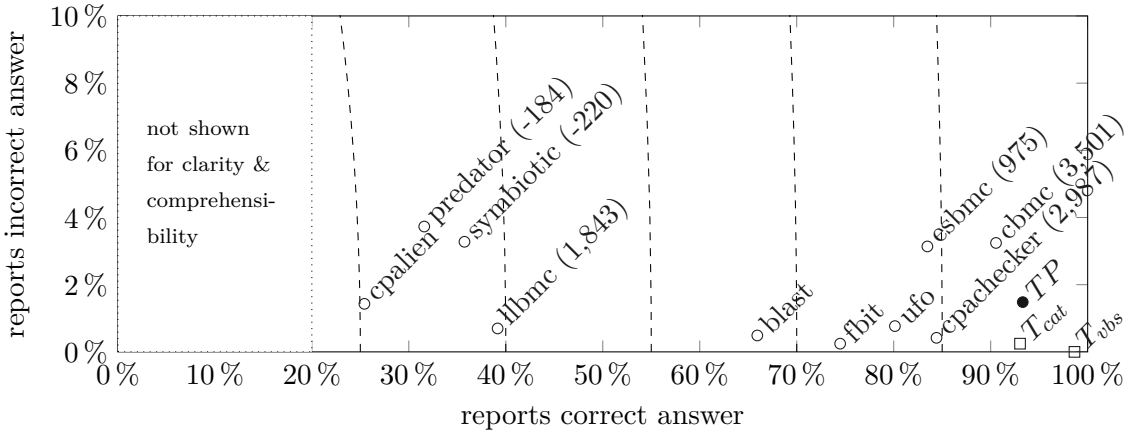
- *For 2014* (Figure 4.2a), all the tools are performing quite well on soundness: none of them gives more than 4% of incorrect answers. CPAchecker, ESBMC and CBMC are highly decisive tools, with more than 83% correct answers.

- *For 2015* (Figure 4.2b), the number of verification tasks more than doubled, and there is more variety in the results: We see that very reliable tools (BLAST, SMACK, and CPAchecker) are limited in decisiveness – they report "unknown" in more than 40% of cases. The bounded model checkers CBMC and ESBMC are more decisive at the cost of giving up to 10% incorrect answers.

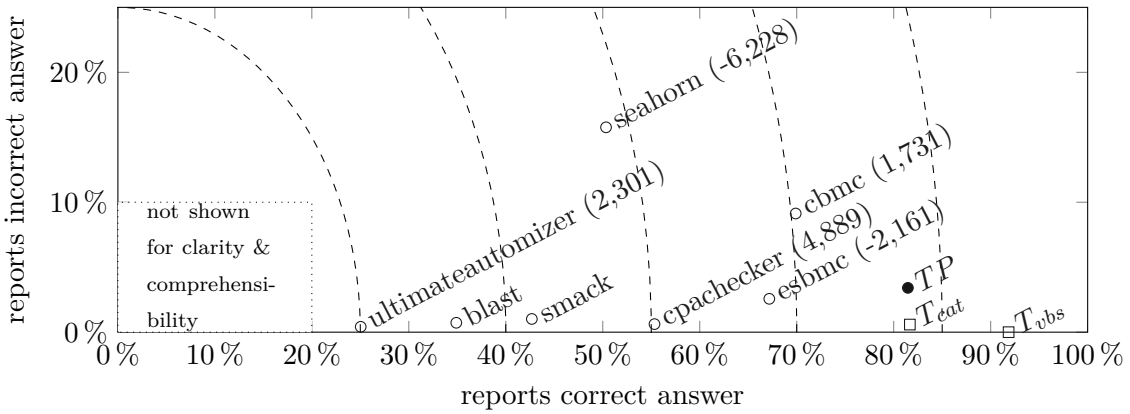
- *For 2016* (Figure 4.2c), there is again a close field of very reliable tools (CPAchecker, SMACK, and UltimateAutomizer) that give around 50% of correct answers and almost no incorrect answers. Bounded model checker CBMC is still highly decisive, but gives 6% of incorrect answers.

We also give *Overall* SV-COMP scores (where applicable) in parentheses. Clearly, tools close together in the DR-plot not necessarily have similar scores because of the different score weights prescribed by the SV-COMP scoring policy.

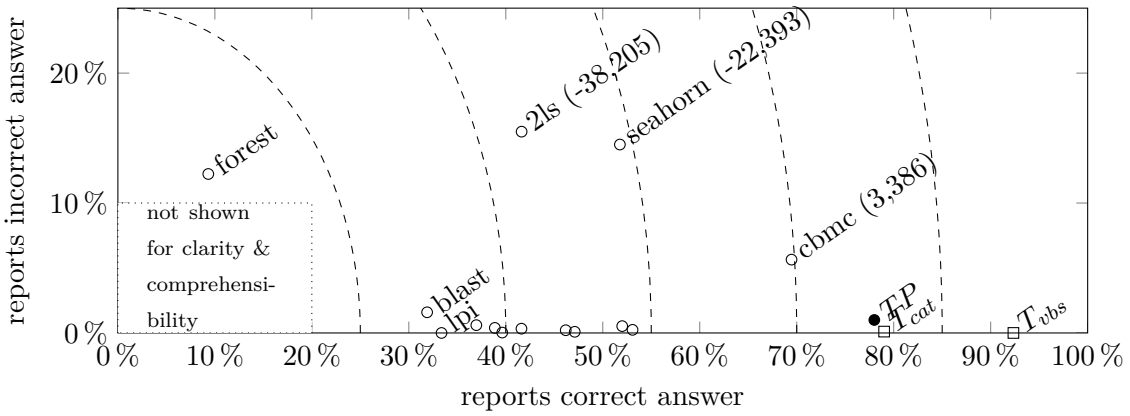
Referring back to Figures 4.2a–4.2c, we also show the theoretic strategies T_{cat} and T_{vbs} (see Section 4.2.4) marked by a square $\square$. Both strategies illustrate that combining tools can yield an almost perfect solver, with ≥ 90% correct and 0% incorrect answers. (Note that these figures may give an overly optimistic picture – after all the benchmarks are supplied by the competition participants.) The results for T_{vbs} compared to T_{cat} indicate that leveraging not just the category winner, but making a per-task decision provides an advantage both in reliability and decisiveness. A useful portfolio would thus lie somewhere between CPAchecker, CBMC, T_{cat}, and T_{vbs}, i.e. improve upon the decisiveness of constituent tools while minimizing the number of incorrect answers.



(a) Decisiveness-reliability plot for SV-COMP'14.



(b) Decisiveness-reliability plot for SV-COMP'15.



(c) Decisiveness-reliability plot for SV-COMP'16.

Figure 4.2 Decisiveness-reliability plots for SV-COMP'14–'16. The horizontal axis gives the percentage of correct answers c, the vertical axis the number of incorrect answers i. Dashed lines connect points of equal decisiveness $c + i$. The *Overall* SV-COMP score is given (if available) in parentheses.

4.3.3 Evaluation of Our Portfolio Solver

We originally implemented the machine learning-based portfolio $\mathcal{TP}$ for SV-COMP'14 in our tool *Verifolio*.[4] When competition results for SV-COMP'15 became available, we successfully evaluated the existing techniques on the new data, and described our results in [DPVZ15]. We reused the portfolio construction published there to compute the additional results for SV-COMP'16. We present these both in terms of the traditional metrics used by the competition (SV-COMP score and medals) and $\mathcal{TP}$'s placement in DR-plots:

Setup. For our experiments we did not rebuild the infrastructure of SV-COMP, but use numeric results from held competitions to compare our portfolio approach against other tools. Following a standard practice in machine learning [Bis06], we randomly split the verification tasks of SV-COMP'*year* into a training set $train_{year}$ and a test set $test_{year}$ with a ratio of 60:40. We train $\mathcal{TP}$ on $train_{year}$ and evaluate it on $test_{year}$ by comparing it against other tools' results on $test_{year}$. As the partitioning into training and test sets is randomized, we conduct the experiment 10 times and report the arithmetic mean of all figures. Tables 4.3a–4.3c show the *Overall* SV-COMP scores, runtimes and medal counts (we show the detailed scores for each category in Tables 8.1–8.3 on page 173 in the Appendix, Section 8.D). The DR-plots in Figures 4.2a–4.2c show the portfolio marked by a filled circle •.

Discussion. First, we discuss our results in terms of *Overall* SV-COMP score and medals:

- For SV-COMP'14 (Table 4.3a), our portfolio $\mathcal{TP}$ overtakes the original *Overall* winner CBMC with 16% more points. It wins a total of seven medals (1/5/1 gold/silver/bronze) compared to CBMC's six medals (2/2/2).
- For SV-COMP'15 (Table 4.3b), $\mathcal{TP}$ is again the strongest tool, collecting 13% more points than the original *Overall* winner CPAchecker. Both CPAchecker and $\mathcal{TP}$ collect 8 medals, with CPAchecker's 2/1/5 against $\mathcal{TP}$'s 1/6/1.
- For SV-COMP'16 (Table 4.3c), $\mathcal{TP}$ beats the original *Overall* winner UltimateAutomizer, collecting 66% more points. $\mathcal{TP}$ collects 6 medals, compared to the original winner UltimateAutomizer with 2 medals (0/2/0) and the original runner-up CPA-Seq with 5 medals (2/1/2).

Second, we discuss the DR-plots in Figures 4.2a–4.2c. Our portfolio $\mathcal{TP}$ positions itself between CBMC, CPAchecker and the theoretic strategies T_{cat} and T_{vbs}. Furthermore, $\mathcal{TP}$ falls halfway between the concrete tools and idealized strategies. We think this is a promising result, but there is still room for future work. Here we invite the community

[4] `http://forsyte.at/software/verifolio`. Accessed 23 January 2018.

	blast	cbmc	cpa-checker	cpa-lien	esbmc	fbit	llbmc	ufo	$\mathcal{TP}$	T_{cat}	T_{vbs}
Overall	468	**1292**	**1235**	266	695	666	853	735	**1494**	1732	1840
	2066	**4991**	**1865**	776	4024	898	978	381	**2211**	1310	270
Medals	1/0/0	2/2/2	2/1/1	0/0/0	1/0/1	0/0/2	1/0/1	1/1/0	1/5/1	-	-

(a) *Overall* SV-COMP score, runtime and medal counts for SV-COMP'14.

	blast	cascade	cbmc	cpa-checker	predator-hp	smack	ultimate-kojak	ulcseq	$\mathcal{TP}$	T_{cat}	T_{vbs}
Overall	737	806	684	**2228**	389	**1542**	1215	273	**2511**	3231	3768
	4546	5146	11936	**6288**	96	**8727**	7979	12563	**6260**	4360	1882
Medals	1/0/0	0/0/0	1/1/1	2/1/5	1/0/1	2/1/1	0/2/0	0/0/0	1/6/1	-	-

(b) *Overall* SV-COMP score, runtime and medal counts for SV-COMP'15.

	cpa-bam	cpa-kind	cpa-refsel	cpa-seq	esbmc	esbmc-depthk	smack	uautomizer	$\mathcal{TP}$	T_{cat}	T_{vbs}
Overall	898	1678	1151	**1907**	1699	1283	1684	**1965**	**3269**	3800	4238
	11775	12587	10240	**12509**	8396	9920	14218	**11210**	**8544**	8883	2547
Medals	0/0/0	0/1/1	1/0/0	2/1/2	0/2/0	0/0/0	0/0/1	0/2/0	2/1/3	-	-

(c) *Overall* SV-COMP score, runtime and medal counts for SV-COMP'16.

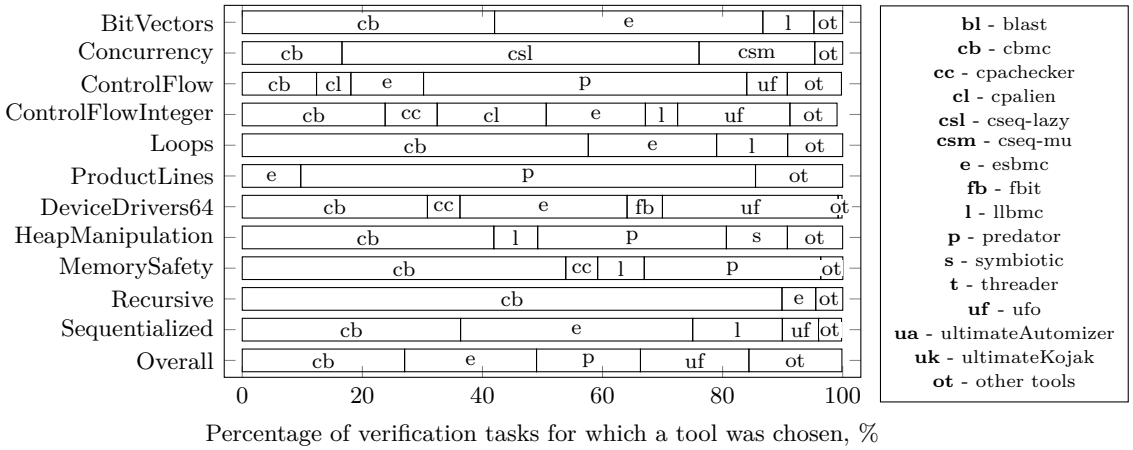
Figure 4.3 Experimental results for the eight best competition participants in *Overall*, plus our portfolio $\mathcal{TP}$ on *random subsets* of SV-COMP, given as arithmetic mean of 10 experiments on the respective test sets $test_{year}$. The two last columns show the idealized strategies T_{cat}, T_{vbs} (not competing, for comparison only). The first row shows the *Overall* SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The second row shows the number of gold/silver/bronze medals won in individual categories.

to contribute further feature definitions, learning techniques, portfolio setups, etc. to enhance this approach.

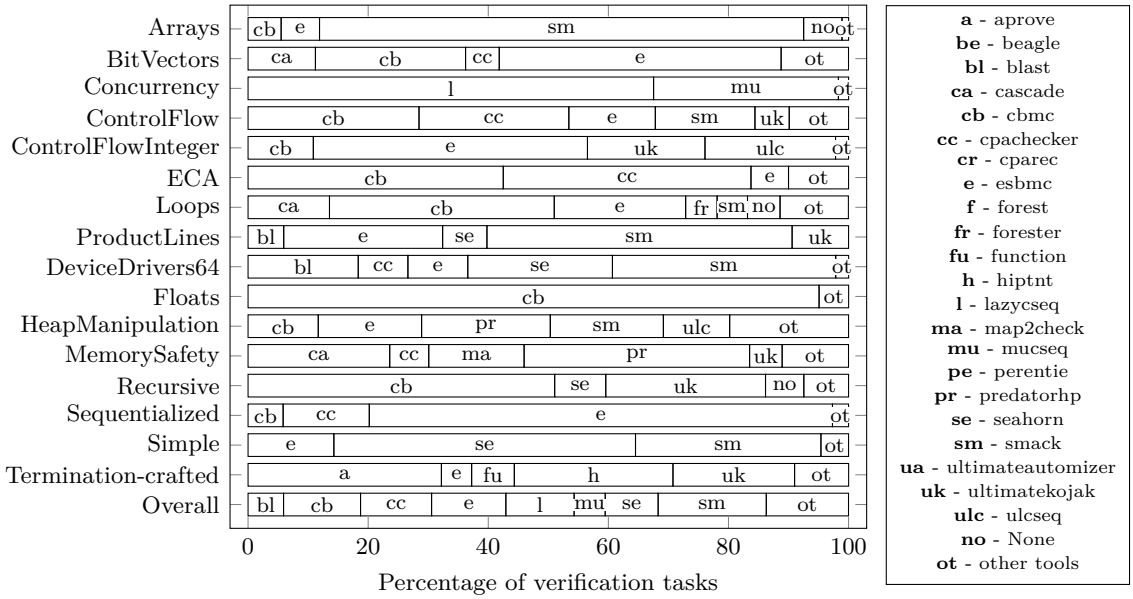
In the following we discuss three aspects of $\mathcal{TP}$'s behavior in greater detail: The runtime overhead of feature extraction, diversity in the tools chosen by $\mathcal{TP}$, and cases in which $\mathcal{TP}$ selects a tool that gives the wrong answer.

Constituent Verifiers Employed by our Portfolio

Our results could suggest that $\mathcal{TP}$ implements a trade-off between CPAchecker's conservative-and-sound and CBMC's decisive-but-sometimes-unsound approach. Contrarily, our experiments show that significantly more tools get selected by our portfolio solver (cf. Figures 4.4a–4.4c). Additionally, we find that our approach is able to select domain-specific solvers: For example, in the Concurrency category, $\mathcal{TP}$ almost exclusively selects variants of CSeq (and for 2016 also CIVL), which are specifically aimed at concurrent problems.

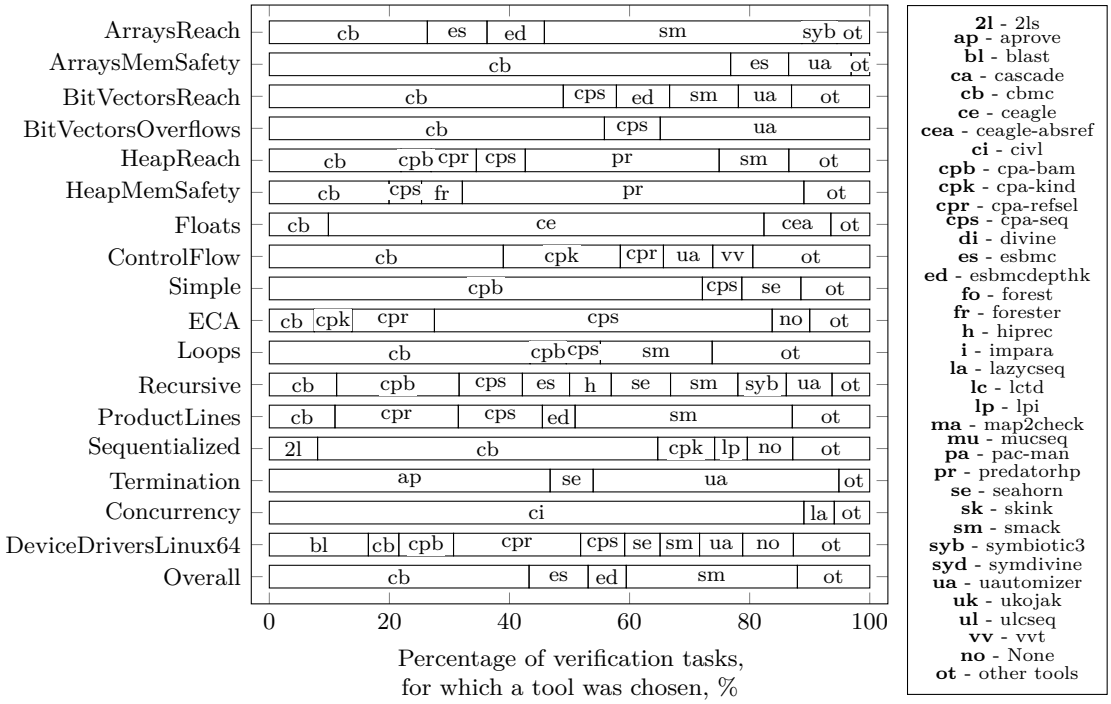


(a) Tools selected by $\mathcal{TP}$ for SV-COMP'14.



(b) Tools selected by $\mathcal{TP}$ for SV-COMP'15.

Figure 4.4 Compositionality of the portfolio $\mathcal{TP}$: Constituent tools selected per competition category. Tools selected in less than 5% of cases are summarized under label "other tools".



(c) Tools selected by $\mathcal{TP}$ for SV-COMP'16.

Figure 4.4 Compositionality of the portfolio $\mathcal{TP}$: Constituent tools selected per competition category. Tools selected in less than 5% of cases are summarized under label "other tools".

Wrong Predictions

We manually investigated cases of wrong predictions made by the portfolio solver. We identify i. imperfect tools and ii. data imbalances as the two main reasons for bad predictions. In the following, we discuss them in more detail:

Imperfect tools. In SV-COMP, many unsafe ($\text{ExpAns}(v) = \text{false}$) benchmarks are manually derived from their safe ($\text{ExpAns}(v') = \text{true}$) counterparts with minor changes (e.g. flipping a comparison operator). Two such files have similar or even the same metrics ($\mathbf{x}(v) \approx \mathbf{x}(v')$), but imperfect tools don't solve or fail to solve both of them ($L_t(v) \neq L_t(v')$). In particular, tools in SV-COMP are

- **unsound:** for example, in SV-COMP'16 the benchmarks `loops/count_up_down_{true,false}-unreach-call_true-termination.i` differ in a single comparison operator, namely equality is changed to inequality. Tool BLAST solves the unsafe task correctly, and the safe one incorrectly (i.e. gives the same answer for both).
- **buggy:** similarly to above, in SV-COMP'16 benchmarks `recursive-simple/`

`fibo_2calls_10_{true,false}-unreach-call.c` differ in a single comparison operator. The tool Forest solves the safe task correctly, and crashes on the unsafe one.

- **incomplete:** the benchmarks `ldv-regression/mutex_lock_int.c_{true,false}-unreach-call_1.i`, also taken from SV-COMP'16, differ in a single function call, namely `mutex_unlock()` is changed to `mutex_lock()`. The tool CASCADE correctly solves the safe benchmark, and answers unknown for the unsafe one.

This is unfortunate, as machine learning builds on the following assumption: Given two feature vectors $\mathbf{x}$ and $\mathbf{x}'$ with actual labels y and y', if $\mathbf{x} \approx \mathbf{x}'$ (where approximate equality $\approx$ is defined by the machine learning procedure), then $y = y'$. This assumption is violated in the cases illustrated above.

Counter-measures: In all cases, our metrics do not distinguish the given benchmark pairs. To mitigate these results, the obvious solution is to improve the participating tools. To solve the issue on the side of our portfolio, we believe more expensive analyses would have to be implemented for feature extraction. However, these analyses would i. be equivalent to correctly solving the verification problem directly and ii. increase the overhead spent on feature extraction. A practical portfolio is thus limited by the inconsistencies exhibited by its individual tools.

Data imbalances. In our training data we can find feature vectors on which, for a given tool t, e.g. the number of correct answers noticeably outweighs the number of incorrect answers. This corresponds to the problem of data imbalances (cf. Section 4.2.1), which leads to the following bias in machine learning: For a verification tool that is correct most of the time, machine learning prefers the error of predicting that the tool is correct (when in fact incorrect) over the error that a tool is incorrect (when in fact correct). In other words, "good" tools are predicted to be even "better".

Counter-measures: As described in Section 4.2.1, the standard technique to overcome data imbalances are weighting functions. Discovering data imbalances and countering multiple of them in a single weighting function is a hard problem. Our weighting function (cf. Section 4.2.3) mitigates this issue by compensating several imbalances that we identified in our training data, and was empirically tuned to improve results while staying general.

Overhead of Feature Extraction

By construction, our portfolio incurs an overhead for feature extraction and prediction before actually executing the selected tool. In our experiments, we measured this overhead to take a median time of $\tilde{x}_{\text{features}} = 0.5$ seconds for feature extraction and $\tilde{x}_{\text{prediction}} = 0.5$ seconds for prediction. We find this overhead to be negligible, when compared to verification time. For example, the *Overall* winner of SV-COMP'16, UltimateAutomizer,

exhibits a median verification time of $\tilde{x}^{\text{ua}}_{\text{verif}} = 24.9$ seconds computed over all tasks in SV-COMP'16.

Note that these numbers are not directly comparable, as $\tilde{x}^{\text{ua}}_{\text{verif}}$ stems from the SV-COMP results on the SV-COMP cluster, whereas $\tilde{x}_t$ for $t \in \{\text{features}, \text{prediction}\}$ was measured during our own experiments on a different system.

CHAPTER 5

Role-Based Heuristics for Systematic Predicate Abstraction

In this chapter we explore another application of variable roles in software verification, namely we present a method for systematic specification of heuristics for generating program-specific abstractions. To construct a program abstraction, a verification tool needs information about variables and data structures used in the program. Our algorithm collects this information using heuristics, which are based on variable roles.

As a case study, we use variable roles to specify heuristics for the model checker Eldarica [RHK13]. As the core procedure, Eldarica applies predicate abstraction and counterexample-guided abstraction refinement. Eldarica requires a program to be translated to a set of logic formulae of specific form called *Horn clauses*. We describe the basics of the Eldarica's functioning in Section 5.1.

In Section 5.2 we describe our heuristics for Eldarica, which use the roles defined in Section 3.3.3. The heuristics are motivated by Eldarica's previous built-in heuristics and typical verification benchmarks from the literature and SV-COMP.

We implement a prototype tool which uses the specification of role-based heuristics to guide Eldarica to a suitable abstraction. In Section 5.3 we describe the experimental evaluation of our algorithm.

5.1 Software Model Checking with Horn Clauses

We start with an outline of the algorithm implemented by Eldarica [RHK13] as well as a number of other tools [GLPR12, HB12] which reduce checking safety properties of a software program to checking the satisfiability of *Horn constraints* (or *Horn clauses*) to which a program is translated.

5.1.1 Definitions

An *atomic formula* a is a formula that contains no logical connectives.

A *literal* ℓ is an atomic formula a or its negation $\neg a$. In the rest of the definitions we will use the (possibly indexed) symbol ℓ to denote a literal.

A *clause* is a finite disjunction of literals $\ell_1 \vee \ldots \vee \ell_n$.

A *Horn clause* is a clause with at most one unnegated literal: $\ell_1 \wedge \ldots \wedge \ell_n \rightarrow \ell_0$. The unnegated literal ℓ_0 forms the *head* of the clause, and the negative symbols $\ell_1 \wedge \ldots \wedge \ell_n$ form the *body* of the clause.

A Horn clause is *linear* if it contains at most one relation symbol in its body.

A formula is in *disjunctive normal form* (DNF) if it is a disjunction of one or more conjunctions of one or more literals: $(\ell_1^1 \wedge \ldots \wedge \ell_n^1) \vee \ldots \vee (\ell_1^k \wedge \ldots \wedge \ell_n^k)$.

5.1.2 Translation of a Program to Horn Constraints

Eldarica has a frontend which translates a C or C++ program to a set of Horn clauses HC. In the translation, a Horn clause takes the form

$$B_1 \wedge \cdots \wedge B_n \wedge \varphi \rightarrow H,$$

where

- B_i is an application $Inv_i(t_1^i, \ldots, t_k^i)$ of a relation symbol $Inv_i \in \mathcal{R}$ to first-order terms $t_1^i, \ldots, t_k^i$. Each relation symbol Inv_i corresponds to a control location i in the program;
- φ is a relation-free constraint (i.e. a first-order formula) over variables occurring in the literals B_i;

  In our experiments, φ is always a formula in quantifier-free Presburger arithmetic, but extension to other theories (e.g., arrays) is possible;
- H is either an application $Inv(t_1, \ldots, t_k)$ of a symbol $Inv \in \mathcal{R}$ to first-order terms, or *false*.

Types of Horn Clauses in a Program Translation. Each Horn clause expresses one of the following:

1. A pre-condition $Pre(\bar{s_e}) \rightarrow Inv_e(\bar{s_e})$ for the program entry point e, where $Inv_e \in \mathcal{R}$ is a relation symbol and Pre is a first-order formula; and the formulae $Pre(\bar{s}_e)$ and $Inv(\bar{s}_e)$ are defined over a vector of variables $\bar{s}_e = (v_1^e, \ldots, v_k^e)$;
2. An *inductiveness condition* $T(\bar{s}_c, \bar{s}_d) \wedge Inv_c(\bar{s}_c) \rightarrow Inv_d(\bar{s}_d)$, where T is a first-order formula encoding the transition relation between control locations c and d;

3. A safety assertion $Inv_c(\bar{s}_c) \rightarrow P(\bar{s}_c)$, where P is a first-order formula encoding an assertion condition at control location 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_i can be interpreted in such a way that all clauses become valid.

5.1.3 Solving Horn Clauses with Predicate Abstraction

In Section 2.1 we gave preliminaries on the predicate abstraction and Craig interpolation techniques as well as the CEGAR algorithm. In this section we describe a generalisation of predicate abstraction from programs to Horn clauses. We focus in this thesis on *linear* Horn clauses, which are sufficient to represent sequential programs, provided that functions are handled via inlining, as done in our experiments.

Similarly to the predicate abstraction technique which we described in Section 2.1.2, the predicate abstraction of Horn clauses is defined with a set of predicates P. Assume, a set of program locations Loc is given; then a relation symbol $Inv_i \in \mathcal{R}$ is associated with each location $i \in Loc$. Next, given a subset of predicates $Q \subseteq P$, let us denote by $DNF(Q)$ the set of DNF formulae over Q.

Model checkers like HSF [GLPR12] or Eldarica [RHK13] construct a solution

$$Sol : \mathcal{R} \rightarrow DNF(P)$$

of a set of Horn clauses HC, which assigns to each relation symbol $Inv_i \in \mathcal{R}$ a formula $f \in DNF(P)$ in disjunctive normal form. To this end, a Horn solver maintains a mapping

$$\Pi : \mathcal{R} \rightarrow \mathcal{P}(P)$$

from relation symbols $Inv_i \in \mathcal{R}$ to finite subsets $P_i \subseteq P$ of predicates P.

To construct a mapping Π, a Horn solver goes through the following steps:

1. The solver starts from some initial mapping $\Pi = \Pi_0$; for instance, mapping every relation symbol to an empty set of predicates.

2. The solver will then attempt to find a solution Sol for the system of Horn clauses HC, s.t.
$$\forall i \in Loc.\ Sol(Inv_i) \in DNF(\Pi(Inv_i)).$$
For example, in Appendix, Section 8.B we give an algorithm for creating a solution, which we take from [RHK13]. The construction of a solution can fail because some assertion clause
$$Inv(\bar{t}) \rightarrow P(\bar{t})$$
is violated during the construction. In this case, the algorithm of Eldarica [RHK13] extracts a counterexample C (in particular, C is in the form of a resolution proof for deriving *false* from the set HC).

3. A theorem prover is used to check the satisfiability of the counterexample C. If C is satisfiable, then the set HC is unsolvable, with the counterexample C used as an explanation;

   Otherwise, additional predicates $\Pi^{ref} : \mathcal{R} \to \mathcal{P}(P)$ are generated from C using Craig interpolation, leading to an extended mapping

   $$\forall i \in Loc.\ \Pi'(Inv_i) = \Pi(Inv_i) \cup \Pi^{ref}(Inv_i),$$

   and the algorithm continues from Step 1.

The procedure has two main parameters that can be used to tune the abstraction process:

- **initial predicates** Π_0 for predicate abstraction (see Step 1 above);
- **interpolation templates** T that guide Craig interpolation towards meaningful predicates during abstraction refinement (see Section 5.1.4).

The pair (Π_0, T) can be computed with the help of variable roles, as outlined in the Section 3.1.3. It is important to note that neither parameter has any effect on *soundness* of a model checker, only termination is affected.

We will now discuss interpolation templates in more detail.

5.1.4 Craig Interpolation with Templates

For every extracted counterexample, 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. Eldarica uses *interpolation abstraction* [LRS16] 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 [LRS16].

In the scope of this thesis, we focus on templates in the form of *terms.* For an example of interpolation with templates in Eldarica we refer the reader to the Example 2.1.7 on page 27, which we have previously given in Section 2.1.4.

Eldarica provides an interface to annotate programs to express preference of certain interpolants. For instance, line 6 of the code in Fig. 1.2 on page 9 in Section 1.4.1 can be annotated to express that the differences `i-k` and `j-k` are preferred templates:

```
4 int k, /*@ terms_tpl {i-k} @*/ i, /*@ terms_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 [LRS16].

5.2 Role-Based Predicate Abstraction

We will now describe heuristics which we devise for the ELDARICA model checker in the form of initial predicates and predicate templates. To define these heuristics, we will use the 5 variable roles which we defined in the Section 3.3.3, namely *assertion parameter*, *dynamic enumeration*, *extremum*, *local counter* and *parity*. We will list the heuristics by the roles used in them.

5.2.1 Role-based Initial Predicates

First, we give the definitions of role-based predicates.

Assertion parameter. For each assertion expression `Expr` (denoted as `assert_expr(Expr)`) with a literal `Pred`, our algorithm generates the predicate `PredStr` (denoted as `pred(PredStr)`, lines 273–274 in the listing below), where `PredStr` is a string representation of `Pred`.

```
273 pred(PredStr):- assert_expr(Expr), literal(Expr,Pred),
274   expr_str(Expr,PredStr).
```

We define the relation `expr_str(Expr,ExprStr)`, which computes the string representation `ExprStr` of an expression `Expr`, in Appendix, Section 8.A, lines 76–86.

Example 5.2.1. Recall the example in Fig. 3.27 on page 100, which represents the CFG and the corresponding logic program for the statement `if (i>=1) assert(cnt==1)`.

We have shown in Section 3.3.3, Example 3.3.34 on page 100 that the evaluation of the rule for the relation `assert_expr` (see Section 3.3.3, lines 247–248) computes the fact `assert_expr(94)`, where the node `94` corresponds to the expression `cnt==1`.

Next, the evaluation of the rule defining the relation `literal` (see Section 8.A, line 27) and of the rule defining the relation `expr_str` (Section 8.A, lines 76–86), infers the facts `literal(94,94)` and `expr_str(94,"cnt==1")` respectively.

Finally, the evaluation of the rule in lines 273–274 generates the predicate `cnt==1`, encoded as the fact `pred("cnt==1")`. ▲

Dynamic enumeration. For each *dynamic enumeration* variable `X` with parameter `Y` (encoded as `dyn_enum(X,Y)`), our algorithm generates the predicate `PredStr`

(lines 275–276 in the listing below), s.t. `Xname` and `Yname` are the identifiers of the variables `X` and `Y` respectively, and `PredStr` is `Xname==Yname`.

```
275 pred(PredStr):- dyn_enum(X,Y), name(X,Xname), name(Y,Yname),
276   PredStr=@concat(Xname,"==",Yname).
```

The term `@concat(Str`$_1$`,...,Str`$_n$`)` evaluates to the concatenation of the strings `Str`$_1$, ..., `Str`$_n$.

Example 5.2.2. Recall the example in Fig. 3.28 on page 102, which shows the CFG and the corresponding logic program for the statements `id1=nondet_char();`, `id2=nondet_char();`, `id3=nondet_char();` the statements `max1=id1;`, `max2=id2;`, `max3=id3;` and the statements `max1=max3;`, `max2=max1;` and `max3=max2;`.

We have shown in Section 3.3.3, Example 3.3.35 on page 102 that the evaluation of the rule defining the role *dynamic enumeration* (see Section 3.3.3, lines 253–254) infers 9 facts `dyn_enum(node`$_{max1}$`,`$node_{id1}$`)`, `dyn_enum(node`$_{max1}$`,`$node_{id2}$`)`, `dyn_enum(node`$_{max1}$`,`$node_{id3}$`)`, etc.

Given these facts, the evaluation of the rule in lines 275–276, generates the predicates `max1==id1`, `max1==id2`, `max1==id3`, encoded with the facts `pred("max1==id1")`, `pred("max1==id2")` and `pred("max1==id3")`.

Similarly, the predicates `max2==id1`, `max2==id2`, `max2==id3` for the *dynamic enumeration* `max2` and the predicates `max3==id1`, `max3==id2`, `max3==id3` for the *dynamic enumeration* `max3` are generated. ▲

Extremum. For each variable `X` which is *extremum* (encoded as `extremum(X)`) and *dynamic enumeration* with two distinct parameters `Y` and `Z` (encoded as `dyn_enum(X,Y)`, `dyn_enum(X,Z)` and the fact `X!=Y`), our algorithm generates the predicate `PredStr` (lines 277–278 in the listing below), s.t. the identifiers of the variables `X` and `Y` are `Xname` and `Yname` respectively, and `PredStr` is `Yname<Zname`.

```
277 pred(PredStr):- extremum(X), dyn_enum(X,Y), dyn_enum(X,Z), Y!=Z,
278   name(Y,Yname), name(Z,Zname), PredStr=@concat(Yname,"<",Zname).
```

Example 5.2.3. Consider again the example in Fig. 3.28 on page 102.

We have shown in Section 3.3.3, Example 3.3.31 on page 96, that the variable `max3` has the role *extremum*, encoded as `extremum(node`$_{max3}$`)`.

Next, in Example 5.2.2 we have shown how the facts `dyn_enum(max3,id1)`, `dyn_enum(max3, id2)` and `dyn_enum(max3,id3)` are inferred for *dynamic enumeration* `max3`.

Given these facts, the evaluation of the rule in lines 277–278 generates the predicates `id1<id2`, `id2<id1`, `id1<id3`, `id3<id1`, `id2<id3` and `id3<id2`, encoded with the facts `pred("id1<id2")`, `pred("id2<id1")`, `pred("id1<id3")`,

`pred("id3<id1")`, `pred("id2<id3")` and `pred("id3<id2")` respectively. ▲

5.2.2 Role-based Predicate Templates

Next, we give the definitions of role-based templates.

Local counter. For each pair of *local counters* `X` and `Y` which have same parameter `WhileStmt` (encoded as `local_cnt(X,WhileStmt)` and `local_cnt(Y, WhileStmt)` respectively), our algorithm generates the template `TplStr` (lines 279–280 in the listing below), s.t. the identifiers of the variables `X` and `Y` are `Xname` and `Yname` respectively, and `TplStr` is `Xname-Yname`.

```
279 tpl(TplStr):- local_cnt(X,WhileStmt), local_cnt(Y,WhileStmt),
280   X!=Y, name(X,Xname), name(Y,Yname), TplStr=@concat(Xname,"-",Yname).
```

Example 5.2.4. Recall the example in Fig. 3.25 on page 97, which shows the CFG and the corresponding logic program for the statement `for (k=0,i=0; i<n; i++,k++);`.

We have shown in Section 3.3.3, Example 3.3.32 on page 97 that the variables `i` and `k` have the role *local counter* in same loop, encoded with the facts `local_cnt(node`$_i$`, 12)` and `local_cnt(node`$_k$`,12)`, where the node `12` corresponds to the `while` statement (recall that our algorithm rewrites the `for` statement to a `while` loop).

Given these facts, the evaluation of the rule in lines 279–280 generates the predicate `i-k`, encoded with the fact `tpl("i-k")`. ▲

Parity. For each *parity* variable `X` with constant literal parameter `Num` the template `TplStr` is generated (lines 281–282 in the listing below), s.t. the identifier of `X` is `Xname`, the string representation of `Num` is `NumStr` and `TplStr` is `Xname%Val` (where `%` is the remainder operator in the C language).

```
281 tpl(TplStr):- parity(X,Num), name(X,Xname), val(Num,NumStr),
282   TplStr=@concat(Xname,"%",NumStr).
```

Example 5.2.5. Recall the example in Fig. 3.26 on page 99, which shows the CFG and the corresponding logic program for the statement `for (i=0; i<1000000; i+=2);`.

We have shown in Section 3.3.3, Example 3.3.33 on page 99 that the variable `i` has role *parity* with parameter `2`, encoded as `parity(node`$_i$`,16)`, where the node `16` corresponds to the constant literal `2`.

Given this fact, the evaluation of the rule in lines 281–282 generates the predicate `i%2`, encoded as `tpl("i%2")`. ▲

5.3 Evaluation

We implemented our approach in a prototype tool and evaluated the tool on altogether 549 C benchmarks.[1]

Benchmarks. Table 5.1 lists the benchmarks and gives their characteristics. Specifically, the benchmarks contain (listed in the same order as in Table 5.1):

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*[2] We excluded 234 duplicate benchmarks contained in SV-COMP CFI, and 2 benchmarks, for which the transition relations cannot be expressed with Presburger arithmetic;
4. Simplified versions[3] of the benchmarks of tool llrêve for automated program equivalence checking [FGK+14];
5. Loop invariant generation benchmarks of the verication tool HOLA [DDLM13].

Tools for comparison. In our experiments we compare the following tools:

- We evaluate the following configurations of Eldarica:
  - Two unmodified versions of Eldarica: without interpolation abstraction (to which we refer by Eld) and with interpolation abstraction and built-in templates (Eld+B),
  - Two versions of Eldarica enhanced with roles, with interpolation abstraction: without built-in templates (Eld+R) and with built-in templates (Eld+BR).

  Table 5.2 lists different choices for the parameters Π_0 and T described in Section 5.1.3.
- As a baseline we also compare Eldarica to SMT solvers Z3 [dMB08] and Spacer [KGCC13].

  We could not compare to the Duality engine of Z3 because of a bug in Duality, which was not fixed by the time of submitting the paper with our results [DRZ17].

[1] The tool, the set of used benchmarks and the results of our evaluation are available at `http://forsyte.at/software/demy/nfm17.tar.gz`. Accessed 23 January 2018.

[2] `http://map.uniroma2.it/vcgen/benchmark320.tar.gz`. Accessed 23 January 2018.

[3] The original benchmarks are accessible at `http://formal.iti.kit.edu/projects/improve/reve` and `https://www.matul.de/reve`. Accessed 23 January 2018.

Table 5.1 Characteristics of the benchmarks used for experimental evaluation

#	Name	Number of files			Size,
		Total	Safe	Unsafe	KLOC
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 5.2 Different configurations of Eldarica: T_{Eld} denotes the templates generated by built-in heuristics of Eldarica.

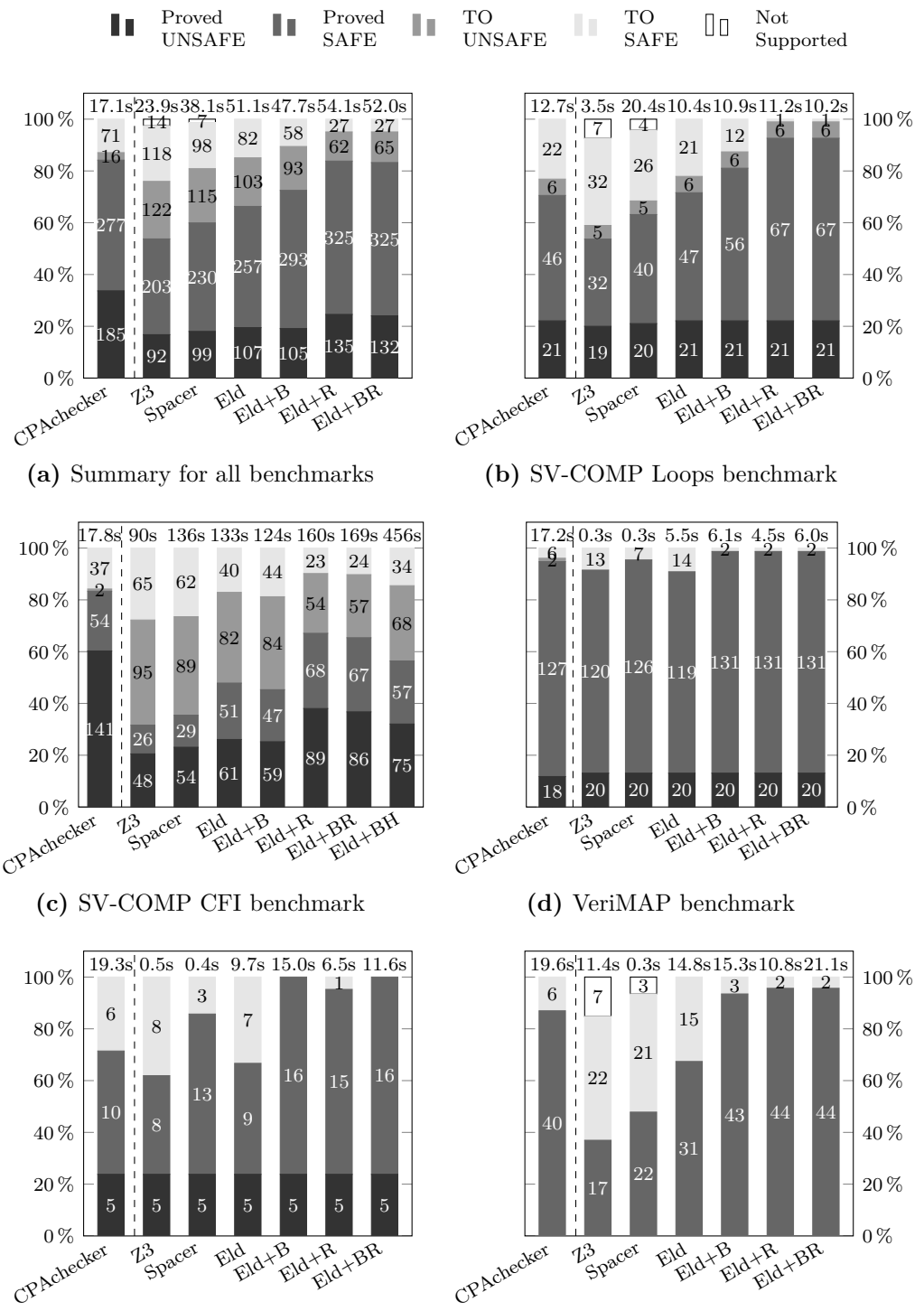
Name	Π_0	T
Eld	$\emptyset$	$\emptyset$
Eld+B	$\emptyset$	T_{Eld}
Eld+R	Π_{roles}	T_{roles}
Eld+BR	Π_{roles}	$T_{roles} \cup T_{Eld}$

- 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. 5.1, which shows the number of solved and unsolved tasks, with safe and unsafe tasks counted separately. Specifically, Fig. 5.1a gives a summary for all benchmarks, and Figures 5.1b-5.1f 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 respectively. 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. In particular, the mean runtime of Eld+R on the subset of the benchmarks solved by all tools is 1.3 times higher than the mean runtime of Eld and 1.2 times higher than the mean runtime of Eld+B. The mean runtime of the configuration Eld+BR is 1.1 times higher than the mean runtime of Eld+R on the same subset of benchmarks.



(a) Summary for all benchmarks

(b) SV-COMP Loops benchmark

(c) SV-COMP CFI benchmark

(d) VeriMAP benchmark

(e) Llreve benchmark

(f) HOLA benchmark

Figure 5.1 Bar plots comparing the percentage of proved tasks for CPAchecker, Z3, Spacer and different Eldarica configurations. Inside each bar is the percentage of the respective answers. On top of each bar is the mean runtime computed *without timeouts* (for solved tasks).

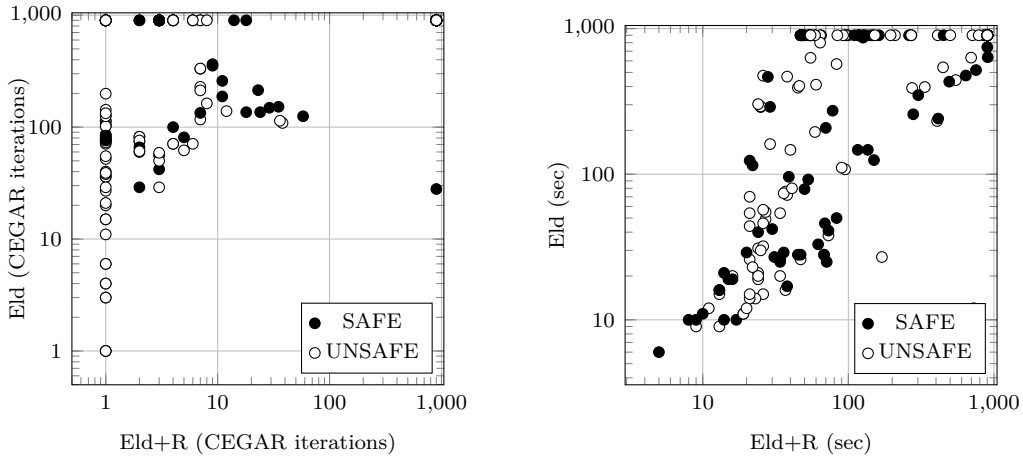


Figure 5.2 Scatter plots comparing the number of CEGAR iterations and runtime, both in logarithmic scale, of configurations Eld+R and Eld for benchmark SV-COMP CFI. The mean runtime of Eld+R is 1.5 times smaller than that of Eld, and the average number of CEGAR iterations of Eld+R is 19.0 times smaller than that of Eld, the four values calculated on the tasks solved by both Eld and Eld+R.

For the benchmarks SV-COMP CFI we observe a significant speedup of Eld+R, as shown in Fig. 5.2. 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. 3.6a on page 57. Note that in Fig. 5.2 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. 5.1c). We observe that Eld+BH solves 12.8% more unsafe and 9.0% more safe tasks than Eld+B.

The runtime of Eldarica is in average comparable to the runtime of the SMT solvers on the set of the benchmarks solved by all tools: the mean runtime of Eld is 1.2 higher than the mean runtime of Z3 and 1.6 lower than the mean runtime of Spacer.

To conclude, Eld+R does not considerably 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. 5.1 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. 5.1c 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[4]. We note that a small number of tasks in benchmarks SV-COMP Loops and HOLA cannot be processed by Z3 and Spacer 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. 5.1. We observe that, on one hand, all configurations of Eldarica outperform both Z3 and Spacer in the number of solved tasks, in particular 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.*

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

CHAPTER 6

Related Work

In this chapter we discuss the work related to variable roles and to the two applications of variable roles which we explored. We structure it as follows: First, we describe the work from different fields, where, similarly to variable roles, information is used about usage patterns of variables. Then, we discuss the work related to the formalisation of variable roles. In particular, we consider the related work in the theory of types and in program query languages. Finally, we briefly speak about the work related to portfolio solvers and about techniques aimed at choosing interpolants containing the *suitable* predicates.

6.1 Variable Usage Patterns

6.1.1 Teaching Programming Languages, Program Visualisation, Pattern Recognision

The notion of *variable roles*, as patterns of how variables are initialised and updated, is introduced by Sajaniemi et al. [Saj02], with a verbal definition of nine variable roles for sorting algorithms from textbooks. The described roles are applied in teaching of programming languages [SK05] and program visualisation [SK03]. Bishop et al. [BJ05] implement a tool which applies static analysis techniques to automatically check role annotations in Java code. Taherkhani [Tah10] uses variable roles in a decision tree classifier to recognise sorting algorithms. This field of work serves as a starting point and inspiration for the thesis; we come up with a more general set of variable roles for practical open-source programs, provide a framework to formally specify and automatically infer the roles, and make an extensive evaluation of our method on a large set of real-world programs.

6.1.2 Bug Finding

The commercial bug-finding tool Coverity[1], evolved from an academic project [ECH+01], uses so called *programmer's beliefs* – propositional statements about variables and functions. Using static analysis, it generates two kinds of statements. Beliefs of the first kind are sound statements which follow from the requirements of safety, non-redundancy, and reachability of the code, for example "a pointer is not null". The second kind of beliefs are derived from the source code using statistical methods, for example "the calls to functions f() and g() should be paired". This second type of rules, automatically inferred from source code, is used as heuristics for ranking possible bugs. Here bugs are violations of user-specified rules.

In addition, a language for an explicit specification of rules is devised [HCXE02, ECCH00], which is similar in spirit to our specification language for specifying variable roles. On one hand, our formalism is more expressive than the specification language of [HCXE02], since the former uses set operations and allows to reason about multiple roles assigned to a single variable and multiple variables assigned same role, which the latter does not allow. On the other hand, the implementation of Coverity is no doubt more efficient and scales to millions of lines of source code. However, Coverity is a closed commercial tool.

6.1.3 Software Verification

Model checker CPAchecker chooses between explicit-value and BDD representation of a variable, based on the operations in which the variable is used [ABF+13]. To this end, Apel et al. extend the type system of C language with five types, called *domain types*, which overapproximate the sets of variables used in logical, equality comparison, arithmetic and bit operations and the set of variables that are used as loop counters respectively. The domain types are also used in CPAchecker for predicate selection [BLW15]: a heuristic assigns costs to predicates based on the costs of the variables they use, with the costs of variables assigned according to their domain types.

The domain types in [BLW15] and [ABF+13] can be viewed as a restricted class of variable roles. Differently from our work, where variable roles guide the generation of interpolants (see Chapter 5), the domain types are used in [BLW15] to choose the "best" interpolant from a set of already generated interpolants. In addition, our method generates role-based initial abstraction (namely, predicates), while the method of [BLW15] does not.

In contrast to domain types devised for optimising the model checker CPAchecker, we did not tie the concept of variable roles to a particular application beforehand. Therefore, we devise a more general set of variable roles, and, as a result, choose a more expressive specification formalism for roles. However, the price which we have to pay for a more involved analysis is that the approach of CPAchecker for choosing predicates is more scalable than our approach introduced in Chapter 5, since with the increase of the

[1] `http://www.coverity.com`. Accessed 23 January 2018.

Technique	**Specification language**	**Decidability of type inference**	**Features**
Refinement types	Regular tree grammar	No	Decidable type inference obtained with restrictions on refinements
Dependent + index types	Typed λ-calculus with existential and universal quantifiers	No	Manual type annotations combined with automatic type checking
Liquid types	Same as in dependent types, but λ-expressions use a *finite* set of predicates	Yes	SMT solver used to solve a system of implications between refinements
Type qualifiers	Propositional language	Yes	Inference rules specified implicitly via annotation of library functions
Semantic type qualifiers	Custom specification language which uses C expressions	Yes	Type invariants used for checking soundness of specification

Table 6.1 Summary of related work in type theory field.

number of role specifications or of the variables in program source code, the amount of generated predicates and templates can become too large and have a negative effect on the performance of Eldarica. In Chapter 5 we compare our prototype tool, which infers role-based source code annotations for Eldarica, to the default configuration of CPAchecker, and it would be an interesting future work to make a comparison to the configuration of CPAchecker amended with domain types.

Finally, we believe that the representation of roles as logic queries is more readable than rules in a type system, which are used in [ABF+13] to formalise domain types.

6.2 Type Theory

One possible way to formalise variable roles is to define a type system, where roles correspond to types and are inferred during type inference, the approach taken in domain types [ABF+13]. In this section we discuss the direction in type systems which extends type systems to assign more fine-grained types to program variables and expressions, rather than accept more programs as type correct. Below we list the directions of related work, a summary of which we give in Table 6.1.

Refinement types [Fre94] refine types with atomic propositions, for example, "list is not empty", "list is singleton", etc., the atomic propositions being organised in a lattice. Refinement types use intersection type theory [Pie91], where an expression has type $\sigma \wedge \tau$ when it has both type σ and type τ. Type inference for intersection types is in general undecidable, and to obtain decidable type inference for refinement

types, the intersection is restricted to refinements of the same base type, and the lattice of refinements is required to be finite. The specifications of refinement types are regular tree sets, and the inference reduces to abstract interpretation. The inference for variable roles is similar to the inference of refinement types: abstract transformers of datatype constructors in refinement types correspond to constraints (in the form of logic rules) for statements and expressions in which a variable is used, and a fixed point is computed for both systems. Our specification language of variable roles is though more expressive than regular tree grammar used in refinement types. In particular, the language of variable roles allows negation, which is not allowed in the refinement types.

Dependent types [Xi98] introduce existential and universal quantifiers into types, which allow to express invariants for complex data structures, e.g. red-black trees, and function pre- and post-conditions, e.g. "function `concat()` takes two lists of the length `n` and `m` respectively and returns a list of the length `n+m`". Dependent types are usually combined with *index types*, which refine base types with Boolean-valued expressions in some logic, for example "the value of the expression belongs to a given interval". Type inference for dependent types is in general undecidable, therefore, a combination of manual type annotations with automatic type checking is typically used. We intentionally chose a less expressive language for the specification language of variable roles for the sake of efficient inference of roles.

Liquid types [RKJ08] restrict dependent types to obtain a decidable inference procedure. Liquid types refine base types with the conjunctions of predicates instantiated from a fixed number of template linear inequalities with program variables, for example `x>0∧x<5`. The inference procedure for liquid types is based on predicate abstraction and reduces to computing a fixed point of a system of implications between refinements, and uses an SMT solver for its implementation. This work is complementary to our approach, since variable roles do not describe which *values* a variable can take, but rather in which *operations*, functions etc. a variable is used.

Type qualifiers [Fos02], similarly to refinement types, refine data types with atomic propositions, organised in a lattice, with the ordering imposed only on refined types with the same base type. Type inference of type qualifiers reduces to solving a system of inclusion constraints on lattice subsets. The main limitation of type qualifiers is that their semantics is specified implicitly via the annotation of the corresponding library functions, for example "when a library function `lock()` (respectively `unlock()`) is called, the mutex variable which is passed to it is assigned type *locked* (respectively *unlocked*)".

Semantic type qualifiers [CMM05] extend type qualifiers so that the specification is given explicitly in the form of rules. In addition, the implementation of semantic type qualifiers includes an interface for the specification of an invariant for the values taken by variables of a given type. Besides checking the specification for completeness and consistency, the invariant can be used to infer a set of rules

Tool	Specification language	Features
JQuery	Prolog-like language	Annotations needed to ensure termination
ASTLog	Prolog-like language with custom semantics	Light-weight analysis without fixed point computations
CodeQuest	Datalog	Queries translated to SQL and evaluated with an off-the-shelf engine
PQL	Custom imperative language	Queries translated to SQL and evaluated with a non-standard BDD-based algorithm
Doop	Datalog	Framework for logic-based specification of pointer analysis with an efficient implementation

Table 6.2 Summary of related work in the field of code query languages.

automatically. Examples of semantic qualifiers are positive and negative values, tainted pointers (i.e. "unchecked" pointers passed to the kernel by a user), pointers which uniquely reference some memory location, etc. Both variable roles and semantic type qualifiers follow an approach of separating the specification from the implementation. Automatic inference of the specification for variable roles from an invariant is an open interesting question, since for most variable roles it is not clear how to formulate the invariant. Here the same remark applies as for the liquid types: the variable roles specify how a variable is used rather than what it stores, for example, consider variable role *file descriptor* from Example 1.3.2.

6.3 Program Query Languages

An alternative to the rule-based definition of variable roles using type systems is a declarative definition in a specification language. Since variable roles describe the usage patterns of variables, roles can be formalised as queries on the syntactical structure of a program. In this section we will make an overview of the languages, designed for this purpose, which are called *program query* languages.

6.3.1 Logic Languages for Program Analysis

A common approach in the design of code query languages is to represent the abstract syntax tree (AST) of a program as a set of relations and to use logic programming to query these relations. On one hand, the specification of static analysis using a logic language is concise and is separated from its evaluation. On the other hand, the evaluation of such an analysis is time and space consuming. Below we give an overview of tools implementing different approaches to code query languages, which we summarise in Table 6.2.

JQuery. A state-of-the-art code query language for Java language is JQuery [JV03].

For query specification JQuery uses a logic programming language TyRuBa, similar to Prolog. Prolog is a Turing-complete language, but query evaluation in Prolog might not terminate and is inefficient due to repeated subcomputations of relations. TyRuba overcomes these problems with memoisation [CW96], i.e. re-uses intermediate results. A drawback is that the relations which should be memoised need annotations.

ASTLog. A light-weight alternative for code querying is the language ASTLog [C+97]. ASTLog achieves fast evaluation by restricting the queries to traversing the syntax tree without support for fixed point computation. ASTLog has syntax similar to Prolog. The semantics is different in that a query is evaluated on an AST subtree rooted in the node matching the query parameter, rather than on the whole database of facts. ASTLog is not expressive enough to express variable roles, since data-flow analysis needs fixed point computation.

CodeQuest and PQL. Since the default implementation of Prolog stores the relations in RAM, the analysis defined in Prolog does not scale to static analysis of real-world programs. An alternative approach is to store facts in a relational database on a disk. Examples of works following this approach are the code query languages CodeQuest [HVdM06] and PQL [MLL05]. In CodeQuest queries are specified in Datalog and translated to SQL extended with recursion [GP99], and then executed using an off-the-shelf efficient interpreter. In PQL, aimed for finding bugs in Java programs, queries are specified in an own imperative language and translated to Datalog. Similarly to CodeQuest, PQL uses a database to evaluate queries, but implements a non-standard BDD-based evaluation algorithm [WACL05].

Doop. An efficient tool Doop [BS09] for pointer analysis uses Datalog for query specification, but does not build on top of a database. Doop drastically reduces evaluation times for queries using manually introduced query optimisations, in particular indexes and defining the order of joins in queries.

6.3.2 XML for Representing Code Structure

A different direction in code query languages is to represent AST as an XML[2], see e.g. JavaML [Bad00]. Such a representation allows to evaluate code queries and perform other static analyses using a rich infrastructure of XML-related tools and techniques. Specifically, XPath[3] is a declarative language, targeted at writing queries on XML trees using a compact expression-like syntax. Similarly to ASTLog, XPath evaluates a query on a subtree rooted in the node matching its parameter and can not express a system of recursive constraints. A Turing-complete functional language XQuery[4,5] is extention

[2] `https://www.w3.org/TR/xml/`. Accessed 24 January 2018.

[3] `https://www.w3.org/TR/xpath-3/`. Accessed 24 January 2018.

[4] `https://www.w3.org/TR/xquery-3/`. Accessed 24 January 2018.

[5] The formal semantics of XQuery 1.0 and XPath 2.0 is described at `https://www.w3.org/TR/query-algebra/`. Accessed 24 January 2018.

of XPath. A disadvantage of XQuery compared to a specification written in a logic language is that the former is much more lengthy.

6.3.3 State Automata-Based Approach

Finally, an example of a code query languages neither based on logic programming, nor on XML, is Metal [ECCH00, HCXE02]. Implemented as a compiler extension, the framework evolved in the bug-finding tool Coverity, which we already mentioned in Section 6.1. A specification in metal describes a state machine, with a state assigned to each variable, and transitions fired when a pattern is matched. Transitions, however, depend on the state of not more than one variable, therefore, recursively defined specifications are not allowed. The framework allows for both flow-sensitive and flow-insensitive analyses. The syntax of the Metal language is not specified in [ECCH00, HCXE02], but as far as we can judge, Metal is as expressive as the specification language of variable roles, modulo the absence of recursively defined state transitions in Metal, which is a not fundamental restriction, but forced due to performance reasons. The translation between a specification in Metal an a specification of a variable role be done as follows: the states of variables correspond to relations, and state transitions correspond to rules, the pattern defining the body of the rule.

6.4 Portfolio Solvers

Portfolio solvers have been successful in combinatorially cleaner domains such as SAT solving [XHHL08, KMS+11, Rou12], quantified Boolean satisfiability (QSAT) [SM07, PT07, PT09], answer set programming (ASP) [GKK+11, MPR12], and various constraint satisfaction problems (CSP) [LL98, GS01, OHH+08]. In contrast to software verification, in these areas constituent tools are usually assumed to be correct.

A machine-learning based method for selecting model checkers was previously introduced in [TKK+14]. Similar to our work, the authors use SVM classification with weights (cf. Section 4.2.1). Our approach is novel in the following ways:

1. The results in [TKK+14] are not reproducible because i. the benchmark is not publicly available, ii. the verification properties are not described, and iii. the weighting function – in our experience crucial for good predictions – is not documented.

2. We demonstrate the continued viability of our approach by applying it to new results of recent SV-COMP editions.

3. We use a larger set of verification tools (35 tools vs. 3). Our benchmark is not restricted to device drivers and is >10 times larger (56 MLOC vs. 4 MLOC in [TKK+14]).

4. In contrast to structural metrics of [TKK+14] our metrics are computed using data-flow analysis. Based on tool designer reports (Table 4.1) we believe that they have

superior predictive power. Precise comparison is difficult due to non-reproducibility of [TKK+14].

6.5 Choosing Interpolants with Suitable Predicates

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.

In Section 6.1.3 we have already compared our method for defining role-based heuristics in order to find *suitable* predicates (see Chapter 5) to *domain types* which were used in CPAchecker in a heuristic for choosing interpolants [BLW15].

For a further detailed survey of the techniques for choosing interpolants containing *suitable* predicates we refer the reader to the work which implements in Eldarica one of such techniques [LRS16].

CHAPTER 7

Future Work and Conclusions

7.1 Summary of Contributions

We have formulated the major contributions of our thesis in Section 1.7. In this section we discuss the contributions in more detail.

1. We give a *formal* definition of the concept of variable roles for imperative programming languages. As we mention in our discussion of related literature (see Section 6.1), concepts similar to the notion of variable roles have been used in different fields, such as software verification [BLW15], program understanding and teaching programming languages [Saj02] and bug finding [ECH+01]. However, to the best of our knowledge, our work is the first one where the concept of variable role is stated formally.

   a) We do a case study on a comprehensive code base and devise a classification of variable roles which capture the typical usage patterns of variables. In particular, we formulate 18 domain-independent variable roles, using a collection of benchmarks from industry (see Section 3.1.1). The classifications suggested by previous methods are either very coarse [BLW15], or targeted for a specific application domain for roles [ECH+01], or derived from programs of a specific kind and hence are restricted to a narrow set of variable usage patterns [Saj02]. The variable roles formulated in this thesis are more general than the roles suggested in the previous methods and are derived from practical open-source benchmarks.

   b) We propose a concise specification formalism for variable roles based on logic programming, which at the same time lends itself as a technique for automatic inference of roles (see Chapter 3). Our specification framework uses Datalog rules to define variable roles. We believe that Datalog rules are more concise and understandable than type system rules of [BLW15]. Another advantage of

Datalog is the possibility to evaluate role definitions using off-the-shelf logic engines.

2. We explore the application of variable roles in software verification:
   a) First, we identify 12 additional variable roles important for the benchmarks of the software competition SV-COMP. We use variable roles, loop patterns and control flow features to devise source code metrics.[1] Using these metrics, we build a portfolio solver for software verification (see Chapter 4).

      As we mention in Section 6.4, a similar approach of a machine-learning based portfolio solver which employs code metrics to choose a verification tool was formulated and implemented in [TKK+14]. Nonetheless, our method is different from [TKK+14] in that our algorithm employs semantic, rather than structural metrics. Next, our work is re-producible since we formally define our metrics and our weighting function which we use to handle data imbalances. In addition, our work includes a more extensive evaluation: our portfolio is based on 35 verification tools vs. 3 in [TKK+14] and is evaluated on the source based of 56 MLOC vs. 4 MLOC in [TKK+14].

      Our evaluation demonstrates the generality of our approach: devised for the setting of the competition SV-COMP'14, our portfolio solver would become a hypothetical overall winner of SV-COMP'14 as well as SV-COMP'15 and '16 without major changes. We believe that the success of our approach is to a great extend due to our weighting function (see Section 4.2.3), which on one hand formalises the knowledge about the competition setting and scoring policies, and on the other hand is general enough to be applied to settings other than SV-COMP.

   b) Second, we suggest a variable role-based method for the specification of heuristics for choosing program-specific abstraction (see Chapter 5). To the best of our knowledge, our thesis is the first work to formally describe heuristics in software verification (authors of verification tools either describe their heuristics informally [NR10, LRS16, CCF+09] or do not describe them at all).

      As a case study, we use variable roles to formulate heuristics for the model checker Eldarica. We identify 5 new variable roles important for Eldarica on a set of SV-COMP verification benchmarks. We implement existing built-in heuristics of Eldarica as well as 5 new heuristics derived from our set of benchmarks.

      Finally, we evaluate Eldarica extended with role-based heuristics on an extensive set of benchmarks from SV-COMP and literature. We stress that we derive our heuristics from a small fraction of SV-COMP benchmarks (appr. 35 benchmarks altogether) and evaluate the extended tool on a much larger set including benchmarks from other sources than SV-COMP (see Section 5.3 for

[1] The loop patterns and control flow features are not the contributions of the author of this thesis.

details). Our evaluation demonstrates that Eldarica extended with role-based heuristics solves 11.2 % more tasks on our set of benchmarks, and shows a significant speedup on certain benchmark families.

7.2 Threats to Validity

In this section we discuss threats to internal and external validity.

Threats to internal validity

Overlap between training and test data. The major threat to *internal validity* for results in the machine learning field, and therefore for the evaluation of our portfolio solver, is a selection bias when there is an overlap between training and test data, leading to falsely pronounced results in the machine learning accuracy on the test data. As discussed in Section 4.2.3, a feature vector in our formulation of a machine learning task consists of program metrics and the type of a verification task, i.e. reachability, memory safety, etc. The threat of overlap between training and test data arises because only 70% of SV-COMP benchmarks in the year 2016 (65% and 77% in the years 2015 and 2014 respectively) have unique metrics, since many unsafe benchmarks are manually derived from their safe counterparts with minor changes, e.g. flipping a comparison operator, etc. We therefore mitigate this threat by partitioning the learning data in such a way that every pair of programs with same metrics occurs either in training, or in test set, but not partitioned to different sets.

Random effects. We use pseudo-random partitioning of the machine learning data to a training and test set, modulo the restriction mentioned in the previous paragraph. To mitigate random effects in the evaluation of our portfolio solver, we perform 10 experiments with different partitionings, and we report the mean results for the 10 experiments (see Section 4.3.3 and Tables 8.1 on page 173, 8.2 on page 175 and 8.3 on page 177).

Non-determinism in constituent verifiers of the portfolio solver. In our algorithm, we assume that all the verifiers used by our portfolio solver are *deterministic*. Non-determinism might happen, for example, if a verification tool employs an SMT solver which uses different random seeds on each run. In case the assumption of determinism is not satisfied, machine learning data might be *inconsistent*, i.e. to a feature vector $\mathbf{x}(v)$ of a verification task v may correspond two (or more) different labels $L_1(v), L_2(v), \ldots$ (we use the notation introduced in Section 4.2.3). For example, a verification tool might give an answer "safe" on a first run and not terminate on a second run. In case inconsistent data is partitioned into training and test set, misclassification (i.e. a high classification error) will occur. We exclude this threat by the way the benchmarks are partitioned. Even more importantly, usually an effort is made by designers of verification tools to make their tools deterministic, in particular to achieve reproducible results.

Uncovering additional information about benchmarks to the portfolio solver. We stress that our portfolio solver receives same input as the participants of the competition, e.g. the test data does not include the *competition category* of a verification task, neither the *anticipated answer*, i.e. safe/unsafe.

Errors in labelling benchmarks as safe/unsafe. The next threat to internal validity are possible errors in labelling verification tasks as safe or unsafe. In our evaluation of both the portfolio solver and heuristics for Eldarica (Sections 4.3.3 and 5.3 respectively), we assumed that the labelling made by the designers of the benchmarks is correct. Though, in the evaluation of the heuristics for Eldarica in Section 5.3, we found two errors in labelling benchmarks from the set Llreve, when safe benchmarks were labelled by the designers as unsafe. We reported our results for the corrected labelling.

Neglecting time overhead. As explained in Section 4.2, our portfolio solver performs additional computations such as program feature extraction and prediction of the optimal verification tool for a task. We report in Section 4.3.3 the median overhead time, which amounts to $\tilde{x}_{\text{features}} = 0.5$ seconds for feature extraction and $\tilde{x}_{\text{prediction}} = 0.5$ seconds for prediction. We find this overhead to be negligible, when compared to median verification time, the median verification time of the winner of SV-COMP'16 UltimateAutomizer is 24.9 seconds.

Nonuniform time measurement. Recall that for the evaluation of our portfolio solver we did not re-run the SV-COMP competition, but used the results published on the competition website. Since we executed our algorithm for the feature extraction and the prediction of the optimal tool on a different machine than the one on which the competition was executed, the overhead time is not directly comparable to the run-times of the tools, though the overhead time can be used as an estimation.

Threats to external validity

Overfitting of portfolio solver. Threats to *external validity* of our portfolio solver evaluation can arise because these results may not generalise to other types of benchmarks or other verification tools. In the machine learning field this problem is called *overfitting*.

Indeed, we did not evaluate our setting on benchmarks other than the SV-COMP benchmarks, because for this evaluation we would need to have a sufficient for machine learning number of benchmarks labelled as safe/unsafe, and it would be difficult to get these benchmarks outside of SV-COMP. Next, we would need to run all constituent verification tools of the portfolio solver on the new set of benchmarks (recall again, that for our experiments, we did not run the tools ourselves, but used the results of the competition). We believe, though, that this additional effort is time consuming and goes outside of the scope of our thesis. On top of that, we note that the set of the benchmarks of the SV-COMP competition is quite manifold: first, the set contains different types of programs, such as Linux device drivers as well

as verification tasks submitted by competition participants, including concurrent, recursive and other types of programs; second, the SV-COMP verification tasks include verification properties of different types such as reachability, memory safety, overflow and termination.

Finally, we note that in our experiments we evaluated the portfolio solver in the setting of three editions of SV-COMP in the years 2014, 2015 and 2016 (see Section 4.3.3), with a number of participating tools added, removed or updated between the editions. Taking into account all the above arguments, we hope that our portfolio solver will generalise to previously unseen benchmarks and verification tools.

Non-generalisation of heuristics. A similar threat of external validity arises for our evaluation of Eldarica extended with heuristics, i.e. the extended tool may show performance regression on other benchmarks than those from which the heuristics were derived. As mentioned in Section 3.1.3, we designed our heuristics based on appr. 30 benchmarks which could not be solved by Eldarica within the time limit of 15 minutes. In particular, we analysed the SV-COMP'16 benchmarks from categories "Integers and Control Flow" (SV-COMP CFI) and "Loops" (SV-COMP Loops) and loop invariant generation benchmarks (HOLA). The analysed benchmarks constitute 5% of the set of 549 C benchmarks which we used to evaluate Eldarica enhanced with heuristics, see Table 5.1 on page 145 for details.

In our experiments the enhanced tool shows improvement in the number of solved tasks on the sets of benchmarks SV-COMP CFI, SV-COMP Loop and HOLA . On the remaining benchmark sets the unmodified version of Eldarica with built-in heuristics Eld+B (see Table 5.2 on page 146 for the description of used configurations of Eldarica) already solves 98% of the tasks, and the enhanced tool does not show the improvement on these sets of benchmarks. However, on the overall set of benchmarks, the enhanced tool does not show performance regression, as reported in Section 5.3. These observations give us hope that our algorithm and implementation can be successfully applied to other benchmarks.

7.3 Future Work

As future work, we believe that variable roles are interesting to study both on their own and in application to software verification techniques.

First, it would be interesting to incorporate into our notion of variable roles the information from comments in source code and from names of variables. To this end, one can use the methods of natural language processing and the existing work in this direction in the program understanding field [DP06, LFB07].

Second, the framework for defining variable roles can be enhanced by incorporating flow-sensitivity into roles. With this approach, variable roles would be assigned to a variable per control location or per block of code – to compare, in our thesis a single role

is assigned to a variable at all control locations, with the role summarising the usage patterns of the variable across the whole program. Flow-sensitivity would provide more accurate information about usage patterns of variables. In addition, flow-sensitive roles could be used to detect possible bugs when a role of a variable changes; for instance, this approach is implemented in the Coverity bug finding tool [ECH+01].

Next, it would be interesting to evaluate our portfolio solver in a new setting, where the portfolio solver is trained on the benchmarks of the software verification competition (or some other representative set of benchmarks) and is tested on verification tasks *not included* in the competition benchmarks. We stress that in the experiments which we present in this thesis, we use disjoint sets of benchmarks for training and testing of our portfolio solver. Nonetheless it would be interesting to explore the effectiveness of our approach applied to other verification tasks.

Finally, the implementation of the algorithm which we present in this thesis can be enhanced in terms of efficiency. In particular, the time of evaluating a logic program can be reduced by optimising Datalog queries [BS09]; an orthogonal enhancement increasing the scalability of the approach would be to store the translated program (for which the roles are inferred) in a relational database and to translate Datalog queries to the SQL language [HVdM06, MLL05].

CHAPTER 8

Appendices

8.A Definitions of Supplementary Relations for Variable Roles

```
1 % Variable X is assigned expression Expr
2 assigned(X,Expr):- assigment_stmt(Stmt), lhs_expr(Stmt,X)
3   rhs_expr(Stmt,Expr).
4
5 % Transitive closure of the relation assigned
6 assigned_tc(X,Expr):- assigned(X,Expr).
7 assigned_tc(X,Expr):- assigned(X,Z), assigned_tc(Z,Expr).
8
9 % Expression SubExpr is an operand of the operator Expr
10 operand(Expr,SubExpr):- lhs_expr(Expr,SubExpr).
11 operand(Expr,SubExpr):- rhs_expr(Expr,SubExpr).
12
13 % function Func is called
14 called(Func,CallExpr):- call_expr(CallExpr), function(CallExpr,Func).
15
16 % Expression Arg is I-th actual parameter of a function/macro call
17 act_arg(Func,I,Arg):- call_expr(CallExpr), function(CallExpr,Func),
18   param(CallExpr,I,Arg).
19
20 act_arg(Macro,I,Arg):- macro_call_expr(MacroCallExpr),
21   macro(MacroCallExpr,Macro), param(MacroCallExpr,I,Arg).
22
23 % Variable X is assigned a call to function Func
24 assigned_call(X,Func):- assigned(X,CallExpr), call_expr(CallExpr),
25   function(CallExpr,Func).
```

Figure 8.1 Specification of supplementary relations for defining variable roles.

```
% Expression Lit is a literal of the boolean expression Expr
literal(Expr,Lit):- bool_subexpr(Expr,Lit), not comp_bool_expr(Lit).

% Expression SubExpr is a boolean-valued subexpression of expression Expr
bool_subexpr(Expr,SubExpr):- bool_subexpr(Expr,Expr1), bop_expr(Expr1),
  opcode(Expr1,Opcode), logical_opcode(Opcode), operand(Expr1,SubExpr).

bool_subexpr(Expr,SubExpr):- bool_subexpr(Expr,Expr1), uop_expr(Expr1),
  opcode(Expr1,Opcode), logical_opcode(Opcode), operand(Expr1,SubExpr).

bool_subexpr(Expr,Expr).

% Compound boolean expression -- logical AND, OR or NOT.
comp_bool_expr(Expr):- bop_expr(Expr), opcode(Expr,Opcode),
  logical_opcode(Opcode).

comp_bool_expr(Expr):- uop_expr(Expr), opcode(Expr,Opcode),
  logical_opcode(Opcode).

% Expression Atom is an atom of Expr
atom(Expr,Atom):- arithm_subexpr(Expr,Atom), not bop_expr(Atom),
  not uop_expr(Atom).

% Expression SubExpr is an integer-valued subexpression of expression Expr
arithm_subexpr(Expr,SubExpr):- arithm_subexpr(Expr,Expr1), bop_expr(Expr1),
  opcode(Expr1,Opcode), arithm_opcode(Opcode), operand(Expr1,SubExpr).

arithm_subexpr(Expr,SubExpr):- arithm_subexpr(Expr,Expr1), uop_expr(Expr1),
  opcode(Expr1,Opcode), arithm_opcode(Opcode), operand(Expr1,SubExpr).

arithm_subexpr(Expr,Expr).

% Statement SubStmt is a sub-statement of statement Stmt
sub_stmt(Stmt,SubStmt) :- sub_stmt(Stmt,SeqStmt), sequence_stmt(SeqStmt),
  sub_stmt1(SeqStmt,SubStmt).

sub_stmt(Stmt,SubStmt) :- sub_stmt(Stmt,SeqStmt), sequence_stmt(SeqStmt),
  sub_stmt2(SeqStmt,SubStmt).

sub_stmt(Stmt,SubStmt) :- sub_stmt(Stmt,WhileStmt), while_stmt(WhileStmt),
  body(WhileStmt,SubStmt).

sub_stmt(Stmt,SubStmt) :- sub_stmt(Stmt,IfStmt), if_stmt(IfStmt),
  then_stmt(IfStmt,SubStmt).

sub_stmt(Stmt,SubStmt) :- sub_stmt(Stmt,IfStmt), if_stmt(IfStmt),
  else_stmt(IfStmt,SubStmt).

sub_stmt(Stmt,Stmt).
```

Figure 8.2 (Cont.) Specification of supplementary relations for defining variable roles.

```
% Str is the string representation of the expression Expr
expr_str(Expr,Str):- bop_expr(Expr), opcode(Expr,Opcode),
  lhs_expr(Expr,LhsExpr), rhs_expr(Expr,RhsExpr),
  expr_str(LhsExpr,LhsExprStr), expr_str(RhsExpr,RhsExprStr),
  Str=@concat("(",LhsExprStr,Opcode,RhsExprStr,")").

expr_str(Expr,Str):- uop_expr(Expr), opcode(Expr,Opcode),
  sub_expr(Expr,SubExpr), expr_str(SubExpr,SubExprStr),
  Str=@concat("(",Opcode,SubExprStr,")").

expr_str(Expr,Str):- var(Expr), name(Expr,Str).
expr_str(Expr,Str):- const_literal(Expr), val(Expr,Str).

% Func is an external function
ext_func(Func):- func_decl(Func), not has_body(Func).
has_body(Func):- func_decl(Func), body(Func,Body).

% Relations which encode function names for roles
% Standard library functions which manipulate characters
char_use_func("fputc",0).
char_use_func("putc",0).
char_use_func("putchar",0).
char_use_func("isalnum",0).
char_use_func("isalpha",0).
char_use_func("isascii",0).
char_use_func("isblank",0).
char_use_func("iscntrl",0).
char_use_func("isdigit",0).
char_use_func("islower",0).
char_use_func("isprint",0).
char_use_func("ispunct",0).
char_use_func("isspace",0).
char_use_func("isupper",0).
char_use_func("isxdigit",0).

% Standard library functions which return a character
char_def_func("fgetc",fres).
char_def_func("getc",fres).
char_def_func("fgetwc",fres).
char_def_func("getwc",fres).
char_def_func("getchar",fres).
char_def_func("getwchar",fres).
```

Figure 8.3 (Cont.) Specification of supplementary relations for defining variable roles.

```
117 % Standard library functions which manipulate file descriptors
118 file_use_func("read",0).
119 file_use_func("write",0).
120 file_use_func("fstat",0).
121 file_use_func("openat",0).
122 file_use_func("fcntl",0).
123 file_use_func("fchmod",0).
124 file_use_func("fchown",0).
125 file_use_func("dup",0).
126 file_use_func("dup2",0).
127 file_use_func("dup3",0).
128 file_use_func("lseek",0).
129 file_use_func("mmap",0).
130
131 % Standard library functions which return a file descriptor
132 file_def_func("open",fres).
133 file_def_func("openat",fres).
134 file_def_func("creat",fres).
135 file_def_func("dup",fres).
136 file_def_func("dup2",fres).
137 file_def_func("dup3",fres).
138 file_def_func("fcntl",fres).
139
140 % Standard library functions which manipulate thread descriptors
141 thread_descr_use_func("pthread_cancel",0).
142 thread_descr_use_func("pthread_detach",0).
143 thread_descr_use_func("pthread_equal",0).
144 thread_descr_use_func("pthread_equal",1).
145 thread_descr_use_func("pthread_getschedparam",0).
146 thread_descr_use_func("pthread_join",0).
147 thread_descr_use_func("pthread_setschedparam",0).
148
149 % Standard library functions which return a thread descriptor
150 thread_descr_def_func("pthread_create",0).
151 thread_descr_def_func("pthread_self",fres).
152
153 % Standard library functions which manipulate
154 %  dynamically allocated memory
155 dyn_mem_use_func("free",0).
156 dyn_mem_use_func("cfree",0).
157 dyn_mem_use_func("malloc_usable_use",0).
158 dyn_mem_use_func("munmap",0).
159 dyn_mem_use_func("mprotect",0).
160 dyn_mem_use_func("msync",0).
161 dyn_mem_use_func("madvise",0).
162 dyn_mem_use_func("mlock",0).
163 dyn_mem_use_func("munlock",0).
164 dyn_mem_use_func("mincore",0).
```

Figure 8.4 (Cont.) Specification of supplementary relations for defining variable roles.

```
165 % Standard library functions which return the address
166 %  of dynamically allocated memory
167 dyn_mem_def_func("malloc",fres).
168 dyn_mem_def_func("calloc",fres).
169 dyn_mem_def_func("realloc",fres).
170 dyn_mem_def_func("memalign",fres).
171 dyn_mem_def_func("valloc",fres).
172 dyn_mem_def_func("pvalloc",fres).
173 dyn_mem_def_func("mmap",fres).
174 dyn_mem_def_func("mremap",fres).
175
176 % Standard library functions which allocate memory dynamically
177 % or manipulate dynamically allocated memory
178 % and take as a parameter the size of the memory allocation
179 dyn_mem_size_func("malloc",0).
180 dyn_mem_size_func("calloc",1).
181 dyn_mem_size_func("realloc",1).
182 dyn_mem_size_func("memalign",1).
183 dyn_mem_size_func("valloc",0).
184 dyn_mem_size_func("pvalloc",0).
185 dyn_mem_size_func("mmap",1).
186 dyn_mem_size_func("mremap",2).
187
188 % Assertion functions
189 assert_func("assert",0).
190
191 %
192 % OPERATION CODES
193 %
194 % Bitwise operators
195 bit_opcode("BIT_AND").
196 bit_opcode("BIT_OR").
197 bit_opcode("BIT_XOR").
198 bit_opcode("BIT_NOT").
199
200 % Comparison operators
201 compar_opcode(Opcode):- rel_opcode(Opcode).
202 compar_opcode(Opcode):- eq_opcode(Opcode).
203
204 % Relational operators
205 rel_opcode("<").
206 rel_opcode("<=").
207 rel_opcode(">").
208 rel_opcode(">=").
209
210 % Strict relational operators
211 strict_rel_opcode("<").
212 strict_rel_opcode(">").
213
214 % Equality operators
215 eq_opcode("==").
216 eq_opcode("!=").
```

Figure 8.5 (Cont.) Specification of supplementary relations for defining variable roles.

```
217 % Arithmetic operators
218 arithm_opcode(Opcode):- rel_opcode(Opcode).
219 arithm_opcode(Opcode):- add_opcode(Opcode).
220 arithm_opcode(Opcode):- uadd_opcode(Opcode).
221 arithm_opcode("*").
222 arithm_opcode("/").
223
224 % Addition and subtraction operators
225 add_opcode("+").
226 add_opcode("-").
227
228 % Unary plus and minus operators
229 uadd_opcode("+").
230 uadd_opcode("-").
231
232 % Logical operators
233 logical_opcode("LOR").
234 logical_opcode("LAND").
235 logical_opcode("LNOT").
236
237 % Boolean-valued operators
238 bool_res_opcode(Opcode):- logical_opcode(Opcode).
239 bool_res_opcode(Opcode):- rel_opcode(Opcode).
240 bool_res_opcode(Opcode):- eq_opcode(Opcode).
241
242 % Codes of supported binary operators
243 bin_opcode("+").
244 bin_opcode("-").
245 bin_opcode("*").
246 bin_opcode("/").
247 bin_opcode("REM"). % remainder operator
248 bin_opcode("PTR_PLUS_INT"). % pointer plus integer
249 bin_opcode("PTR_MINUS_INT"). % pointer minus integer
250 bin_opcode(">").
251 bin_opcode(">=").
252 bin_opcode("<").
253 bin_opcode("<=").
254 bin_opcode("==").
255 bin_opcode("!=").
256 bin_opcode("BIT_SHL"). % left bit shift
257 bin_opcode("BIT_SHR"). % right bit shift
258 bin_opcode("BIT_AND"). % bit and
259 bin_opcode("BIT_OR"). % bit or
260 bin_opcode("BIT_XOR"). % bit xor
261 bin_opcode("LAND"). % logical and
262 bin_opcode("LOR"). % logical or
```

Figure 8.6 (Cont.) Specification of supplementary relations for defining variable roles.

```
263 % Codes of supported unary operators
264 un_opcode("+"). % unary plus
265 un_opcode("-"). % unary minus
266 un_opcode("ADDR_OF"). % address-of
267 un_opcode("DEREF"). % pointer dereference
268 un_opcode("BIT_NOT"). % bit not
269 un_opcode("LNOT"). % logical not
270
271 %
272 % DATA TYPES
273 %
274 % Integral type
275 integral_type(int).
276 integral_type(char).
277
278 % Floating-point type
279 floating_point_type(float).
280
281 % Scalar type
282 scalar_type(Type):- integral_type(Type).
283 scalar_type(Type):- floating_point_type(Type).
```

Figure 8.7 (Cont.) Specification of supplementary relations for defining variable roles.

8.B Algorithm for Solving a System of Horn Clauses

Here we describe an algorithm from [RHK13] which we use in Step 2 of the algorithm on page 139, Section 5.1.3.

We assume the following:

- A mapping $\Pi : \mathcal{R} \to \mathcal{P}(P)$ from relation symbols $Inv_i \in \mathcal{R}$ to finite sets $P_i \subseteq P$ of predicates P.
- For each relation symbol $Inv_i \in \mathcal{R}$ a vector $\bar{x}_i$ of *formal* argument variables has been fixed.

The algorithm constructs an *abstract reachability graph* (ARG), which is a hyper-graph (S, E), where

- $S \subseteq \{(Inv_i, Q) \mid Inv_i \in \mathcal{R}, Q \subseteq \Pi(Inv_i)\}$ is the set of nodes.
  Each node is a pair consisting of a relation symbol Inv_i and a set of predicates Q.
- $E \subseteq S^* \times HC \times S$ is a hyper-edge relation.
  An edge $e \in E$
  $$e = (\langle s_1, \ldots, s_n \rangle, h, s),$$
  labelled with a Horn clause $h \in HC$,
  $$h = (B_1 \wedge \cdots \wedge B_n \wedge \varphi \to H),$$
  implies that
  - $s_i = (Inv_i, Q_i)$, and $B_i = Inv_i(\bar{t}_i)$ for all $i = 1, \ldots, n$.
    Note that $\bar{t}_i$ are vectors of terms passed as *actual* arguments to the relation symbols Inv_i in the clause h;
  - $s = (Inv, Q)$, $H = Inv(\bar{t})$,
    where again $\bar{t}$ is a vector of argument terms passed as *actual* arguments to the relation symbol Inv in the clause h; and Q is calculated as
    $$Q = \{\psi \in \Pi(Inv) \mid Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \wedge \varphi \models \psi[\bar{t}]\}.$$
    Here, we write $Q_i[\bar{t}_i]$ for the conjunction of the predicates Q_i, with the formal arguments $\bar{x}_i$ replaced by the argument terms $\bar{t}_i$.

An ARG (S, E) is called *closed* if the edge relation E represents all Horn clauses in HC: For every clause $h \in HC$,
$$h = (Inv_1(\bar{t}_1) \wedge \cdots \wedge Inv_n(\bar{t}_n) \wedge \varphi \to H)$$
and every sequence of nodes
$$(Inv_1, Q_1), \ldots, (Inv_n, Q_n) \in S,$$
one of the following holds:

- $Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \wedge \varphi \models \mathit{false}$, or
- there is an edge $E(\langle (\mathit{Inv}_1, Q_1), \ldots, (\mathit{Inv}_n, Q_n)\rangle, h, (\mathit{Inv}, Q))$ such that
  - $H = \mathit{Inv}(\bar{t})$, and
  - $Q = \{\psi \in \Pi(\mathit{Inv}) \mid \varphi \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \psi[\bar{t}]\}$.

If (and only if) the solution can be expressed symbolically using DNF formulae from predicates P, then the set HC of Horn clauses has a closed ARG (see [RHK13] for a proof).

A closed ARG is created using a fixed-point procedure, starting from the empty sets of nodes and edges: $S = \emptyset, E = \emptyset$ (for details see [RHK13]).

Given a closed ARG (S, E) for Horn clauses HC, a solution $\mathit{Sol} : HC \to \mathcal{P}(P)$ for HC is extracted from (S, E) as follows:

$$\mathit{Sol}(\mathit{Inv}_i) = \bigvee_{\{s \in S \mid s = (\mathit{Inv}_i, Q_i)\}} \Big(\bigwedge_{q \in Q_i} q \Big).$$

8.C Definitions of Loop Patterns

In this section we define the loop patterns which we introduce in Section 4.1.2.

1. A loop is a *FOR* loop if the following conditions hold:
   a) The loop iterator i is monotonically incremented by a constant number;
   b) The loop condition is a predicate of the form $i \circ \mathit{expr}$, where $\circ \in \{\geq, \leq, >, <, ==, !=\}$ and *expr* is an expression;
   c) The expression *expr* is a loop invariant;
   d) The loop condition is evaluated at every loop iteration;
   e) The loop condition eventually evaluates to *true*.
2. A loop is syntactically bounded if the loop itself and all the nesting loops are *FOR* loops;
3. A loop is a *generalized FOR* loop if all of the conditions for the *FOR* loop hold except for one of the following conditions:
   - 1e) and 1a);
   - 1c);
   - 1d).
4. A loop is a *hard* loop if it is not *generalized FOR* loop.

For a formal definition using data-flow analysis and a detailed discussion see [PVZ15].

8.D Experimental Results for Portfolio Solver

In Tables 8.1–8.3 we show the detailed results of the evaluation of our portfolio solver $\mathcal{TP}$ and idealised strategies T_{cat} and T_{vbs}.

continued on next page...

Category	blast	cbmc	cpa-checker	cpa-lien	cseq-lazy	cseq-mu	esbmc	fbit	llbmc
BitVectors	-	**33** **15**	30 28	-	-	-	**30** **9**	-	**33** **0**
Concurrency	-	49 187	0 1z	-	**53** **6**	**53** **9**	20 209	-	0 0
ControlFlow	202 1763	218 694	**418** **393**	164 523	-	-	396 614	**406** **512**	405 348
DeviceDrivers64	**1084** **150**	987 3201	1055 455	-	-	-	941 2013	**1066** **293**	0 1
HeapManipulation	-	**52** **114**	39 69	25 127	-	-	37 69	-	39 99
MemorySafety	-	-2 94	**38** **6**	7 127	-	-	-24 117	-	**15** **159**
Recursive	-	**13** **78**	0 0	-	-	-	-21 96	-	1 0
Sequentialized	-	**91** **512**	44 910	-	-	-	**98** **689**	-	84 371
Overall	468 2066	**1292** **4991**	**1235** **1865**	266 776	183 6	183 9	695 4024	666 898	853 978
Medals	1/0/0	2/2/2	2/1/1	0/0/0	1/0/0	0/1/0	1/0/1	0/0/2	1/0/1

Table 8.1 Experimental results for the competition participants, plus our portfolio $\mathcal{TP}$ on *random subsets* of SV-COMP'14, given as arithmetic mean of 10 experiments on the respective test sets $test_{year}$. The two last columns show the idealized strategies T_{cat}, T_{vbs} (not competing, for comparison only). The first rows show the respective SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The last row shows the number of gold/silver/bronze medals won in individual categories.

. . . continued from previous page

predator	symbiotic	threader	ufo	ultimate-Automizer	ultimate-Kojak	$\mathcal{TP}$	T_{cat}	T_{vbs}
-34	15	-	-	4	-8	30	33	33
1	114			42	110	9	0	0
0	-30	40	-	0	0	**52**	53	53
0	204	53		1	1	**26**	6	1
183	29	-	373	66	70	**409**	469	503
792	3046		176	563	694	**278**	170	46
21	384	-	**1067**	0	0	1036	1084	1111
83	3848		**163**	24	24	1276	150	42
43	38	-	-	5	5	**50**	52	53
11	150			1	1	**64**	114	0
6	-61	-	-	0	0	**15**	38	38
27	0			1	1	**63**	6	0
-7	2	-	-	**6**	5	**10**	13	15
5	117			**72**	78	**77**	78	57
-19	-13	-	38	20	4	**96**	98	132
622	410		41	111	1061	**389**	689	122
44	-97	137	735	193	94	**1494**	1732	1840
1541	7891	53	381	816	1973	**2211**	1310	270
0/0/1	0/0/0	0/0/0	1/1/0	0/0/1	0/0/0	1/5/1	-	-

Table 8.1 Experimental results for the competition participants, plus our portfolio $\mathcal{TP}$ on *random subsets* of SV-COMP'14, given as arithmetic mean of 10 experiments on the respective test sets $test_{year}$. The two last columns show the idealized strategies T_{cat}, T_{vbs} (not competing, for comparison only). The first rows show the respective SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The last row shows the number of gold/silver/bronze medals won in individual categories.

continued on next page...

Category	aprove	beagle	blast	cascade	cbmc	cpachecker	cparec	esbmc	forest	forester	function	hiptnt
Arrays	-	-	-	-	-85 20	1 87	-	-123 0	-	-	-	-
BitVectors	-	-0 81	-	25 128	**28** **8**	**26** **30**	-	**29** **2**	-	-	-	-
Concurrency	-	-	-	-	402 600	0 17	-	402 290	-	-	-	-
ControlFlow	-	-	436 4091	384 4243	78 6436	**999** **3348**	-	**817** **2664**	247 0	0 18	-	-
Device-Drivers64	-	-	**1092** **329**	-	906 3881	1027 1468	-	894 2354	-	-	-	-
Floats	-	-	-	-	**56** **136**	**34** **160**	-	4 104	-	-	-	-
Heap-Manipulation	-	-	-	31 101	39 98	**40** **62**	-	30 57	0 0	13 0	-	-
Memory-Safety	-	-	-	76 673	-167 117	**131** **40**	-	-306 114	0 3	8 0	-	-
Recursive	-	2 33	-	-	**8** **79**	6 82	7 59	-18 20	-	-	-	-
Sequentia-lized	-	-	-	-	-71 425	**58** **857**	-	**98** **396**	-	-	-	-
Simple	-	-	15 126	-	23 135	**25** **136**	-	14 20	-	-	-	-
Termination	**245** **283**	-	-	-	0 0	0 1	-	-343 11	-	-	144 13	**219** **24**
Overall	241 283	36 114	737 4546	806 5146	684 11936	**2228** **6288**	121 59	-38 6032	194 3	84 18	142 13	215 24
Medals	1/0/0	0/0/0	1/0/0	0/0/0	1/1/1	2/1/5	0/0/0	2/0/1	0/0/0	0/0/0	0/0/0	0/0/1

Table 8.2 Experimental results for the competition participants, plus our portfolio $\mathcal{TP}$ on *random subsets* of SV-COMP'15, given as arithmetic mean of 10 experiments on the respective test sets $test_{year}$. The two last columns show the idealized strategies T_{cat}, T_{vbs} (not competing, for comparison only). The first rows show the respective SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The last row shows the number of gold/silver/bronze medals won in individual categories.

. . . continued from previous page

lazycseq	map2check	mucseq	perentie	predatorhp	seahorn	smack	ultimate-automizer	ultimateko-jak	ulcseq	$\mathcal{TP}$	T_{cat}	T_{vbs}
-	-	-	-	-	**2** **652**	**46** **14**	-	2 666	2 666	**34** **16**	46 14	75 1
-	-	-	-	-	-37 41	-	-	0 10	-28 49	24 16	29 2	34 2
480 **37**	-	**480** **106**	-	-	-3554 1	-	380 994	0 34	0 35	**477** **67**	480 37	480 6
-	-	-	180 75	-	**910** **3622**	713 6588	-	799 5129	381 9694	661 4348	1076 3074	1243 1537
-	-	-	-	-	**1058** **816**	1000 2049	-	105 67	30 72	**1068** **800**	1092 329	1191 134
-	-	-	-	-	-76 0	-	-	-19 3	-18 3	**54** **132**	56 136	57 73
-	-	-	-	**42** **9**	-16 0	**41** **12**	-	31 58	31 87	39 13	42 9	52 0
-	23 50	-	-	**93** **87**	0 0	-	-	39 586	27 580	**115** **247**	131 40	146 1
-	-	-	-	-	-38 0	**14** **16**	-	**10** **38**	4 103	3 53	14 16	16 7
-	-	-	-	-	5 581	-	-	12 950	2 1093	**83** **408**	98 396	130 109
-	-	-	-	-	**30** **25**	25 49	-	-0 160	3 182	**29** **37**	30 25	32 3
-	-	-	-	-	0 0	-	-	**233** **276**	0 0	205 124	245 283	295 10
190 37	40 50	190 106	142 75	389 96	-1534 5740	**1542** **8727**	150 994	1215 7979	273 12563	**2511** **6260**	3231 4360	3768 1882
1/0/0	0/0/0	0/1/0	0/0/0	1/0/1	1/1/2	2/1/1	0/0/0	0/2/0	0/0/0	1/6/1	-	-

Table 8.2 Experimental results for the competition participants, plus our portfolio $\mathcal{TP}$ on *random subsets* of SV-COMP'15, given as arithmetic mean of 10 experiments on the respective test sets $test_{year}$. The two last columns show the idealized strategies T_{cat}, T_{vbs} (not competing, for comparison only). The first rows show the respective SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The last row shows the number of gold/silver/bronze medals won in individual categories.

continued on next page...

Category	2ls	aprove	blast	cascade	cbmc	ceagle	civl	cpa-kind	cpa-refsel	cpa-seq
Arrays	-173 227	-	-	22 228	32 322	-	-	2 241	-35 153	-38 260
BitVectors	-242 97	-	-	-	18 86	-	-	**30** **90**	13 113	**34** **53**
Heap	-623 62	-	-	86 413	62 256	-	-	67 73	63 61	**100** **279**
Floats	**64** **38**	-	-	-	**64** **169**	**63** **38**	-	41 164	19 101	38 165
Integers-ControlFlow	541 6899	-	-633 5025	-	-515 7091	-	-	**850** **6427**	626 6708	**1058** **4914**
Termination	-1336 3	**360** **619**	-	-	-	-	-	0 2	0 2	0 2
Concurrency	-9789 9	-	-	-	361 634	-	**513** **188**	0 34	0 32	127 2642
DeviceDrivers-Linux64	768 916	-	1037 888	-	747 8329	-	-	887 5558	**1216** **3070**	**1067** **4196**
Overall	-15055 8251	484 619	202 5912	394 642	1577 16887	549 38	415 188	1678 12587	1151 10240	**1907** **12509**
Medals	1/0/0	1/0/0	0/0/0	0/0/0	0/1/0	0/0/1	0/0/1	0/1/1	1/0/0	2/1/2

Table 8.3 Experimental results for the competition participants, plus our portfolio $\mathcal{TP}$ on *random subsets* of SV-COMP'16, given as arithmetic mean of 10 experiments on the respective test sets $test_{year}$. The two last columns show the idealized strategies T_{cat}, T_{vbs} (not competing, for comparison only). The first rows show the respective SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The last row shows the number of gold/silver/bronze medals won in individual categories. For readability, we omit tools that did not win any medals in the original competition.

. . . continued from previous page

esbmc	lazycseq	mucseq	predatorhp	seahorn	smack	symbiotic3	uautomizer	$\mathcal{TP}$	T_{cat}	T_{vbs}
82	-	-	-	-122	**62**	47	36	**89**	119	136
305				424	**114**	205	763	**342**	248	51
32	-	-	-	-56	17	1	26	29	33	36
55				6	96	151	81	72	22	0
69	-	-	**118**	-102	65	45	79	**104**	129	153
211			**94**	0	49	49	172	**85**	194	7
-4	-	-	-	-	-	-4	1	61	64	66
147						14	6	42	38	5
515	-	-	-	678	787	263	743	**806**	1181	1428
3242				5629	9027	10982	7163	**5202**	4668	2041
-	-	-	-	199	0	-	**351**	**344**	360	416
				551	5		**470**	**430**	619	63
305	**513**	**513**	-	-9827	422	0	0	492	513	513
1490	**73**	**23**		2	668	1	57	206	23	8
622	-	-	-	640	829	369	1044	**1059**	1216	1394
2946				2590	4260	6653	2499	**2164**	3070	372
1699	415	415	411	-8865	1684	580	**1965**	**3269**	3800	4238
8396	73	23	94	9202	14218	18056	**11210**	**8544**	8883	2547
0/2/0	0/1/0	1/0/0	1/0/0	0/0/0	0/0/1	0/0/0	0/2/0	2/1/3	-	-

Table 8.3 Experimental results for the competition participants, plus our portfolio $\mathcal{TP}$ on *random subsets* of SV-COMP'16, given as arithmetic mean of 10 experiments on the respective test sets $test_{year}$. The two last columns show the idealized strategies T_{cat}, T_{vbs} (not competing, for comparison only). The first rows show the respective SV-COMP score and beneath it the runtime in minutes. We highlight the gold, silver, and bronze medal in dark gray, light gray and white+bold font, respectively. The last row shows the number of gold/silver/bronze medals won in individual categories. For readability, we omit tools that did not win any medals in the original competition.

List of Figures

List of Tables

Bibliography

[ABF+13] Sven Apel, Dirk Beyer, Karlheinz Friedberger, Franco Raimondi, and Alexander von Rhein. Domain types: Abstract-domain selection based on variable usage. *Hardware and Software: Verification and Testing*, 8244(1):262–278, 2013.

[Bad00] Greg J Badros. Javaml: a markup language for java source code. *Computer Networks*, 33(1):159–177, 2000.

[BCC+07] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W O'hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *Computer Aided Verification (CAV)*, pages 178–192. Springer, 2007.

[BCLZ04] Thomas Ball, Byron Cook, Shuvendu K Lahiri, and Lintao Zhang. Zapato: Automatic theorem proving for predicate abstraction refinement. In *International Conference on Computer Aided Verification*, pages 457–461. Springer, 2004.

[Bey14] Dirk Beyer. Status report on software verification - (competition summary SV-COMP 2014). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 373–388. Springer, 2014.

[Bey15] Dirk Beyer. Software verification and verifiable witnesses - (report on SV-COMP 2015). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Computer Science*, pages 401–416. Springer, 2015.

[Bey16] Dirk Beyer. 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)*, volume 9636 of *Lecture Notes in Computer Science*, pages 887–904. Springer, 2016.

[BGV92] Bernhard E. Boser, Isabelle Guyon, and Vladimir Vapnik. A training algorithm for optimal margin classifiers. In *Workshop on Computational learning theory (COLT)*, pages 144–152. ACM, 1992.

[BHT07] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification (CAV)*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007.

[BHvM09] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*, volume 185. IOS press, 2009.

[Bis06] Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

[BJ05] Craig Bishop and Colin G Johnson. Assessing roles of variables by program analysis. In *Conference on Computer Science Education*, TUCS General Publication, pages 131–136. Turku Centre for Computer Science, 2005.

[BLW15] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Refinement selection. In *Model Checking of Software (SPIN)*, pages 20–38. Springer, 2015.

[BM07] Aaron R Bradley and Zohar Manna. *The calculus of computation: decision procedures with applications to verification*. Springer, 2007.

[BR02] Thomas Ball and Sriram K. Rajamani. The SLAM project: debugging system software via static analysis. In *Principles of Programming Languages (POPL)*, pages 1–3. ACM, 2002.

[BS09] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. *ACM SIGPLAN Notices*, 44(10):243–262, 2009.

[C+97] Roger F Crew et al. Astlog: A language for examining abstract syntax trees. In *Conference on Domain-Specific Languages (DSL)*, volume 97, page 18. USENIX, 1997.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages (POPL)*, pages 238–252. ACM, 1977.

[CCF+05] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *Programming Languages and Systems*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the astrée static analyzer. In *Asian Computing Science Conference (ASIAN)*, pages 272–300. Springer, 2006.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design (FMSD)*, 35(3):229–264, 2009.

[CGJ+03] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, 2003.

[CGT89] Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[CKL04] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[CKSY05] Edmund M. Clarke, Daniel Kroening, Natasha Sharygina, and Karen Yorav. SATABS: sat-based predicate abstraction for ANSI-C. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005.

[CL11] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[CMM05] Brian Chin, Shane Markstrum, and Todd D. Millstein. Semantic type qualifiers. In *Programming language design and implementation (PLDI)*, volume 40, pages 85–95. ACM, 2005.

[CPR11] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Proving program termination. *Communications of the ACM*, 54(5):88–98, 2011.

[Cra57] William Craig. Linear reasoning. a new form of the herbrand-gentzen theorem. *Journal of Symbolic Logic*, 22(3):250–268, 1957.

[CV95] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20(3):273–297, 1995.

[CW96] Weidong Chen and David S Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, 1996.

[DDLM13] Isil Dillig, Thomas Dillig, Boyang Li, and Kenneth L. McMillan. Inductive invariant generation via abductive inference. In *Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, volume 48, pages 443–456. ACM, 2013.

[DLW15] Matthias Dangl, Stefan Löwe, and Philipp Wendler. Cpachecker with support for recursive programs and floating-point arithmetic - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9035 of *Lecture Notes in Computer Science*, pages 423–425. Springer, 2015.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DP06] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[DPV13] Kamil Dudka, Petr Peringer, and Tomáš Vojnar. Byte-precise verification of low-level list manipulation. In *Static Analysis Symposium (SAS)*, volume 7935 of *Lecture Notes in Computer Science*, pages 215–237. Springer, 2013.

[DPVZ15] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. Empirical software metrics for benchmarking of verification tools. In *Computer Aided Verification (CAV), Part I*, volume 9206 of *Lecture Notes in Computer Science*, pages 561–579. Springer, 2015.

[DPVZ16] Yulia Demyanova, Thomas Pani, Helmut Veith, and Florian Zuleger. Empirical software metrics for benchmarking of verification tools. In *Software Engineering*, volume 252 of *Lecture Notes in Informatics*, pages 67–68. Gesellschaft für Informatik, 2016.

[DRZ17] Yulia Demyanova, Philipp Rümmer, and Florian Zuleger. Systematic predicate abstraction using variable roles. In *NASA Formal Methods Symposium (NFM)*, volume 10227 of *Lecture Notes in Computer Science*, pages 265–281. Springer, 2017.

[DVZ13] Yulia Demyanova, Helmut Veith, and Florian Zuleger. On the concept of variable roles and its use in software analysis. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 226–230. IEEE, 2013.

[ECCH00] Dawson R. Engler, Benjamin Chelf, Andy Chou, and Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Symposium on Operating System Design & Implementation (OSDI)*, pages 1–16. USENIX, 2000.

[ECH+01] Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Symposium on Operating System Principles (SOSP)*, volume 35, pages 57–72. ACM, 2001.

[FGK+14] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Automated software engineering (ASE)*, pages 349–360. ACM, 2014.

[FO76] Lloyd D Fosdick and Leon J Osterweil. Data flow analysis in software reliability. *ACM Computing Surveys (CSUR)*, 8(3):305–330, 1976.

[Fos02] Jeffrey Scott Foster. *Type qualifiers: lightweight specifications to improve software quality*. PhD thesis, University of California Berkeley, 2002.

[Fre94] Tim Freeman. *Refinement Types for ML*. PhD thesis, Bell Laboratories, 1994.

[GB14] Arie Gurfinkel and Anton Belov. Frankenbit: Bit-precise verification with many bits - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of *Lecture Notes in Computer Science*, pages 408–411. Springer, 2014.

[GKK+11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Torsten Schaub, Marius Thomas Schneider, and Stefan Ziller. A portfolio solver for answer set programming: Preliminary report. In *Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 6645 of *Lecture Notes in Computer Science*, pages 352–357. Springer, 2011.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. *CoRR*, abs/1405.3694, 2014.

[GLPR12] Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *Programming Language Design and Implementation (PLDI)*, pages 405–416. ACM, 2012.

[GP99] Peter Gulutzan and Trudy Pelzer. *SQL-99 complete, really*. CMP Books, 1999.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verification (CAV)*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997.

[GS01] Carla P. Gomes and Bart Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.

[HB12] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[HCL+03] Chih-Wei Hsu, Chih-Chung Chang, Chih-Jen Lin, et al. A practical guide to support vector classification. Technical report, National Taiwan University, 2003.

[HCXE02] Seth Hallem, Benjamin Chelf, Yichen Xie, and Dawson R. Engler. A system and language for building system-specific, static analyses. In *Programming Language Design and Implementation (PLDI)*, pages 69–82. ACM, 2002.

[HD05] Yi-Min Huang and Shu-Xin Du. Weighted support vector machine for classification with uneven training class sizes. In *Machine Learning and Cybernetics*, volume 7, pages 4365–4369, 2005.

[HDG+16] Matthias Heizmann, Daniel Dietsch, Marius Greitschus, Jan Leike, Betim Musa, Claus Schätzle, and Andreas Podelski. Ultimate automizer with two-track proofs - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 950–953. Springer, 2016.

[HG09] Haibo He and Edwardo A Garcia. Learning from imbalanced data. *Knowledge and Data Engineering*, 21(9):1263–1284, 2009.

[Hin01] Michael Hind. Pointer analysis: haven't we solved this problem yet? In *Program Analysis For Software Tools and Engineering (PASTE)*, pages 54–61. ACM, 2001.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003.

[HKG+12] Hossein Hojjat, Filip Konecný, Florent Garnier, Radu Iosif, Viktor Kuncak, and Philipp Rümmer. A verification toolkit for numerical transition systems - tool paper. In *Formal Methods (FM)*, volume 7436 of *Lecture Notes in Computer Science*, pages 247–251. Springer, 2012.

[HLH97] Bernardo A Huberman, Rajan M Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275(5296):51–54, 1997.

[HVdM06] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. *codeQuest:* scalable source code queries with datalog. In *European Conference on Object-oriented Programming (ECOOP)*, volume 4067 of *Lecture Notes in Computer Science*, pages 2–27. Springer, 2006.

[ITF+14] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *Computer Aided Verification (CAV)*, volume 8559 of *Lecture Notes in Computer Science*, pages 585–602. Springer, 2014.

[JM09] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys*, 41(4):21, 2009.

[JV03] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *International Conference on Aspect-oriented software development (AOSD)*, pages 178–187. ACM, 2003.

[KGCC13] Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M Clarke. Automatic abstraction in smt-based unbounded software model checking. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 846–862. Springer, 2013.

[KMS+11] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. Algorithm selection and scheduling. In *Principles and Practice of Constraint Programming (CP)*, volume 6876 of *Lecture Notes in Computer Science*, pages 454–469. Springer, 2011.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures - An Algorithmic Point of View*. Texts in Theoretical Computer Science (EATCS). Springer, 2008.

[LFB07] Dawn Lawrie, Henry Feild, and David Binkley. Extracting meaning from abbreviated identifiers. In *Source Code Analysis and Manipulation (SCAM)*, pages 213–222. IEEE, 2007.

[LL98] Lionel Lobjois and Michel Lemaître. Branch and bound algorithm selection by performance prediction. In *National Conference on Artificial Intelligence and Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI)*, pages 353–358. AAAI Press / The MIT Press, 1998.

[LRS16] Jérôme Leroux, Philipp Rümmer, and Pavle Subotić. Guiding craig interpolation with domain-specific abstractions. *Acta Informatica*, 53(4):387–424, 2016.

[McM05] Kenneth L. McMillan. Applications of craig interpolants in model checking. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[MLL05] Michael C. Martin, V. Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In *Conference on Object-Oriented Programming (OOPSLA)*, volume 40, pages 365–383. ACM, 2005.

[MPR12] Marco Maratea, Luca Pulina, and Francesco Ricca. The multi-engine ASP solver me-asp. In *Logics in Artificial Intelligence (JELIA)*, volume 7519 of *Lecture Notes in Computer Science*, pages 484–487. Springer, 2012.

[MR05] Laurent Mauborgne and Xavier Rival. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*

(ESOP), volume 3444 of *Lecture Notes in Computer Science*, pages 5–20. Springer, 2005.

[NNH99] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis*. Springer, 1999.

[NR10] Aditya V. Nori and Sriram K. Rajamani. An empirical study of optimizations in YOGI. In *International Conference on Software Engineering (ICSE, Volume 1)*, pages 355–364. ACM, 2010.

[OHH+08] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent, and Barry O'Sullivan. Using case-based reasoning in an algorithm portfolio for constraint solving. In *Irish Conference on Artificial Intelligence and Cognitive Science (AICS)*, 2008.

[Pie91] Benjamin C Pierce. *Programming with intersection types and bounded polymorphism*. PhD thesis, Carnegie Mellon University Pittsburgh, 1991.

[PT07] Luca Pulina and Armando Tacchella. A multi-engine solver for quantified boolean formulas. In *Principles and Practice of Constraint Programming (CP)*, volume 4741 of *Lecture Notes in Computer Science*, pages 574–589. Springer, 2007.

[PT09] Luca Pulina and Armando Tacchella. A self-adaptive multi-engine solver for quantified boolean formulas. *Constraints*, 14(1):80–116, 2009.

[PVZ15] Thomas Pani, Helmut Veith, and Florian Zuleger. Loop patterns in C programs. *Electronic Communication of the European Association of Software Science and Technology (ECEASST)*, 72, 2015.

[Rep95] Thomas W Reps. Demand interprocedural program analysis using logic databases. In *Applications of Logic Databases*, pages 163–196. Springer, 1995.

[RHK13] Philipp Rümmer, Hossein Hojjat, and Viktor Kuncak. Disjunctive interpolants for horn-clause verification. In *Computer Aided Verification (CAV)*, volume 8044 of *Lecture Notes in Computer Science*, pages 347–363. Springer, 2013.

[Ric76] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.

[RKJ08] Patrick Maxim Rondon, Ming Kawaguchi, and Ranjit Jhala. Liquid types. In *Programming Language Design and Implementation (PLDI)*, volume 43, pages 159–169. ACM, 2008.

[RM07] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(5):26, 2007.

[Rou12] Olivier Roussel. Description of ppfolio 2012. In *SAT Challenge*, page 46, 2012.

[Saj02] Jorma Sajaniemi. An empirical analysis of roles of variables in novice-level procedural programs. In *International Symposium on Human-Centric Computing Languages and Environments (HCC)*, pages 37–39. IEEE, 2002.

[SK03] Jorma Sajaniemi and Marja Kuittinen. Program animation based on the roles of variables. In *ACM symposium on Software visualization*, pages 7–16, 205. ACM, 2003.

[SK04] Jorma Sajaniemi and Marja Kuittinen. Visualizing roles of variables in program animation. *Information Visualization*, 3(3):137–153, 2004.

[SK05] Jorma Sajaniemi and Marja Kuittinen. An experiment on using roles of variables in teaching introductory programming. *Computer Science Education*, 15(1):59–82, 2005.

[SK16] Peter Schrammel and Daniel Kroening. 2ls for program analysis - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 905–907. Springer, 2016.

[SM07] Horst Samulowitz and Roland Memisevic. Learning to solve QBF. In *Conference on Artificial Intelligence (AAAI)*, pages 255–260. AAAI Press, 2007.

[SMM11] Pavel Shved, Vadim Mutilin, and Mikhail Mandrykin. Static verification "under the hood": Implementation details and improvements of blast. In *Spring/Summer Young Researchers' Colloquium on Software Engineering*, number 5, 2011.

[Tah10] Ahmad Taherkhani. Recognizing sorting algorithms with the c4. 5 decision tree classifier. In *Program Comprehension (ICPC), 2010 IEEE 18th International Conference on*, pages 72–75. IEEE, 2010.

[TKK+14] Varun Tulsian, Aditya Kanade, Rahul Kumar, Akash Lal, and Aditya V Nori. Mux: algorithm selection for software model checkers, 2014.

[TKM11] Ahmad Taherkhani, Ari Korhonen, and Lauri Malmi. Recognizing algorithms using language constructs, software metrics and roles of variables: An experiment with sorting algorithms. *The Computer Journal*, 54(7):1049–1066, 2011.

[TMK08] Ahmad Taherkhani, Lauri Malmi, and Ari Korhonen. Algorithm recognition by static analysis and its application in students' submissions assessment. In *International Conference on Computing Education Research (ICER)*, pages 88–91. ACM, 2008.

[TNI+16] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. Mu-cseq 0.4: Individual memory location unwindings - (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 938–941. Springer, 2016.

[WACL05] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using datalog with binary decision diagrams for program analysis. In *Asian Symposium on Programming Languages and Systems (APLAS)*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

[WLW04] Ting-Fan Wu, Chih-Jen Lin, and Ruby C. Weng. Probability estimates for multi-class classification by pairwise coupling. *Journal of Machine Learning Research*, 5:975–1005, 2004.

[XHHL08] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Satzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.

[XHHL12] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. Evaluating component solver contributions to portfolio-based algorithm selectors. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *Lecture Notes in Computer Science*, pages 228–241. Springer, 2012.

[Xi98] Hongwei Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University Pittsburgh, 1998.

[ZEL+16] Manchun Zheng, John G. Edenhofner, Ziqing Luo, Mitchell J. Gerrard, Michael S. Rogers, Matthew B. Dwyer, and Stephen F. Siegel. CIVL: applying a general concurrency verification framework to c/pthreads programs (competition contribution). In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 9636 of *Lecture Notes in Computer Science*, pages 908–911. Springer, 2016.