ASSESSING THE SCOPE OF SAFETY PROPERTIES

PETER DAHLBERG

Master's Thesis

Faculty of Computer Science and Mathematics
University of Passau

March 29, 2017

Peter Dahlberg: *Assessing the Scope of Safety Properties,* Master's Thesis,
© March 29, 2017

SUPERVISORS:
Prof. Dr.-Ing. Sven Apel
Prof. Dr. Christian Lengauer
Andreas Stahlbauer

ABSTRACT

Recent advancements in the efficiency of software model checking tools have increased the feasibility to formally verify large real-word software systems. This also moves the formal specification they prove more into the focus of the attention which, for such a software system, typically is complex consisting of a considerably vast number of safety properties. This work determines and assess the scope of such safety properties which we define as that part of the abstract state space of the model checker which is needed to prove the respective property. We present a heuristic to determine the scope of a safety property which greatly leverages of existing analyzes present in the software model checking framework CPAchecker and show visualization approaches which emphasize that part of the state space as well as show the distribution of property scopes inside a program. Our experimental evaluation on two different commonly used benchmark sets with a total of 6986 program and property pairs yields interesting insights into the data coverage, data locality and distribution of property scopes inside the programs of benchmark sets. We also explore the effect of overlapping scopes of safety properties on the performance of a multi-property analysis.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

LIST OF ALGORITHMS

1 INTRODUCTION

Defective computer programs are a major problem in the industry causing huge costs while often even posing threads to safety and security. It is often desirable to find defects before actually running the program using tools for static software analysis on its source code. Especially in security or safety critical systems *formal verification* methods such as *software model checking* which are able to *prove* that the program's behavior adheres to a given *formal specification* are of high value.

A software system is typically formally specified by an often large set of *safety properties*, each describing a particular wanted or unwanted behavior. For example a property may describe a specific rule for the usage of an API like, in a C program, ensuring that *free()* is called on a resource obtained by *malloc()* or may simply forbid the program to ever execute a certain statement.

As tools for *software model checking* mature, getting more and more suitable for verification of real world software, properties become an increasingly interesting subject. One problem is that verifying each property inside an individual analysis run is expensive due to their typically large number. Previous work [4] comes to the conclusion that verifying multiple properties together in one analysis run can safe a lot of verification time. Certain combinations of properties however interact with each other which can lead to a state space explosion and slow down the analysis, sometimes even hinder its completion in a reasonable time frame.

Another aspect where safety properties are interesting is the development and evaluation of software model checking tools. They often rely on a rather fixed set of benchmarks which are used to compare to competing efforts and to which they optimize for but in contrast to real word scenarios often a rather limited set of properties is verified.

1.1 GOALS

The goal of this work is to determine and asses the *scope* of safety properties. The scope of a safety property is that part of the abstract state space (i.e. program statements and variables) which is necessary to prove it.

The first objective is to identify such scopes inside the state space and to develop useful visualizations for usage on a case by case basis. Additionally we want to experimentally asses the scopes of safety properties in existing common benchmark sets quantified by various

metrics to learn about how strongly they differ from each other and if we can spot interesting patterns. Furthermore, we aim to evaluate how scopes of safety properties can be used to understand how overlapping property scopes impact the performance of a multi property analysis.

1.2 CONTRIBUTIONS

The following summarizes the contributions of this work:

- We implement a heuristic analysis built into the software model checking framework CPAchecker [7] which determines the scope of safety properties based on predicate abstraction with adjustable-block encoding [8].
- We describe three visualization methods for property scopes where we reduce the often large visual representation of the state space focusing on making scopes more visible as well as present ways to visually compare scopes of different properties in a program.
- We perform an extensive experimental evaluation on two different benchmark existing benchmark sets. One consisting of 3486 programs with one property for each and the other of 250 programs and 14 properties. We determine and measure the scope of each pair of program and property by various metrics and investigate the performance impact of overlapping scopes on a multi- property analysis on the second set.

1.3 STRUCTURE

In chapter 2 we start with an overview about the software model checking techniques and other important concepts the work bases on. With that knowledge in mind, chapter 3 presents our approach to determine scopes of safety properties. Chapter 4 is concerned with visualizing several aspects of such scopes. In chapter 5 we describe our experiments as well as present and discuss the results. Before we conclude the work in chapter 8 we are proposing opportunities for future work in chapter 6 and take a look at related work in chapter 7.

2 BACKGROUND

To familiarize the reader with the foundations our work is based on, this chapter provides background information on *software model checking* using *predicate abstraction with adjustable block encoding* based the concept of *configurable program analysis*. Furthermore, we talk about *safety properties* and their representation as a *specification automaton*. At the end we give a short overview on *program slicing*.

2.1 SOFTWARE MODEL CHECKING

Software Model checking is a formal approach for static software verification. A software model checker verifies that a given program adheres to a given formal specification in an exhaustive, precise and automatic way by constructing an abstract model of the program. The output of the model checker is either TRUE if the program satisfies the specification, FALSE if a violation is found (typically it outputs a witness in addition) or UNKNOWN if time or space limits are hit. The key challenge is to find sound and complete abstract representations of the typically huge state space of real world programs to make the analysis efficient.

2.2 CONFIGURABLE PROGRAM ANALYSIS

The work relies on the model checking framework CPAchecker [7] for verification of programs in the C programming language, which implements *configurable program analysis* (CPA) $\mathbb{D} = (D, \rightsquigarrow, \mathit{merge}, \mathit{stop}, \mathit{prec})$ [5] with dynamic precision adjustment [6]. The following explains key concepts and definitions needed later following closely the definitions in the literature.

2.2.1 *Control-Flow Automata*

A *control-flow automaton* $\mathrm{CFA}(L, l_0, G)$ describes a program as a set of control locations L with $l_0 \in L$ as the entry location. These locations are connected by a set of *control-flow edges* $G \subseteq L \ \mathrm{Ops}\ L$ with Ops being the operations which let the control flow from one location to another. Each such operation may either be an assignment operation or an assume operation which represents the boolean condition allowing the control flow during execution.

In practice a CFA is typically enriched by additional types of *control-flow edges* representing e.g. variable declarations or function calls and

returns to provide the analysis with additional information the programming language provides. Figure 1 shows an example of a CFA constructed from a C program with two functions.

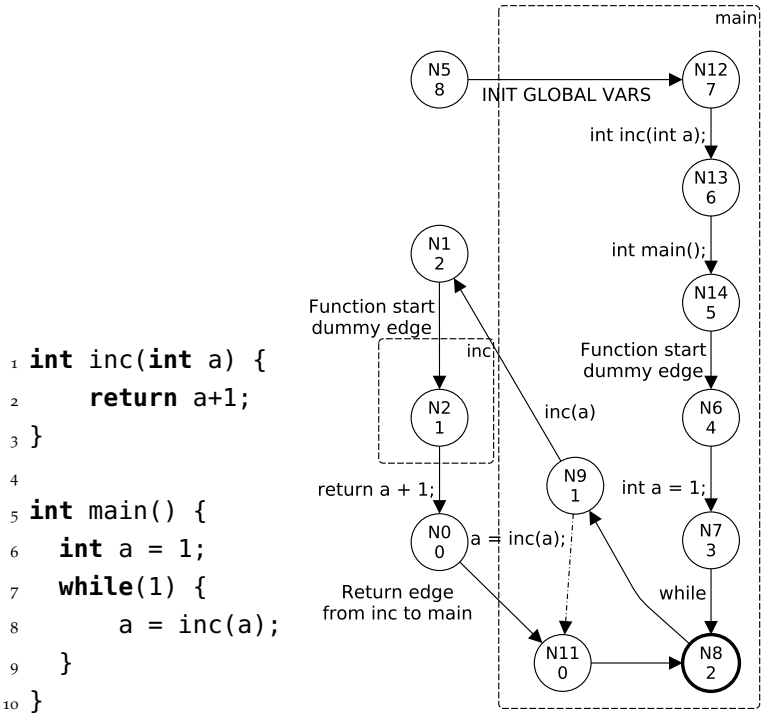


Figure 1: Expample of a CFA for a simple program as represented in CPAchecker

2.2.2 *Concrete States and Reachability*

A *concrete state* of a program c is a specific assignment of all variables of the program to a value at a specific location $l \in L$. The set of all concrete states of a program is denoted by C and a subset r of C is called a *region*. To make a transition from one *concrete state* to another we go along a control flow edge $g \in G$ which leads to a labeled transition relation $\xrightarrow{g} \subseteq C \times \{g\} \times C$ and finally to a transition relation $\longrightarrow$ which is the union of $\xrightarrow{\hat{g}}$ for every control flow edge $\hat{g} \in G$.

"A *concrete state* c_n is *reachable* from a region r denoted by $c_n \in \mathtt{Reach}(r)$ if there exists a sequence of *concrete states* c_i such that $c_0 \in r$ and for all $1 \leqslant i \leqslant n$, we have $c_{i-1} \longrightarrow c_i$." [6]

2.2.3 *Data Structures and Operators of CPA*

According to the literature [6] the *abstract domain* $D = (C, \xi, [[\cdot]])$ consists of the set of *concrete states* C, a semi-lattice $\xi = (E, \top, \bot, \sqsubseteq, \sqcup)$ of *abstract states* and a concretization function $[[\cdot]]$ which takes an *ab-*

stract state and returns the corresponding set of *concrete states*. The set E contains the abstract domain elements (possibly infinite), including the top element $\top$ and the bottom element $\bot$. $\sqsubseteq$ is a partial order over the elements of E and $\sqcup : E \times E \to E$ is the join operator. For soundness of the analysis it is required that:

1. $[[\top]] = C$ and $[[\bot]] = \emptyset$
2. $\forall e, e' \in E : e \sqsubseteq e' \implies [[e]] \subseteq [[e']]$
3. $\forall e, e' \in E : e \sqcup e' \supseteq [[e]] \cup [[e']]$
   (The join operator must always over approximate if not precise)

The set Π determines the possible *precisions* of the abstract domain. Each abstract state has a (possibly different) precision π consisting of elements from Π assigned, thus a pair (e, π) is called an *abstract state with precision* π.

The *transfer relation* $\rightsquigarrow \subseteq E \times G \times E \times \Pi$ assigns new abstract states e' to an existing abstract state e . We write $e \overset{g}{\rightsquigarrow} (e', \pi)$ if such a transfer with the *control-flow edge* $g \in G$ exists. For every fixed precision $\rightsquigarrow$ is required to over approximate the respective concrete transitions:

$$\forall e \in E, g \in G, \pi \in \Pi : \bigcup_{e \overset{g}{\rightsquigarrow} (e', \pi)} [[e']] \supseteq \bigcup_{c \in [[e]]} \left\{ c' | c \xrightarrow{g} c' \right\}$$

The *merge operator* $\mathtt{merge} : E \times E \times \Pi \to E$ weakens the second parameter using information of the first and returns a new abstract state with the precision of the third parameter. The merge result can only be more abstract than the second parameter:

$$\forall e, e' \in E, \pi \in \Pi : e \sqsubseteq \mathtt{merge}(e, e', \pi)$$

The *termination check* $\mathtt{stop} : E \times 2^E \to \mathbb{B}$ checks if the abstract state given as the first parameter is already covered by the set of abstract states given by the second with the precision given by the third. The following is required:

$$\forall e \in E, R \subseteq E, \pi \in \Pi : \mathtt{stop}(e, R, \pi) \implies [[e]] \subseteq \bigcup_{e' \in R} [[e']]$$

The *precision adjustment* function $\mathtt{prec} : E \times \Pi \times 2^{E \times \Pi}$ computes a new abstract state and a new precision, for a given abstract state with precision and a set of abstract states with precision. The precision adjustment function

may perform widening of the abstract state, in addition to a change of precision. The following is required:

$$\forall e, e' \in E, \pi, \pi' \in \Pi, R \subseteq E \times \Pi :$$
$$(e', \pi') = prec(e, \pi, R) \implies [[e]] \subseteq [[e']]$$

[formal definitions reproduced directly from 6]

2.2.4 *Reachability algorithm for CPA*

The algorithm CPA_{alg} (algorithm 1) computes from an initial abstract state with precision a set of reachable abstract states by producing successor abstract states through the usage of the transfer relation of the given CPA. This is iterated until a fixed point is reached. The result is a set $reached$ of abstract states which is thus an over-approximation of the set of reachable concrete states.

Algorithm 1: $CPA_{alg}(\mathbb{D}, e_0, \pi_0)$ (taken from [6])

Input : A CPA $\mathbb{D} = (D, \rightsquigarrow, merge, stop, prec)$; an initial abstract state $e_0 \in E$ where E denotes the set of elements of the semi-lattice of D; an initial precision $\pi_0 \in \Pi$
Output: A set of reachable abstract states
begin
  waitlist := $\{(e_0, \pi_0)\}$
  reached := $\{(e_0, \pi_0)\}$
  while waitlist $\neq \emptyset$ **do**
    $(e, \pi) :=$ pop(waitlist)
    `// Adjust the precision`
    $(\hat{e}, \hat{\pi}) = prec(e, \pi, reached)$
    foreach e' *with* $\hat{e} \rightsquigarrow (e', \hat{\pi})$ **do**
      foreach $(e'', \pi'') \in$ reached **do**
        `// Combine with existing abstract state`
        $e_{new} := merge(e', e'', \hat{\pi})$
        if $e_{new} \neq e''$ **then**
          waitlist := $(waitlist \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
          reached := $(reached \cup \{(e_{new}, \hat{\pi})\}) \setminus \{(e'', \pi'')\}$
      if $\neg stop(e', reached)$ **then**
        waitlist := $waitlist \cup \{(e', \hat{\pi})\}$
        reached := $reached \cup \{(e', \hat{\pi})\}$
  return reached

2.2.5 *Abstract Reachability Graph*

The abstract states contained in the set $reached$ are the nodes of an *abstract reachability graph* (ARG). The ARG is a directed graph with the

root node being the initial abstract state given to CPA_{alg}. An edge in the ARG connects an abstract state with its successor.

2.2.6 *Composition of CPAs*

A typical analysis is composed of several component CPAs: most often in form of a CPA which implements the main part of the analysis, one or more CPAs which check the specification and several generic helper CPAs which e.g. track the call-stack or function pointers.

To compose two[1] CPAs $\mathbb{D}_1$ and $\mathbb{D}_2$ which share the same set of concrete states into one, a *composite* CPA $C = (\mathbb{D}_1, \mathbb{D}_2, \Pi_x, \rightsquigarrow_x, merge_x, stop_x, prec_x)$ as described in the literature [6] can be used. With the cross product $E_1 x E_2$ of their respective sets of abstract states and a composite set of precisions Π_x a composite transfer relation $\rightsquigarrow_x$, a composite merge operator $merge_x$, a composite termination check $stop_x$ and a composite precision adjustment operator $prec_x$ can be defined, analogous to those in section 2.2.3. These composites are expressions over the components of $\mathbb{D}_1, \mathbb{D}_2$ and Π_x $(\rightsquigarrow_i, merge_i, stop_i, prec_i, [[\cdot]]_i, E_i, \top_i, \bot_i, \sqsubseteq_i, \sqcup_i)$.

One important aspect of the *composite* CPA is the introduction of the strengthening operator $\downarrow$ into each of the component CPAs. It is called inside $\rightsquigarrow_x$ after all transfer relations of the wrapped component CPAs are finished and allows the respective component CPA to take information out of the abstract states of the other component CPAs and strengthen it's own abstract state using it. The version of CPAchecker used here is extended by a post precision adjustment strengthening operator $\downarrow^{prec}$ which follows the same idea and is called at the end of $prec_x$.

2.3 PREDICATE ANALYSIS WITH ADJUSTABLE-BLOCK ENCODING

This section recites and explains key concepts of the predicate analysis with adjustable-block encoding (ABE) [8] which serves as a basis for the approach described in this work.

2.3.1 *Logical Representation of the Concrete State Space*

Given the definitions in section 2.2.1 and section 2.2.2 the following describes how to represent a program using first-order logic. A *region* is represented as a first-order formula φ with free variables from the set X of program variables i.e. φ represents all concrete states that imply φ.

The *concrete semantics* of an operation $op \in Ops$ is defined by the strongest postcondition operator $SP_{op}(\cdot)$. For a formula φ. $SP_{op}(\varphi)$

1 This can be trivially extended to any finite number of CPAs

represents all abstract states which are reachable from the region represented by φ after executing op.

$$\mathrm{SP}_{op}(\varphi) = \begin{cases} \exists \hat{s} : \varphi_{[s \to \hat{s}]} \wedge s = e_{[s \to \hat{s}]} & \text{for an assignment } s := e \\ \varphi \wedge p & \text{for an assume operation assume(p)} \end{cases}$$

A *path* σ is defined as a sequence $\langle (op_1, l_1), \cdots, (op_n, l_n) \rangle$ of pairs of program locations and the operation leading to the respective location. A path σ is called a *program path* if it starts at the entry location l_0 and walks along the CFA i.e. there exists a edge $g = (l_{i-1}, op_i, l_i)$ inside the CFA for every $1 \leqslant i \leqslant n$.

The *concrete semantics* of a program path is defined as the successive application of the strongest postcondition operator: $\mathrm{SP}_\sigma(\varphi) = \mathrm{SP}_{op_n}(\cdots \mathrm{SP}_{op_1}(\varphi) \cdots)$. The result of $\mathrm{SP}_\sigma(\varphi)$ is called a *path formula*. "A program path is *feasible* if $\mathrm{SP}_\sigma(\mathrm{true})$ is satisfiable. A concrete state is *reachable* if there exists a feasible program path to its location. A program is *safe* if no error location is reachable." [8]

2.3.2 *Predicate Abstraction*

As it is not practically possible to feed whole programs into an SMT solver, abstractions need to be performed. For the predicate analysis the literature primarily knows about the *Cartesian predicate abstraction* and the *boolean predicate abstraction* with the latter being more precise.

Let $\mathcal{P}$ be the set of predicates over program variables in a quantifier-free theory (a predicate could be e.g. $x = 7$). A *precision for formulas* $\pi \in \Pi$ consists of a finite number of elements from $\mathcal{P}$. The *boolean predicate abstraction* $(\varphi)^\pi$ of a formula φ thus is the strongest boolean combination of all the predicates from the precision π. A technique to obtain the *boolean predicate abstraction* using a SMT solver can be found in the literature [8].

Using the boolean predicate abstraction, a abstract version of the strongest postcondition operator $(\mathrm{SP}_{op}(\varphi))^\pi$ can be defined which first applies $\mathrm{SP}_{op}(\varphi)$ and then performs the boolean predicate abstraction. Given a suitable precision π, $(\mathrm{SP}_{op}(\varphi))^\pi$ greatly over-approximates $\mathrm{SP}_{op}(\varphi)$ leading to a most of the time small enough formula to solve inside a practical time-frame.

2.3.3 *CPA for Predicate Analysis with ABE*

Now we explain the workings of the predicate analysis with ABE on the basis of its implementation as a CPA as described in [8].

The elements $e \in E$ of the *abstract domain* are tuples

$$(l, \psi, l^\psi, \varphi) \in (L \cup \{l_\top\}) \times \mathcal{P} \times (L \cup \{l_\top\}) \times \mathcal{P}$$

with l modeling the program counter, ψ being the abstraction formula consisting of a boolean combination of predicates occurring in the precision Π and φ being a conjunctive path formula representing some or all paths from l^ψ to l. There is a fundamental distinction between an *abstraction state* and a *non-abstraction state*: an abstract state e is an abstraction state iff $\varphi = \mathtt{true}$ and $l^\psi = l$. The top element of the lattice is defined as $\top = (l_\top, \mathtt{true}, l_\top, \mathtt{true})$ The partial order $\sqsubseteq$ is defined such that for elements $e_1, e_2 \in E$ with $e_i = (l_i, \psi_i, l_i^\psi, \varphi_i)$ the following holds:

$$e_1 \sqsubseteq e_2 \Leftrightarrow (e_2 = \top) \vee ((l_1 = l_2) \wedge (\psi_1 \wedge \varphi_1 \Rightarrow \psi_2 \wedge \varphi_2))$$

The join operator $\sqcup$ yields the least upper bound of the operands according to the partial order defined above.

The *transfer relation* $\rightsquigarrow$ operates in two modes producing either abstraction states or non-abstraction states. The decision is made by a *block-adjustment operator*[2] $\mathtt{blk}(e, g)$ (with g being the control-flow edge) which is given to the analysis as a parameter. If a non-abstraction state is to be produced for the new location $l' \rightsquigarrow$ takes the path-formula φ and computes a new path-formula $\varphi' = \mathrm{SP}_{op}(\varphi)$ while copying the abstraction formula ψ and its location l^ψ from the old state. To produce an abstraction state φ' is set to $\mathtt{true}$, $l^{\psi'}$ is set to l' and a new abstraction formula ψ' is computed: $\psi' = (\mathrm{SP}_{op}(\varphi \vee \psi))^\pi$.

The *merge operator* is able to combine two abstract states $e_1, e_2 \in E$ with $e_i = (l_i, \psi_i, l_i^\psi, \varphi_i)$ if their location is the same and their abstraction formula and the location of the abstraction formula is the same. It does so by building a disjunction of the two path formulas, or formally $\mathtt{merge}(e_1, e_2, \pi) =$

$$\begin{cases} (l_2, \psi_2, l_2^\psi, \varphi_1 \vee \varphi_2) & \text{if } (l_1 = l_2) \wedge (\psi_1 = \psi_2) \wedge (l_1^\psi = l_2^\psi) \\ e_2 & \text{otherwise} \end{cases}$$

The *termination check* checks if $e \in E$ is covered by another state in the reached set R by using $\sqsubseteq$:

$$\forall e \in E, R \subseteq E : \mathtt{stop}(e, R) = \exists e' \in R : (e \sqsubseteq e')$$

2.3.4 *Block Adjustment*

The main goal of Adjustable Block Encoding is to reduce expensive calls to the SMT solver. There are multiple possibilities to implement the *block-adjustment operator* $\mathtt{blk}(g, e)$: It can be be tailored to the SMT solver to not overwhelm it e.g. computing abstractions

2 If the transfer leads to an error location $\mathtt{blk}(e, g)$ is ignored and an abstraction state is produced

when the path formula becomes too large. Additionally, one can compute abstractions at specific locations inside the CFA, such as function entries/returns, loop heads or join locations where the control flow meets. It is also possible to configure a simplified variant of the predicate analysis which computes always abstraction states by using a *block-adjustment operator* which always returns true, called $\mathtt{blk}_{\mathtt{SBE}}$.

2.3.5 *Computing a Suitable Precision*

A vital part of the predicate analysis is to compute the predicates to put into the precision Pi. The goal is to track as few as possible to archive good performance but enough to actually prove the program. To archive this an approch based on *Counterexample Guided Abstraction Refinement* (CEGAR) [12] is used which learns the predicates from spurious counterexamples.

Initially the predicate analysis is started with an empty[3] precision $\pi \in \Pi$. As illustrated in Figure 2 the next step is to check if an error location was reached. If that is not the case the abstraction was fine-grained enough and the program is proven safe, otherwise the counterexample leading to that location is is checked for feasibility. A feasible counterexample implies that the program is unsafe. An infeasible one on the other hand indicates that the analysis was too abstract and the precision needs to be refined for the next iteration of the CEGAR loop.

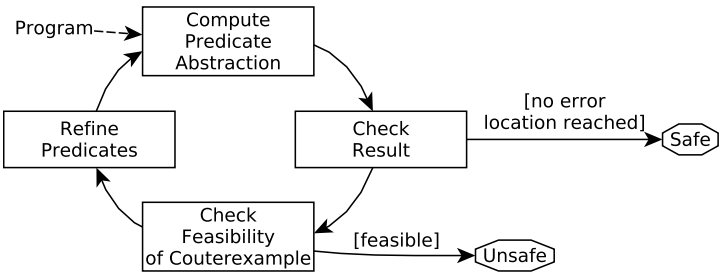


Figure 2: CEGAR Loop

A *counterexample* is a *program path* leading to an error location which gets then converted to a path formula. When using ABE multiple paths through the ARG to an abstract state at an error location are possible but there is still only one formula built: as a merge never occurs at abstraction states the ARG can be simplified to a tree looking only at those, so the final path formula can be constructed from the path formulas which lead to the abstraction states (which may already be disjunctions of multiple path formulas). The *feasibility* check then boils down to asking an SMT solver about the satisfiability of that formula. New predicates to *refine* the precision are learned by

3 It's also possible to prepopulate π with predicates retrieved through some other heuristic

computing the Craig Interpolant [14] of path formulas leading to consecutive abstraction states using, again an SMT solver while examining the counterexample following the method of Lazy Abstraction [18].

2.4 SAFETY PROPERTIES

According to Apel et al. [4] a *safety property* p represents a desired behavior of a program. We focus on properties which can be expressed by a formal specification which represent a set of desired finite program executions and can be expressed by temporal-logic formula or by a finite automaton [19].

"The set $\mathbb{P} = \{p_1, \cdots, p_n\}$ consists of all *properties* of a program to be verified. A *partitioning* $\mathcal{P} \subseteq 2^X$ of a set X is a set of non-empty sets where all elements $P_1, P_2 \in \mathcal{P}$ with $P_1 \neq P_2$ are pairwise disjoint and $X = \bigcup_{P \in \mathcal{P}} P$." [4]

2.4.1 *Specification Automata*

One or more properties can be represented by a *specification automaton*. "A *specification automaton* encodes a set of properties and observes, but not restricts, the state space of an analysis run. A *specification automaton* $(Q, \Sigma, \delta, q_0, F)$ for a $CFA(L, l_0, G)$ is a non-deterministic finite automation with a finite set of control states Q, an alphabet $\Sigma \subseteq 2^G \times S^* \times R^*$, a transition relation $\delta \subseteq Q \times \Sigma \times Q$, an initial control state $q_0 \in Q$ and a set F of accepting control states." [4] S^* represents all possible sequences over assignment operations and R^* represents all possible sequences over assume operations respectively. A transition of the automaton into a state $q_p \in F$ is equivalent to a violation the property $p \in \mathbb{P}$ (a path through the CFA was found which violates p)."A symbol $(\gamma, s, r) \in \Sigma$ consists of a set γ of control flow edges, a sequence s of assignment operations and and a sequence r of assume operations." [4]

The described automaton supports on-the-fly weaving [4] of operations into the transfer relation of the predicate analysis which allows to leverage precision adjustment techniques as described in section 2.3.5. The automaton allows three modes of operation:

- A *pure automata-based mode* where nothing is weaved in and every possible state of the specification is represented as a state of the automaton
- A *pure weaving mode* where the automaton stays in its initial state q_0 until it finally transitions into a target state $q_p \in F$. Every state of the specification is weaved in as variables.

- A *hybrid mode* which combines the first two modes. This can be used to only trigger the weaving process when the program enters a context where the property is relevant.

2.4.2 *Specification Analysis and Specification Language*

CPAchecker implements a specification analysis as a composite CPA $\mathbb{D}_s$ which is based on the previously described automatons. Automatons are given to $\mathbb{D}_s$ in a text based format. Details about $\mathbb{D}_s$ as well as the specification language can be found in the literature [4, 22].

2.4.3 *Relevance of a Property*

"A property p is *relevant* for a given program if the specification automaton has a transition τ to a control state q' with $p \in \mathbb{P}(q')$ or $p \in \mathbb{P}(\tau$ and τ matches a control-flow edge of the program." [4]

2.5 PROGRAM SLICING

Program slicing [11, 16] is a simplification technique for programs which keeps only parts of the program which do effect specific semantic aspects and deletes program statements which do not. These semantics are given by a *slicing criterion*. The literature generally differentiates between *static*, *dynamic* and *conditioned* slicing.

The most basic form is *static slicing* using a slicing criterion (V, n) with V being a set of variables and n being a specific point of interest inside the program. From a given slicing criterion either a *forward* slice or a *backward* slice can be constructed. A backwards slice contains all program statements which have some effect on the variables in V at n, a forward slice contains all program statements which are affected by the variables V at n. Static slicing makes no assumptions on the input of the program, i.e. it considers all possible executions of the program.

Dynamic slicing on the other hand extends the slicing criterion by a set of input values of the program which restricts the set of possible executions. A dynamic slice only includes the statements which affect (or are affected by) the variables in V at n during these executions. If we want to be less precise about the input values and thus the possible executions *conditioned slicing* can be used where rather than a set of input values a boolean expression over the input values is given, e.g. for possible input values x, y we could restrict the executions of the program to compute a slice for to cases where $x > y + 1$ holds.

3

DETERMINATION OF PROPERTY SCOPES

This chapter presents an heuristic approach which was implemented in CPAchecker along with this work to approximate the *scope* of a safety property by primarily observing the predicate analysis as described in section 2.3 and the specification analysis described in section 2.4.2.

We define the *scope* of a *safety property* $p \in \mathbb{P}$ for a given program as a projection of an ARG consisting of a subset Ω of its abstract states which are required to prove that p holds.

In the context of program slicing a safety property $p \in \mathbb{P}$, specified through a specification automation a can be seen as an implicitly stated conditioned slicing criterion for a backwards slicing from the program locations where a transitions into a target state with the condition being equivalent to the set of violating program executions described by p. With this we define the set of control-flow edges $\Gamma = \{\text{control-flow edge entering } e | e \in \Omega\}$ as the *scope slice* of p.

3.1 PROPERTY SCOPE CPA

The *property scope* CPA $PS = (D_{PS}, \rightsquigarrow_{PS}, merge_{PS}, stop_{PS}, prec_{PS})$ is designed to be a component of a composite analysis C_{PS} along with the *location* CPA which tracks the location (i.e. nodes of the control-flow graph) the abstract state is at, the *call-stack* CPA which tracks entering and exiting functions by maintaining a call-stack, the previously described *predicate* CPA and one or more *specification automaton* CPAs.

3.1.1 *Property Scope Location*

A *property scope location* $loc_{PS}(g, id^{PS}, \nabla_{PS})$ consists of a control flow edge $g \in G$, an identifier for an abstract state id^{PS} and a *property scope reason* ∇_{PS} with $\nabla_{PS} \in \{pred_\nabla, automa_\nabla, \top_\nabla\}$. This CPA (implicitly) yields a final set of *property scope locations* which describes an approximation of the *scope* of the properties used in the analysis by using id^{PS} to map back into the abstract state space of the analysis.

3.1.2 *Abstract Domain*

The elements E of the *abstract domain* D_{PS} are tuples consisting of the following:

- The state identifier id^{PS}
- The predecessor property scope state e_{prev}
- A set LOC_{PS} of *property scope locations*
- A set LOC^{c}_{PS} of candidate *property scope locations*
- A set $ASTATES = \bigcup_{1 \leqslant i \leqslant n} (sa^i, q^i, pdm^i)$ where sa^i is a specification automaton out of n automatons which are part of the composite analysis and q^i is the state it's in at the current composite state. pdm^i is the number of matches of the respective automaton
- An abstraction Formula ψ^{lc}
- AM^{PS}, a set of pairs (sa, q_0) where sa is an element of the set of specification automatons which are part of the composite analysis and q_0 being the initial state of sa. It may contain at most one such pair for each specification automaton which is part of the composite analysis

The initial state at the beginning of the analysis is the tuple $(id^{PS} = 0, e_{prev} = \top, LOC_{PS} = \emptyset, LOC^{c}_{PS} = \emptyset, ASTATES = \emptyset, \psi^{lc} = true, AM^{PS} = \emptyset)$. The partial order for abstract states $e_1, e_2 \in E$ is defined by the following relation: $e_1 \sqsubseteq e_2 \Leftrightarrow (e_2 = \top) \vee \Big(\big(LOC_{PS_1} \subseteq LOC_{PS_2}\big) \wedge \big(LOC^{c}_{PS_1} \subseteq LOC^{c}_{PS_2}\big) \Big)$. The join operator $\sqcup$ returns the least upper bound according to the partial order.

3.1.3 *Transfer Relation with Strengthening*

The *transfer relation* $\rightsquigarrow_{PS}$ produces an intermediate abstract state $e_{\rightsquigarrow}$ which will be enriched inside strengthening steps. It carries ψ^{lc}, AM^{PS}, LOC^{c}_{PS} over from the previous state. ASTATES is set to the empty set as well as AM^{PS}.

Inside the *strengthening* operator $\downarrow_{PS} e_{\rightsquigarrow}$ is transformed to $e_{\downarrow}$ by performing the following steps:

1. The state identifier id^{PS} is set to a new unused number id^{PS}_{new}
2. The set of candidate property scope locations LOC^{c}_{PS} is extended by a new property scope location consisting of the current control flow edge g, the current call-stack cst extracted from the call-stack state, the identifier $id^{PS}_{n} ew$ of the new state and a placeholder scope reason $\top_{\nabla}$
3. The set of automaton states $ASTATES_{\downarrow}$ is filled by examining the states of the specification automaton CPAs
4. The previous set of specification states $ASTATES_{prev}$ and current $ASTATES_{\downarrow}$ are observed:

- If any automaton sa^i has matched ($pdm^i_{\downarrow} > pdm^i_{prev}$) a new *property scope location* is added to LOC_{PS} as we can safely assume that locations where an automaton matches are always part of the scope of a property. The new property scope location consists of the current control-flow edge g, the current call-stack cst extracted from the call-stack state and the scope reason $autom_{\nabla}$.
- For each automaton sa^i we add a new pair (sa^i, q^i) to AM^{PS} if sa_i transitioned from its initial state q^i_0 to another state $q^i \neq q^i_0$, remove the corresponding pair from AM^{PS} if it transitioned from a state $q^i \neq q^i_0$ back to the initial state q^i_0 or otherwise do nothing

3.1.4 *Precision Adjustment with Strengthening*

The *precision adjustment* operator $prec_{PS}$ does not change the state so $e_{\downarrow} = e_{prec}$. Because the predicate CPA as implemented in CPAchecker delays the abstraction computation to its precision adjustment operator the post precision adjustment strengthening operator $\downarrow_{prec_{PS}}$ is used to incorporate information about predicate abstractions. See algorithm 2 for the definition of $\downarrow_{prec_{PS}}$.

Algorithm 2: $\downarrow_{prec_{PS}}$

Input : e_{prec}, the result of $prec_{PS}$
Output: $e_{\downarrow_{prec}}$
Data: Predicate abstract state e_{pred} with abstraction formula ψ
begin
  $e_{\downarrow_{prec}} := \texttt{copy}(e_{prec})$
  if `isAbstractionState` (e_{pred}) **then**
    foreach $(g^c, \_) \in e_{prec}\left[LOC^c_{PS}\right]$ **do**
      if `usesAnyVariable`$(g^c, \psi) \wedge e_{prec}\left[\psi^{lc}\right] \neq \psi$ **then**
        $e_{\downarrow_{prec}}[LOC_{PS}] := e_{\downarrow_{prec}}[LOC_{PS}] \cup \{g^c, pred_{\nabla}\}$
    $e_{\downarrow_{prec}}\left[\psi^{lc}\right] := \psi$
    $e_{\downarrow_{prec}}\left[LOC^c_{PS}\right] := \emptyset$

The function $usesAnyVariable(g, \psi)$ tests if any variable v contained in in the formula ψ is actually read or written by the control flow edge g. This step is useful to filter out control-flow edges which are unable to contribute to the scope, especially if we do not configure single block encoding by using blk_{sbe} we would have to assume that the whole block is part of the scope because the abstraction formula abstracts over the whole block. The implementation utilizes a basic dependency analysis already present in CPAchecker called *variable classification* which provides an over-approximated mapping between control-flow edges and variables.

$usesAnyVariable(g, \psi)$ can also be seen in the context of program slicing. We construct a slicing criterion (V, g) with V being the set of variables encoded in ψ and test if the control-flow edge leading to the abstraction state e_{pred} is part of the static forward slice under that criterion.

3.1.5 *Merge Operator and Termination Check*

The *merge operator* $merge_{PS}(e_1, e_2, \pi)$ combines the components of e_1 and e_2 to a new element e_{new}:

$$id^{PS}_{new} = \text{ some not yet used number} \tag{1}$$
$$e_{prev} = \top \tag{2}$$
$$LOC_{PS_{new}} = LOC_{PS_1} \cup LOC_{PS_2} \tag{3}$$
$$LOC^c_{PS_{new}} = LOC^c_{PS_1} \cup LOC^c_{PS_2} \tag{4}$$
$$AM_{PS_{new}} = AM_{PS_1} \cup AM_{PS_2} \tag{5}$$
$$\psi^{lc}_{new} = \psi^{lc}_2 \tag{6}$$
$$ASTATES_{new} = \bigcup_{1 \leqslant i \leqslant n} \left(sa^i_2, q^i_2, max\left(pdm^i_1, pdm^i_2\right)\right) \tag{7}$$

We implicitly presume that $LOC_{PS_{new}}$ and $LOC^c_{PS_{new}}$ contain copies of their *property scope locations* with their state id set to id^{PS}_{new}. It is possible to assume that $\psi^{lc}_2 = \psi^{lc}_1$ and $ASTATES_2$ and $ASTATES_1$ only differ in the number of matches because e_{new} is only used inside the composite CPA if all its other components agree the merge when the control flow meets. ψ^{lc} is only changed when an abstraction state occurs but the predicate CPA only merges iff both of its abstract states have the same abstraction state as a predecessor. The specification automaton CPAs do not merge if the automatons are in the same state.

The *termination check* $stop_{PS}$ is configured to always return true because we only want to observe what the other component CPAs are doing.

3.1.6 *Incompleteness for Violated Properties*

The scope of one or a set of properties determined by this CPA is likely incomplete if at least one property is violated. There may be more than one path through the CFA which possibly cause the property being violated but the predicate CPA causes the analysis to stop declaring the program unsafe after it finds some feasible counterexample and never explores further possibilities.

4 VISUALIZATION OF PROPERTY SCOPES

The following presents tree different ways to visualize the scope of properties. We present reductions of the visual representation of the ARG. Furthermore we visualize the distribution inside the program and the overlap of several property scopes by using arc diagrams.

4.1 OVERLAP AND DISTRIBUTION INSIDE PROGRAMS

This section presents an approach to visualize how the scope of a property is distributed throughout the respective program at a function level. We use *arc diagrams* [17] which due to their one dimensional nature are well suited for comparing scopes of different properties for a program and thus allowing a optical assessment of how much they overlap.

4.1.1 *Property Scope Call Graph*

A *property scope call graph* $PSC = (N^{PSC}, E^{PSC})$ is a directed graph of nodes N^{PSC} and edges E^{PSC} which is built by traversing the ARG of the property scope analysis described in section 3.1 . A node $n^{PSC} \in N^{PSC}$ represents a function of the program and an edge $e^{PSC} \in E^{PSC}$ represents the presence of at at least one call from a function f_1 to a function f_2 inside the ARG, i.e. the control-flow edge between two abstract states indicates a function call. Nodes have some additional attributes attached, most importantly:

- The *property scope importance* imp^{ps} of the function which is a value between 0 and 1 indicating how much of the function takes part in the *scope* of the property. It is the ratio of abstract states which are inside the function to abstract states which are inside the function and part of the *scope*.
- The number of times ncall^{ps} the function is called inside the ARG.

4.1.2 *Arc Diagrams for Property Scopes*

We use the previously described $PSC = (N^{PSC}, E^{PSC})$ as a basis to draw an *arc diagram*. The nodes (representing functions of the program) are laid out as circles in a straight line with the function names annotated below. Two nodes $n_1, n_2 \in N^{PSC}$ are connected by an arc above them if $(n_1, n_2) \in E^{PSC} \vee (n_2, n_1) \in E^{PSC}$, i.e. the direction

of the function calls is ignored. The size of the circle representing a node $n \in N^{PSC}$ is given by the radius r_n which indicates the value of $ncall_n^{ps}$ with $r_n = 1 + 0.5 \cdot \log_{10}(ncall_n^{ps})$ if $ncall_n^{ps} > 0$ else $r_n = 0$. Finally the value of imp_n^{ps} for a node n is represented by coloring the respective circle using a heat coloring which can be seen in Figure 3. At the left side of the line of circles the *relevant properties* for which the scope is visualized are annotated.

Figure 3: Heat coloring used to represent the value of the property scope importance inside the diagrams (for values of 0 white is used)

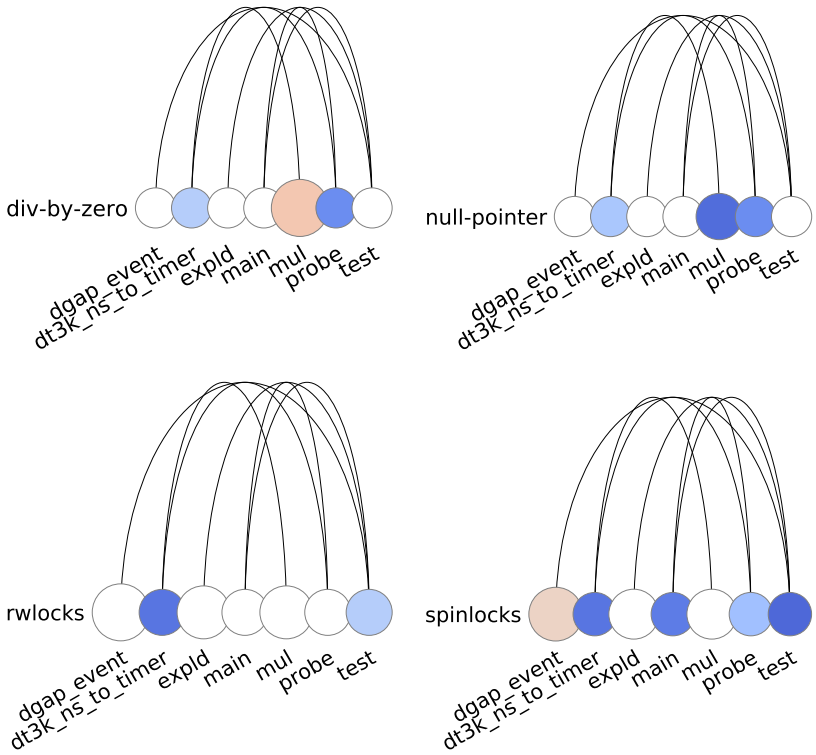


Figure 4: Property scope Arc diagrams for a file *multiprops.c* using 4 different properties

Figure 4 shows four examples of such diagrams. Each represents the result of a different run of the property scope analysis on the same program *multiprops.c* with one distinct relevant property. This provides us with a clear overview about the distribution of the scopes of the respective properties throughout the functions of the program and enable us to compare them. As one example we see that the scope of the property *div-by-zero* seems to be mostly concentrated in the function *mul* where the scope of *spinlocks* clearly concentrates

inside the function *dgap_event* but also spreads comparatively widely throughout the program.

For large real world programs arc diagrams tend to grow considerably in size taking up a lot of space while making overlaps hard to see. Figure 5 presents an example of an approach to draw multiple PSCs for the same program but different properties into one combined arc diagram. Let $PSC_i = (N_i^{PSC}, E_i^{PSC})$ be the i'st of m such PSCs. To save space we only show nodes for the functions which are present in at least one of the ARGs, i.e. a function out of all functions inside the program $f \in F_{prog}$ is shown if $ncall^{ps} > 0$ at at least one node $n_f \in \bigcup_i^{PSC}$ where n_f represents f. For every N_i^{PSC} the respective nodes are drawn in a straight line as explained initially while using a fixed ordering for the functions, but now these lines are stacked on top of each other. Only one set of arcs is drawn at the top where an arc is drawn if $\exists i$ with $1 \leqslant i \leqslant m : (n_1, n_2) \in E_i^{PSC} \vee (n_2, n_1) \in E_i^{PSC}$. At this point we trade in potential information loss for a more compact representation so it might be desirable to single out a PSC_i into a separate diagram if its set of function calls differs considerably from the others.

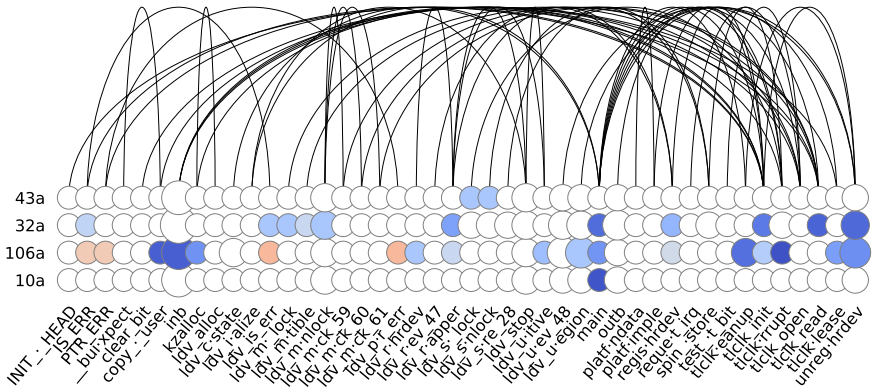


Figure 5: A property scope arc diagram for a linux kernel module *drivers-char-tlclk.c* from the *LDV-250 set* as explained in section 5.2.1 using 4 different properties displayed in a combined view

4.2 PROPERTY SCOPES INSIDE THE STATE SPACE

In this section we present visualizations based on two example programs, *fopen_malloc_expl.c* as shown in listing 1 and *fopen_malloc_noexpl.c* as shown in listing 2. We determine the combined scope of the properties *malloc.spc* (listing 3) and *fopen.spc* (listing 4) by using the approach presented in chapter 3, running the analysis with both specification automatons in parallel. This results in an ARG with annotated scope information. As an ARG typically grows very large, even for such small example programs we want to reduce it showing only the parts which are part of the scope of the respective properties.

4.2.1 *Property Scope Graph*

First we present the *property scope graph* PSG which is a reduced visual representation of the ARG showing only the abstract states in scope of the properties. Figure 7 and Figure 8 show such a graph for each of the previously stated examples. The *property scope graph* is built by collapsing the whole subgraph between two abstract states e_1, e_2 which are part of the scope into one uncolored summary node which is annotated with the number of states inside that subgraph and the number of paths through that subgraph from e_1 to e_2. If an abstract state e which is in scope is only followed by a subgraph of abstract states which are not in scope we collapse that subgraph into an uncolored node to indicate the existence of that "irrelevant" subgraph.

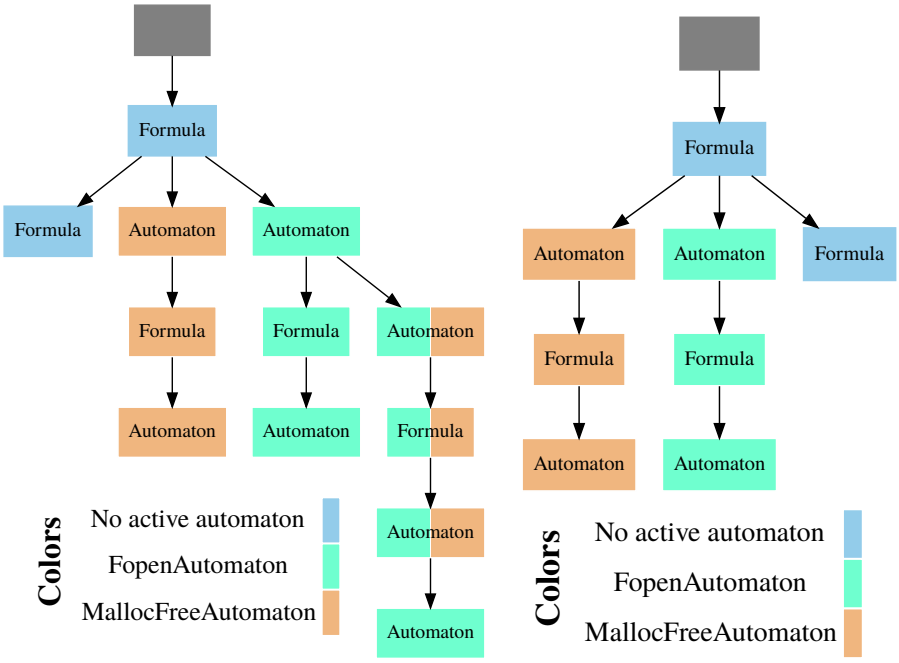
A node representing an abstract state e which is part of the scope is colored with one or more colors. Each distinct color c_a represents a specification automaton a and is shown if a is not inside its initial state at e. If no automaton is outside of its initial state at e we use a special color c_s reserved for this case. An automaton being out of it's initial state is used as a rough estimate for attributing an abstract state which is in scope to the property that automaton represents when determining a combined scope.

Furthermore we annotate the node of an abstract state e which is in scope with the function it is in and add a label *Automaton* if the *property scope reason* $\mathtt{autom}_\nabla$ is present in one of the associated *property scope locations* and a label *Formula* if $\mathtt{pred}_\nabla$ is present. The edge going into the node is labeled with the control-flow edge the abstract successor computation followed producing e.

4.2.2 *Property Scope Structure Graph*

Last we present the *property scope structure graph* PSSG which is a further reduction of the *property scope graph*. Figure 6 shows such graphs for our two examples. The graph is constructed by collapsing the

colored nodes of the PSG which consist the same set of automatons which are out of their initial state and have the same set of scope reasons attached. Uncolored nodes are removed.



(a) fopen_malloc_expl.c (b) fopen_malloc_noexpl.c

Figure 6: Property scope structure graphs for two example programs with the properties *malloc.spc* and *fopen.spc*

This is mostly interesting for a scope determination analysis with multiple properties at once to see in a compact way how the properties alternate inside the state space. For the accuracy of this Visualization the quality of the heuristic which associates a specific property to a specific part of the combined scope is essential. Especially this does not yield to useful results with pure weaving automatons as the presented approach lacks the ability to predict the encoded state.

```c
#include <stdio.h>
#include <stdlib.h>
int nondet() {int nd; return nd;}
char makechar(int a, char** txt) {
  char* text = *txt; int p = nondet();
  if(p > a) return text[(a-1) % p];
  return '0';
}
int handle_files(char* file1) {
  int c, x1; x1 = 5;
  FILE* fd1;
  char* text = "textblafoo";
  int alloc = 0; int open = 0;
  if(file1) {
    fd1 = fopen(file1, "r");
    if (fd1 == NULL) return 88;
    open = 1;
  }
  if(nondet()) {
    text = malloc(5 * sizeof(char));
    alloc = 1;
  }
  char r = makechar(x1, &text);
  if(r == '5') c = 3; else c = 4;
  if(alloc) free(text);
  if(open) {fclose(fd1); fd1 = NULL;}
  return c;
}
int main(int argc, char** argv) {
  if(argc != 2) return 66;
  return handle_files(argv[1]);
}
```

Listing 1: fopen_malloc_expl.c

```c
#include <stdio.h>
#include <stdlib.h>
int nondet() {int nd; return nd;}
char makechar(int a, char** txt) {
  char* text = *txt; int p = nondet();
  if(p > a) return text[(a-1) % p];
  return '0';
}
int handle_files(char* file1) {
  int c, x1; x1 = 5;
  FILE* fd1;
  char* text = "textblafoo";
  int alloc = 0; int open = 0;
  if(file1) {
    fd1 = fopen(file1, "r");
    if (fd1 == NULL) return 88;
    open = 1;
  }
  if(open) {fclose(fd1); fd1 = NULL;}
  if(nondet()) {
    text = malloc(5 * sizeof(char));
    alloc = 1;
  }
  char r = makechar(x1, &text);
  if(r == '5') c = 3; else c = 4;
  if(alloc) free(text);
  return c;
}
int main(int argc, char** argv) {
  if(argc != 2) return 66;
  return handle_files(argv[1]);
}
```

Listing 2: fopen_malloc_noexpl.c

```
OBSERVER AUTOMATON MallocFreeAutomaton
INITIAL STATE Init;
STATE USEFIRST Init :
 MATCH CALL {$? = malloc($1)} ->
  ASSUME {((int)$1) != 0} GOTO Alloc;
 MATCH CALL {free($1)} ->
  ASSUME {((void*)$1) != 0} ERROR;
STATE USEFIRST Alloc :
 MATCH CALL {free($1)} ->
  ASSUME {((void*)$1) != 0} GOTO Init;
 MATCH CALL {$? = malloc($1)} ->
  ASSUME {((int)$1) != 0} ERROR;
END AUTOMATON
```

Listing 3: malloc.spc

```
OBSERVER AUTOMATON FopenAutomaton
INITIAL STATE Init;
STATE USEFIRST Init :
 MATCH RETURN {$1 = fopen($?, $?)} ->
  ASSUME {((void*)$1) != 0} GOTO Opened;
 MATCH RETURN {$1 = fopen($?, $?)} ->
  ASSUME {((void*)$1) == 0} GOTO Init;
 MATCH CALL {fclose($1)} ->
  ASSUME {((void*)$1) != 0} ERROR;
 MATCH CALL {fclose($1)} ->
  ASSUME {((void*)$1) == 0} GOTO Init;
STATE USEFIRST Opened :
 MATCH RETURN {$1 = fopen($?, $?)} ->
  ASSUME {((void*)$1) != 0} ERROR;
 MATCH RETURN {$1 = fopen($?, $?)} ->
  ASSUME {((void*)$1) == 0} GOTO Opened;
 MATCH CALL {fclose($1)} ->
  ASSUME {((void*)$1) != 0} GOTO Init;
 MATCH CALL {fclose($1)} ->
  ASSUME {((void*)$1) == 0} GOTO Opened;
END AUTOMATON
```

Listing 4: fopen.spc

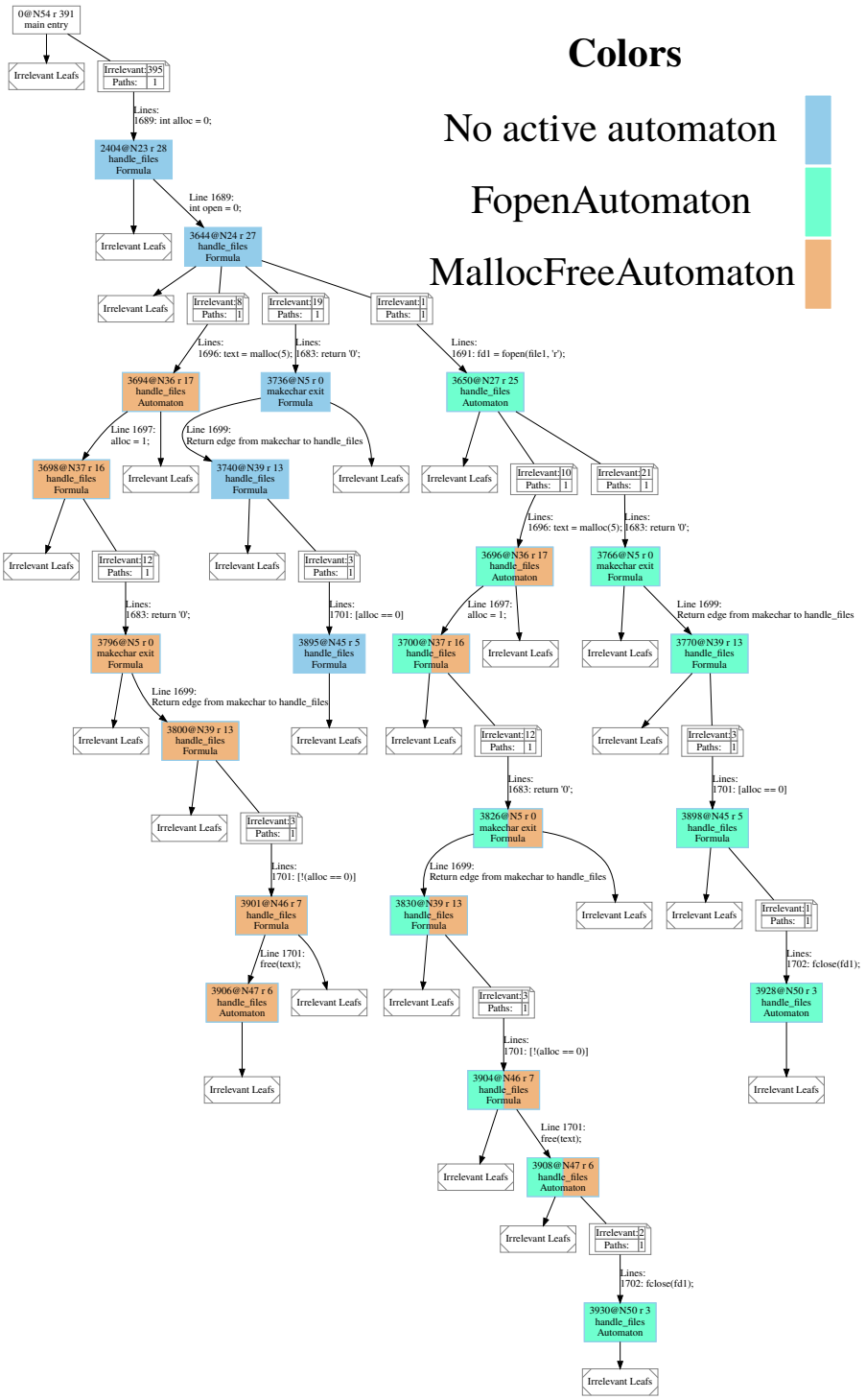


Figure 7: Property scope graph for *fopen_malloc_expl.c* with the properties *malloc.spc* and *fopen.spc*

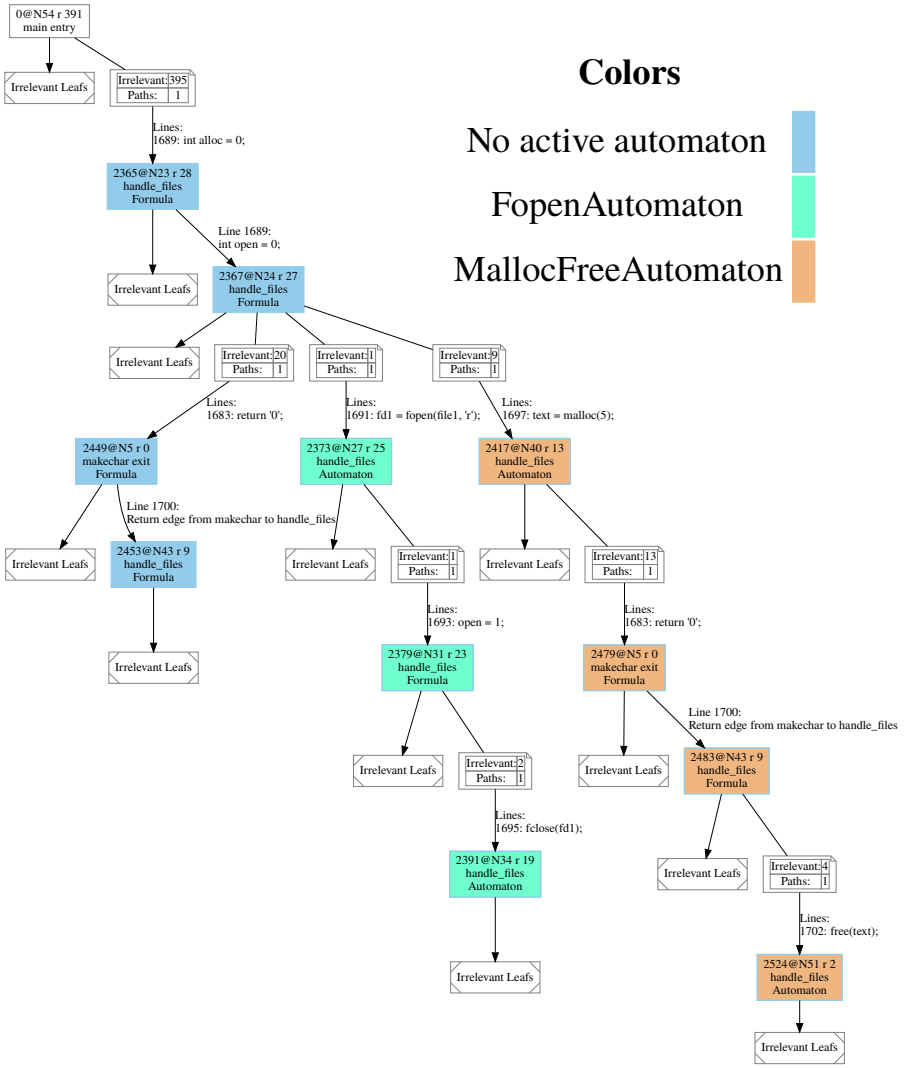


Figure 8: Property scope graph for *fopen_malloc_expl.c* with the properties *malloc.spc* and *fopen.spc*

5 EVALUATION

This chapter presents a series of experiments enabled by the approach developed in this work.

5.1 RESEARCH QUESTIONS

The evaluation is guided by 7 research questions which can be divided into two groups. The first group asses programs and associated properties which are part of various benchmark sets regarding the scope of the properties. The second group focuses on the impact of overlapping property scopes onto a multi-property analysis.

5.1.1 *Assessing Property Scopes of Benchmark Sets*

The following research questions form the basis to asses existing benchmark sets by various metrics regarding the scope of the respective properties.

5.1.1.1 *RQ1.1: Distribution of Property Scopes inside Programs*

We first want to know, how the scope of a property is distributed throughout programs. Is it concentrated inside a small part of a program or more widely spread throughout it? This leads us to the following concrete question: *How many functions are in scope of a property (i.e. at least one abstract state which is part of the scope Ω is located inside the respective function) in comparison to the functions of a program and the functions visited by the analysis?*

5.1.1.2 *RQ1.2: Program Variables inside the Property Scope*

Another interesting aspect is the number of variables of the program which are affected by a property, or more precise: *How many unique variables occur in abstraction formulas at abstraction states which have at least one property scope location attached compared to the number of variables relevant[1] for the analysis?*

5.1.1.3 *RQ1.3: Data Locality inside Property Scopes*

In addition to how much of the program is affected by a property it's also interesting if a property can be proven by using mostly local

1 There may be variables declared inside the program which are never used. CPAchecker computes through a simple dependency analysis a set of relevant variables whose size is used here as a baseline for comparison.

variables: *What percentage of the unique variables inside a function which occur in abstraction formulas at abstraction states with at least one property scope location are defined in the same function?*

5.1.1.4 *RQ1.4: Global Variables inside the Property Scope*

Finally we take the opposite perspective of RQ1.3 and look at how many global variables are involved in proving the property. *What percentage of unique variables which occur in abstraction formulas at abstraction states with at least one property scope location attached are global?*

5.1.2 *Property Scopes and Multi-Property Verification*

This group of research questions focuses on the impact the scope of properties has onto the behavior of an analysis which verifies multiple safety-properties together. According to Apel et al. [4] such a multi-property analysis is often more efficient than proving each property in a separate verification run. Though it is important to identify which subsets of properties should be verified together for a given program to yield the best possible performance overall and even avoid slowdowns in some cases. We want to explore if the knowledge about the scopes of properties can lead to a viable heuristic.

At the first time it would be necessary to verify every property separately and simultaneously collect the scope using the presented method. Subsequent analysis runs could benefit, e.g. for new versions of the analyzed program or new versions of the verification tool. Assuming that both usually evolve through gradual changes, the found partitioning of the property sets should in general still be good enough for some time.

5.1.2.1 *RQ2.1: Overlap of Property Scopes*

To what extent do the scopes of pairs of properties overlap?

5.1.2.2 *RQ2.2: Performance of Verifying Overlapping Properties Together*

Is there a correlation between the performance of a multi-property verification and the overlap of the scopes of the pair of properties which is verified together?

5.1.2.3 *RQ2.3: Reached Set Size with Overlapping Properties*

How does the size of the reached set change when multiple properties are verified together and does the reached set grow over proportion if the property scopes overlap?

5.2 SETUP

5.2.1 *Benchmark Suite*

The experiments are performed on two different benchmark sets. The first set which will be referred to as the *SV-COMP* set is a specific subset taken from the benchmark suite of the *Competition on Software Verification*[2]. It contains 3486 programs in total, divided into the categories *ReachSafety-ControlFlow* (94 programs), *Systems_DeviceDriversLinux64_ReachSafety* (2795 Linux kernel modules from various versions) and *ReachSafety-ProductLines* (597 programs of 3 product-lines originally originating from SPLVERIFIER [3][2, 3]).

These programs are designed to serve as a common benchmark for multiple software verification tools which implement different specification languages. As a result of that they contain safety properties which are weaved into their source code so the verification tool has to check for the reachability of error locations to prove them. The programs inside *ReachSafety-ProductLines* represent an interesting special case as there exists set of different specifications for each of the product lines, so these programs each represent a specific product of the respective line with a specific specification of the line weaved in.

The second benchmark set which will be referred to as the *LDV-250* set consists of 250 randomly chosen Linux 4.0-rc1 kernel modules (out of 4336) which is the same as described in [4]. In contrast to the *SV-COMP* set there are no safety properties weaved into the code but instead the set is accompanied by a set of 14 properties encoded as specification automata. Table 1 describes them in detail.

5.2.2 *Experiments*

As a foundation for all experiments the scope of every property and every program present inside both benchmark sets is determined by running the property scope analysis. This results in 3500 benchmark tasks for the LDV-250 set and 3486 tasks for the SV-COMP set.

5.2.2.1 *RQ1.x: Assessing Property Scopes of Benchmark Sets*

The research questions RQ1.x are answered by examining the ARGs of these analysis runs. We generally only look at the 1654 results for SV-COMP and 3359 for LDV-250 where the analysis has been successfully finished and has proven the program to be safe. Unsafe results are excluded because incomplete results are very likely which would distort the statistics as explained in section 3.1.6. In addition to that

2 The full benchmark suite can be found online at `https://github.com/sosy-lab/sv-benchmarks`

3 SPLVERIFIER can be found online at `http://www.infosun.fim.uni-passau.de/spl/FAV/`

Property	Description
08a	Each module that was referenced with *module_get* must be released by *module_put*.
10a	Each memory allocation that gets performed in the context of an interrupt must use the flag *GFP_ATOMIC*.
32a	The same mutex must not be acquired or released twice in the same process.
43a	Each memory allocation must use the flag *GFP_ATOMIC* if a spinlock is held.
68a	All resources that were allocated with *usb_alloc_urb* must be released with usb_free_urb.
68b	Each DMA-consistent buffer that was allocated with *usb_alloc_coherent* must be released by calling *usb_free_coherent*.
77a	Each memory allocation in a code region with an active mutex must be peformed with the flag GFP_NOIO.
101a	All structs that were obtained with blk_make_request must be released by calling *blk_put_request* afterwards.
106a	The modules gadget, char, and class that were registered with *usb_gadget_probe_driver*, *register_chrdev*, and *class_register* must be unregistered by calling *usb_gadget_unregister_driver*, *unregister_chrdev* and *class_unregister* correspondingly in reverse order of the registration.
118a	Reader-writer spinlocks must be used in the correct order.
129a	An offset argument of a find_bit function must not be greater than the size of the corresponding array.
132a	Each device that was allocated by by *usb_get_dev* must be released with *usb_put_dev*.
134a	The probe functions must return a non-zero value in case of a failed call to *register_netdev* or *usb_register*.
147a	RCU pointer/list update operations must not be used inside RCU read-side critical sections.

Table 1: Safety properties for the *LDV-250* set (taken from [4])

for RQ1.1 we only look at 482 results for LDV-250 and 1263 results for SV-COMP where the property is relevant. For RQ1.2, RQ1.3 and RQ1.4 the result set is restricted to results where the specification automaton once reached a target state so at least one predicate abstraction refinement was made (without every abstraction formula is `true`), this are 171 results for LDV-250 and 1268 results for SV-COMP.

5.2.2.2 *RQ2.x Property Scopes and Multi-Property Verification*

To answer RQ2.x, a multi-property analysis as described above is used. First a subset of 186 programs from the LDV-250 set is assembled which were proven safe during the run of the property scope analysis and where still at least two properties are relevant. This results in 437 benchmark tasks where for each program one property of the remaining relevant and non-violating properties is verified. This is a single property verification but the same multi-property analysis

configuration is used so this can serve as a baseline which we compare to.

We build a second set of 324 benchmark tasks, one for every program and every possible pair of the remaining relevant and non-violating properties for that program as described above. These are then verified together while forcing the multi-property analysis to not decompose them but verify them in one single run of the predicate analysis.

Based on the property scope call-graph PSC as presented in section 4.1.1 we calculate a percentage of how much the scopes of the pairs of properties for each of the 186 programs overlap. For a pair of properties p_1, p_2, the set F of functions of the program, and the respective numbers of functions $n_1^{visited}, n_2^{visited}$ visited during the respective runs of the property scope analysis the following is calculated:

$$overlap^{\%} = \frac{\left|\left\{f \in F \middle| imp_{f_1}^{ps} > 0 \wedge imp_{f_2}^{ps} > 0\right\}\right|}{0.5 \cdot (n_1^{visited} + n_2^{visited})} \cdot 100$$

5.2.3 *Analysis Domain*

To determine the scopes of properties a composite analysis which observers the predicate analysis and the specification analysis as described in section 3.1 is configured. The predicate analysis is configured to use blk_{SBE} to get the most accurate results possible. We also add the *coverage CPA* and configure it in a mode where it extracts information also from all refinement steps which gives us the number of functions visited during the whole analysis and not only those which are part of the final reached set. For the LDV-250 set an existing set of several configuration options (*ldv-adjustments*) which adjust CPAchecker to peculiarities of that set compared to the default configuration is more suited for the SV-COMP set is used in addition.

For the multi-property analysis the existing configuration based on what is described in the literature [4] is used. It is configured to use the *AllThenNone* decomposition operator which effectively disables the specification decomposition heuristics and always analyzes all properties simultaneously.

5.2.4 *Benchmarking Environment*

All experiments are run on a fork *odysseus* of CPAchecker at git revision `9af1e5a9ac` where the approach described here is implemented. CPAchecker is executed using Java 8 on machines running Linux. The predicate analysis uses SMTInterpol version `2.1-238-g1f06d6a-comp` as its underlying SMT solver.

For benchmark tasks where we determine the scope of a property the resource limits are set following other typical setups for a predicate analysis for the respective benchmark set as the property scope CPA should not add a considerable overhead to that. For the set *SV-COMP* the process is limited to 11 GB of Java heap memory, 15 GB of overall memory, 4 CPU cores, a soft runtime limit of 1300 s and a hard runtime limit of 1600 s. Processes of the set *LDV-250* are limited by 26 GB of Java heap memory, 30 GB of overall memory, 4 CPU cores, a soft runtime limit of 12 660 s and a hard runtime limit of 14 000 s. For this type of analysis runs we are not interested in measurements of any time or memory usage but only about the completion of a large enough amount of analysis runs so we do not need to limit this to a specific machine. Machines using The following CPU models equipped with enough memory and CPU cores for the respective task were used: Intel Core i7-{2600, 4770, 6700} and Intel Xeon {E3-1230 v5, E5-2650 v2}

The benchmark tasks which are based on the multi-property analysis configuration, where we measure time are run inside an environment with better comparability. A cluster of identically built machines with 135 GB RAM and a 32 core Intel Xeon E5-2650 v2 CPU is used. The resources for a process executing a single task are limited by 26 GB of Java heap memory, 30 GB of overall memory, 4 CPU cores, a soft runtime limit of 12 660 s and a hard runtime limit of 14 000 s.

5.2.5 *Presentation*

The results are graphically presented in histograms, scatter plots and violin plots with quantile strokes at 25 %, 50 % and 75 %. Numbers in general are rounded to one significant digit.

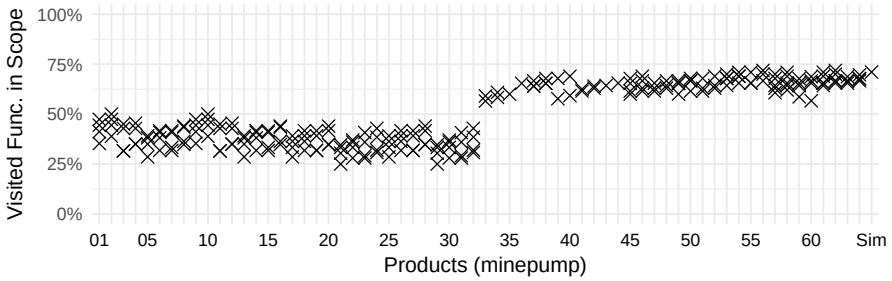
5.3 RESULTS

5.3.1 *RQ1.1: Distribution of Property Scopes inside Programs*

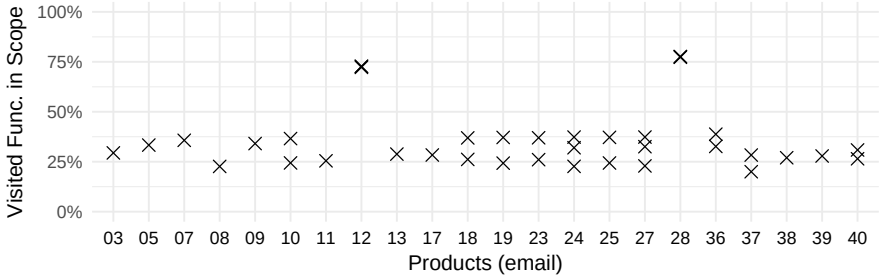
First we look at the *SV-COMP* set. In Figure 10a we see a huge difference in the percentage of functions in scope of the respective property weaved into the program by category of the benchmark set. At most programs inside the *Systems_DeviceDriversLinux64_ReachSafety* category the number of visited functions which are part of the scope stays most of the time below the 25 % mark with a median value at 7.7 % where in contrast the programs in *ReachSafety-ControlFlow* show the opposite behavior with a median value of 85.7 %.

The category *ReachSafety-ProductLines* which lies very much in between these extremes deserve a closer look. First we only see results of the *email* and the *minepump* product-line, the third one (*elevator*) timeouts in all cases with our configuration. As clearly visible in Fig-

ure 10b the percentage of visited functions in the scope of the property for the programs inside the *minepump* product-line falls into two clusters, one between 25 % and 50 %, the other between 50 % and 75 %. It does not look like the specification has a considerable influence but as we see in Figure 9a it is dependent on the product. Products below 33 fall inside the first cluster and those above and including 33 fall inside the second cluster. The product-line *email* shows a similar behavior although we have lesser data points available here due to timeouts. Figure 9b shows a significantly elevated value only for two specific products (12 and 28).



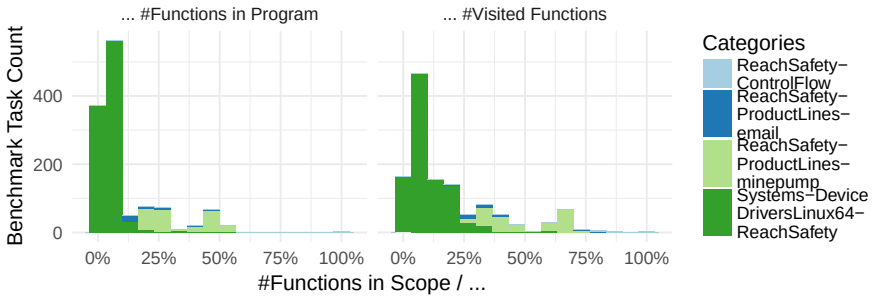
(a) the *minepump* product line



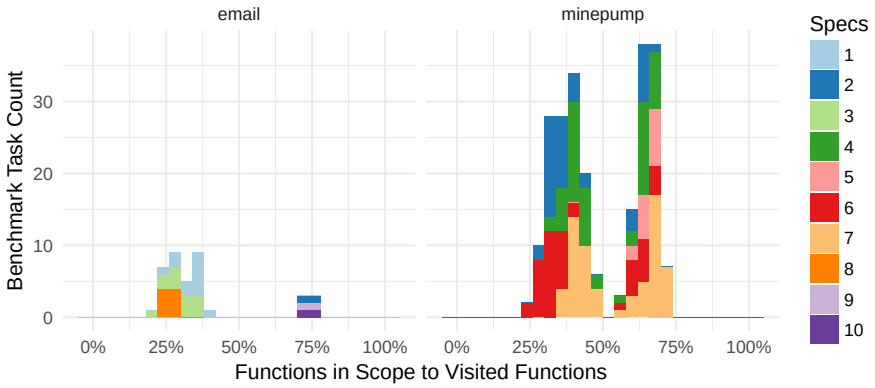
(b) the *email* product line

Figure 9: Number of functions which are part of the scope of the property in relation to the number of functions visited at least once during the whole runtime of the analysis for the product lines inside the SV-COMP set differentiated by product

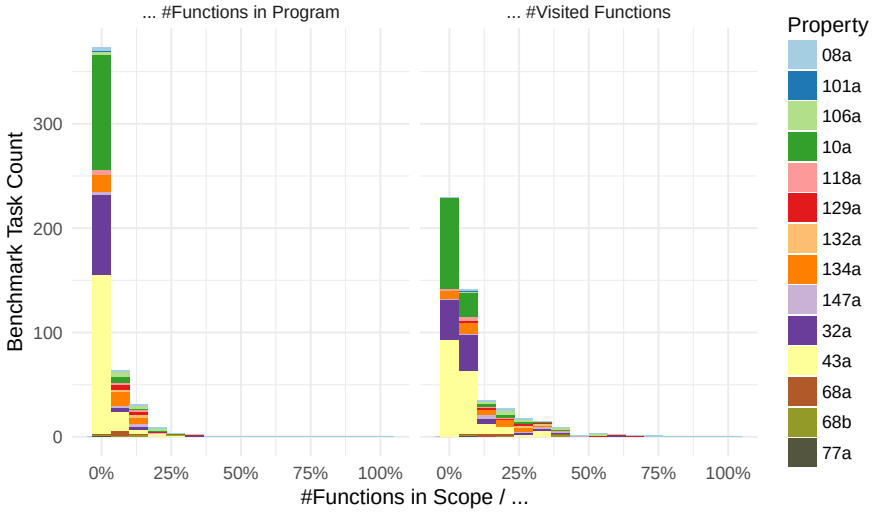
The results for the *LDV-250* set are displayed in Figure 10c. We overall see a very low number of affected functions compared to visited functions with a median value of just around 3.6 %. Although we see a very huge spread with a minimum value of only 0.3 % with property *10a* and a maximum of 71.4 % with property *08a*. The median values for the different properties in isolation range from 2.3 % (*10a*) over 3.2 % (43a), 3.6 % (32a), 4.9 % (77a), 7.7 % (101a), 7.8 % (118a), 9.0 % (134a), 13.7 % (68a), 20.5 % (106a), 21.0 % (08a), 24.9 % (129a), 27.8 % (147a) and 30.2 % (132a) up to 30.7 % (*68b*).



(a) SV-COMP set, 1263 of 3486 tasks total



(b) The product line category of the SV-COMP set



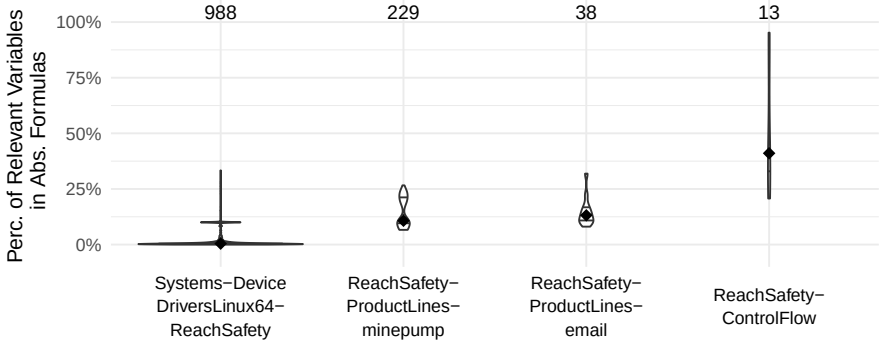
(c) LDV-250 set, 482 of 3500 tasks total

Figure 10: Number of functions part of the scope of the property, both in relation to the total number of functions which are defined inside the verified program and the number of functions visited at least once during the whole runtime of the analysis

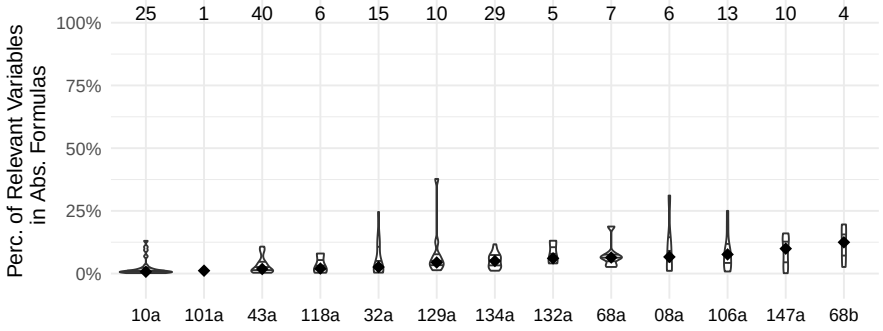
5.3.2 *RQ1.2: Program Variables inside the Property Scope*

The violin plot in Figure 11a shows that the percentage of relevant variables which are part of the scope of the property inside the *Systems_DeviceDriversLinux64_ReachSafety* category of the *SV-COMP* set is very small. The median value is only at 0.4 % with some outliers concentrated at around the 10 % mark. For the programs inside the *ReachSafety-ProductLines* category we see median values of 10.7 % (*minepump*) and 13.2 % (*email*) with a low variation. Especially with the *minepump* product-line we see a split into two clusters, very similar to the observations in section 5.3.1 . The category with the highest percentage of relevant variables in scope is *ReachSafety-ControlFlow* with a median value of 41.0 %, highly scattered between 20.7 % and 95.2 %.

Figure 11b shows that inside the LDV-250 set the percentage of variables in scope is at the lower area in general, ranging from 0.2 % (147a) to 37.7 % (129a). Nevertheless there are still some visible differences between the properties: the median values for the individual properties are scattered between 0.7 % (10a) and 12.5 % (68b).



(a) SV-COMP: Differentiated by Category



(b) LDV-250: Differentiated by Property

Figure 11: Number of unique variables which occur in abstraction formulas at abstraction states which are part of the scope of the property compared to the number of variables relevant for the analysis; the diamond indicates the median value; the number of programs is shown on top

5.3.3 *RQ1.3: Data Locality inside Property Scopes*

When looking at Figure 12a we see that the percentage of variables in abstraction formulas which are defined in the same function where the abstraction state is located varies greatly between the categories of the SV-COMP set. The scopes of the vast majority of benchmark tasks inside the category *Systems_DeviceDriversLinux64_ReachSafety* are not very local at all by this metric, except a very small number of outliers the value is very close to 0 %. The category *ReachSafety-ControlFlow* shows more locality with a median value of 21.3 % and scattered between 12.3 % and 100 %. Looking at *ReachSafety-ProductLines* we see a huge difference between the *minepump* product line with a median value of 26.5 % and the *email* product line with a median value of 79.1 %. Inside the *minepump* product line the values do not vary much. For most cases inside the *email* product line the values stay close to the median but we see a small number of benchmark tasks centering around 30 %.

The LDV-250 set shows great variation of locality depending on property as shown in Figure 12b . The median values range from 40.1 % (147a) to 100 % (10a), although in an overall view the scopes are far more local than not with a median over all tasks of 84.8 % with 147a and 132a being notable exceptions.

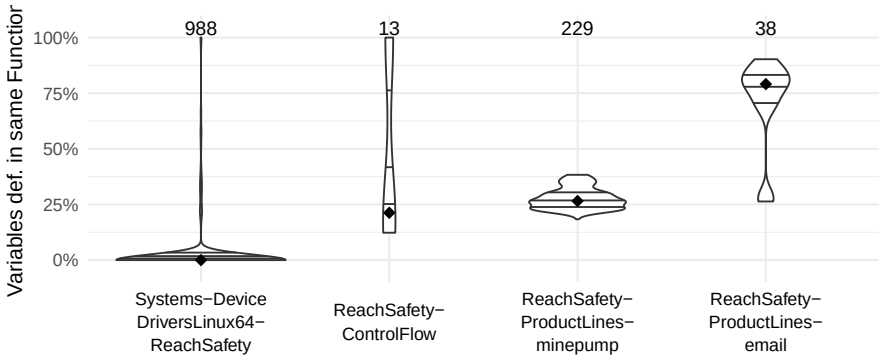
5.3.4 *RQ1.4: Global Variables inside the Property Scope*

Looking at Figure 13 we unsurprisingly notice that we tend to see more global variables where less variables are local to the function but it's not the clear opposite in most cases. The properties of LDV-250 generally do not involve a lot of global variables with a overall median for all tasks at 1.9 % with 147a with a median value of 15.2 % as the most noticeable exception. Most of those may furthermore be variables weaved into by the specification at runtime as those appear like global variables inside the formula.

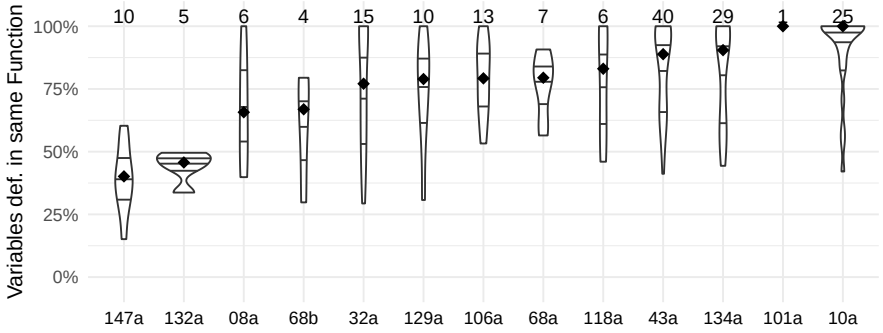
The SV-COMP category *Systems_DeviceDriversLinux64_ReachSafety* sticks out in property scopes being not only not very local but tracking only global variables with negligible exceptions. The *ReachSafety-ProductLines* category differs between the *email* and *minepump* product-lines. Where inside *email* close to 0 % of variables are global in *minepump* the median is 36.4 %. At *ReachSafety-ControlFlow* the median is at 21.7 % but the values for the individual tasks also relatively scattered.

5.3.5 *RQ2.1: Overlap of Property Scopes*

Only in 86 out of 324 cases there is at least one function which is classified as in scope of the property which is common for both prop-



(a) SV-COMP: Differentiated by Category



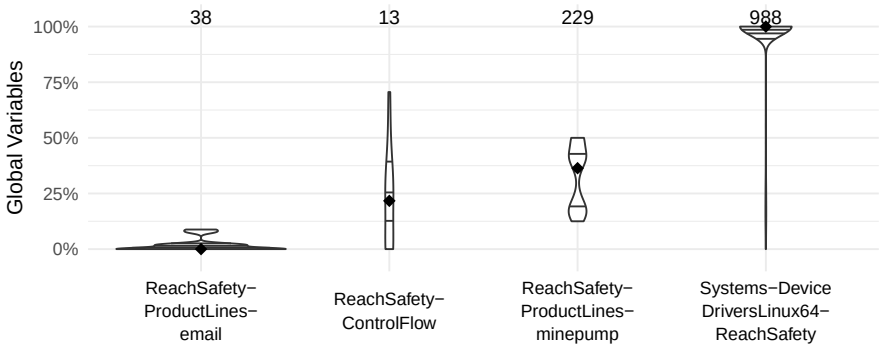
(b) LDV-250: Differentiated by Property

Figure 12: Percentage of unique variables inside a function which occur in abstraction formulas at abstraction states which are part of the scope of the property and are defined in the same function (for each program the mean value over all qualified functions is taken); the diamond indicates the median value; the number of programs is shown on top

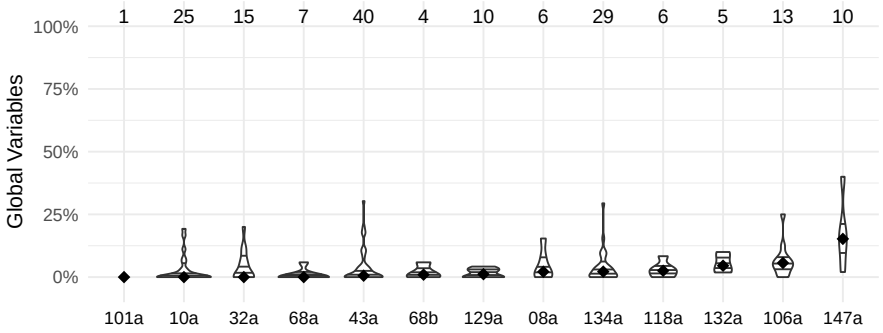
erties. Looking closer only at those cases, we can see in e.g. Figure 14 that at maximum 47.2 % of the functions visited by the analysis are part of the scope of both properties and that most of these overlaps are quite small with a median value of 2.9 %.

5.3.6 *RQ2.2: Performance of Verifying Overlapping Properties Together*

As we can see clearly in Figure 14 there seems to be no direct correlation between the speedup regarding the CPU time for analysis of a multi-property verification and the percentage of visited functions which are in the scope of both properties. It's especially visible that when verifying properties with a considerable scope overlap together the analysis is in most cases still considerably faster and their speedup stays, except rare outliers, inside the same range of those with no overlap.



(a) SV-COMP: Differentiated by Category



(b) LDV-250: Differentiated by Property

Figure 13: Percentage of unique variables which occur in abstraction formulas at abstraction states which are part of the scope of the property which are global; the diamond indicates the median value; the number of programs is shown on top

5.3.7 *RQ2.3: Reached Set Size with Overlapping Properties*

Looking at Figure 15 and Table 2 we see that the set *reached* is most of the time dominated by one of the the two properties verified together or very often the same size. It more or less never shrinks below the size of the set *reached* of the reached set produced in single verifications of either property. Only in 19 of the 324 cases the set reached grows to more than 100.1 % of the larger set of the runs with single properties. Moreover a direct correlation between the overlap of property scopes and the size of the set reached is not visible. If we see the size of the set reached as a rough metric for the difficulty of a verification run this agrees with the result of RQ2.2: a state space explosion rarely occurs at a considerable scale and is not correlated to the overlap of the scopes of the properties.

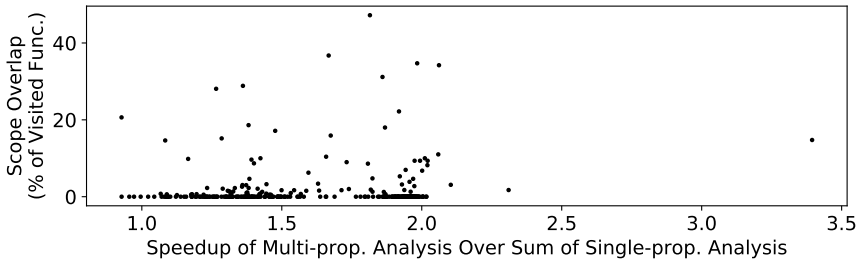


Figure 14: Scatter plot which shows the relationship between the speedup of a multi-property analysis and the overlap of property scopes (calculated per function) for pairs of properties; for the speedup only the cputime of the analysis itself is compared to exclude distorting effects resulting from startup or tear down of CPAchecker; only verification runs where the verification result is true and where both properties are relevant are shown

5.4 DISCUSSION

5.4.1 *RQ1.x: Assessing Property Scopes of Benchmark Sets*

Looking at the SV-COMP category *Systems_DeviceDriversLinux64_-ReachSafety* despite being one of the larger parts of the benchmark set the set of properties used does not seem to be very diverse regarding the shape of their scope. Their scopes seem to be in general very narrow, covering only a smaller part of the program and only very few of the programs variables. Those variables are in most cases global which may be due to the fact that these programs make heavy use of globally defined structs.

The properties used inside the *ReachSafety-ControlFlow* category on the other hand seem to be overall more balanced in their dependence on local and global variables and more diverse in how much of the programs variables they cover. The very high percentage of functions in scope can be explained by the simple fact that many of these programs only consist of a hand full of functions so this metric which assumes a reasonably uniform distribution of code into functions does not fit very well in this case.

We take a closer look at the clearly noticeable clustering of the results around two values inside the *minepump* product-line which is especially visible in the percentage of functions which are in scope of the property. Of the 5 specifications which are used in this product line, all but specification 3 can explicitly only be violated if the pump is active. The pump only ever gets activated by the feature *highWaterSensor* which is only active on product 33 and above, exactly where we see the property scope covering vastly more functions. Specification 3 however which may also be violated if the pump is inactive is

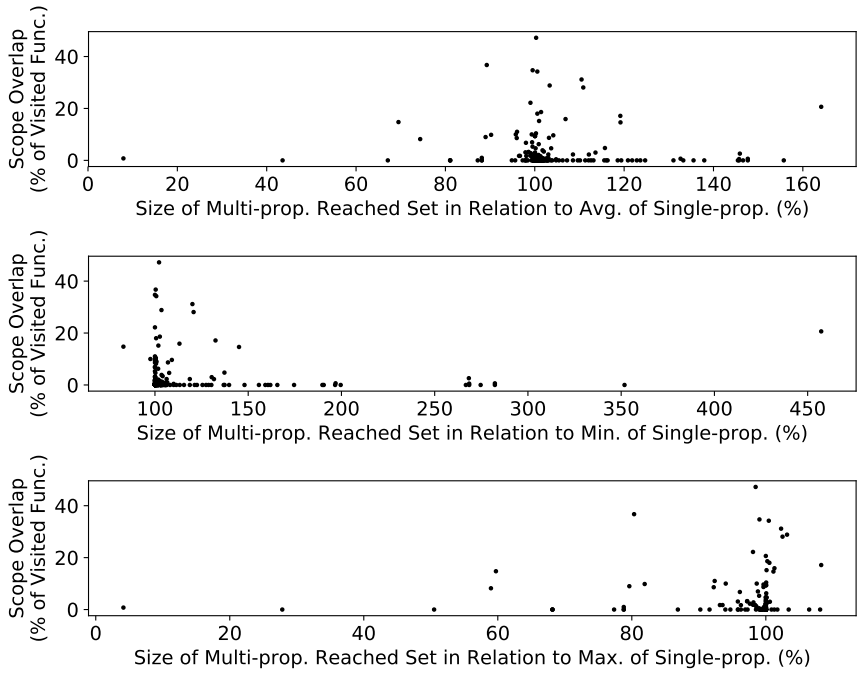


Figure 15: Scatter plots which show the relationship between the growth of the final reached set of a multi-property analysis and the overlap of property scopes (calculated per function) for pairs of properties; only verification runs where the verification result is safe and where both properties are relevant are shown

violated in cases the pump never gets started. As we exclude unsafe verification results we never see the impact of it at products below 33. The similar observation made inside the *email* product line is likely caused by one or more specifications which are only weaved into a program if the *Encrypt* feature is active on the product the program represents. Of our results this is the case only in product 12 and 28 where we see an increased percentage of functions in scope of the property.

Looking at the results for the LDV-250 set we can clearly see that the scopes of the individual properties are shaped differently. As one example properties 10a and 43a which ensure some flag is set to a specific value on specific function calls which occur after entering a specific execution context involve a small amount of variables where many are local and the scope covers a small number of functions. The flag itself is most of the time set to a fixed value shortly before the function which results in a very local property scope. If the flag is set to the desired value in any case it does not enlarge the scope significantly, otherwise it must be proven that the program is not inside the violating context which seems not to depend on a lot of conditions inside these programs for these contexts.

↱	08a	106a	10a	118a	129a	132a	134a	147a	32a	43a	68a	68b
08a		...	...	100	...	...	0	33	100	67	...	...
106a	...		100	...	0	...	...	...	86	75	...	...
10a	...	0		100	14	...	0	0	94	75	...	...
118a	0	...	100		...	...	...	25	...	43	...	...
129a	...	0	100	...		...	0	...	100	80	...	...
132a	...	...	...	...	...		0	...	100	67	0	...
134a	0	...	73	...	0	0		0	100	62	...	...
147a	33	...	100	100	...	...	0		100	80	...	...
32a	100	0	94	...	40	0	0	0		71	50	50
43a	33	0	96	57	0	0	0	0	90		0	0
68a	...	...	...	...	...	0	...	...	100	100		33
68b	...	...	...	...	...	...	...	...	50	100	33	

Table 2: Rows show the percentage of cases (programs where the pair of properties is relevant and the verification result is safe) where the final reached set of a single property verification contains the same number of states as the final reached set of a multi-property verification with the property defining the column in addition.

Another example are 68a and 68b which involve a relatively high number of variables and cover a comparatively high number of functions. This could be due to the fact that there often is a lot of code between a call to *usb_alloc** and the corresponding call to *usb_free** where a lot assumptions in different functions involving different variables must be true to prove that an *usb_free** call is reached in every corner case.

5.4.2 *RQ2.x: Property Scopes and Multi-Property Verification*

The hypothesis that overlapping property scopes can correlate with a negative performance impact of a multi-property analysis is based on the fact that the abstract state space is unable to merge when the control flow meets. The termination check is also unable to recognize coverage if one of the specification automatons changes its state. If such events happen successively, the state space splits again which in a worst case scenario may lead into near exponential explosion of the state space. This can also happen if the state of the automaton is weaved into the analysis as the state is then encoded into a variable which often leads to different abstraction formulas which also hinders the predicate analysis with ABE to merge or report coverage. Listing 1 (chapter 4) shows a program *fopen_malloc_expl.c* which illustrates such behavior when running a predicate analysis with ABE with the properties *malloc.spc* shown in listing 3 and *fopen.spc* shown in listing

4. After the branching of the control flow in line 14 *fopen.spc* changes its state to *Opened* in only one branch 15 which hinders the states to merge or to cover. At line 20 *malloc.spc* transitions in a similar fashion as above to *Alloc* which leads the state space to split again at both of the previous branches as illustrated in Figure 16.

Figure 16: Simplified ARG (reduced to abstraction states) for verification of *fopen_malloc_expl.c* (left) and *fopen_malloc_noexpl.c* (right) with properties *fopen.spc* and *malloc.spc*

Both properties obviously meet our function based scope overlapping criterion in *fopen_malloc_expl.c* but this is also the case in the slightly modified program *fopen_malloc_noexpl.c* shown in listing 2. Here *fopen.spc* returns into its initial state before *malloc.spc* transitions into *Alloc*. As we see in Figure 16 the termination check is now able to report coverage before *malloc.spc* transitions into *Alloc* resulting in a similar behavior as if the properties where analyzed separately and a lot fewer states than in the previous example.

We see that overlapping property scopes can theoretically increase the chance of a state space explosion but even then it only happens to a noticeable extent in rather special cases and is more related to state changes of the specification automaton.

Furthermore other effects may tone down the explosion of the state space as well as the negative impact on verification performance in those cases where the reached set actually grows:

- The at least partial usage of on-the-fly property weaving by the specification automatons increases the likelihood of a successful merge or a successful coverage detection due as the richer abstraction techniques of the predicate analysis.
- A state space explosion might occur in a very deep branch of the ARG and only produce a moderate number of additional abstract states

- The performance penalty of very similar abstraction computations may be very small due to various caching mechanisms employed by the predicate analysis and the underlying SMT-solver.

6 FUTURE WORK

In section 3.1.6 a limitation of the property scope CPA is mentioned which can lead to incomplete property scopes if a property violation is found because the predicate analysis ends in this case. As one possible solution it needs to be investigated if and how the predicate analysis can be modified to continue after a violation is found.

Another task for future improvements is a better attribution of an abstract state which is part of the combined scope of all properties inside a multi-property scope analysis to one specific property. As mentioned in section 4.2.2 the implementation only guesses right now by a specification automaton being out of its initial state i.e. *enabled* or if it matches at a specific abstraction state, which has several problems. First this does not work with pure weaving automata which encode their state into the predicate analysis. It may be possible to extract this implicit state from the formulas by looking at the weaved in state variables of the automaton. Second this is imprecise in both directions as even if the automaton is enabled we can't reliably attribute an abstract state to its scope, we only assume it to be likely. One possibly better approximation could be the usage of static program slicing to rule out states whose incoming control-flow edge has no control or data dependency relation to the program locations where the automaton transitions into a target state. Another possibility to explore is to look more closely to the individual steps of the predicate refinement process and record which property is responsible for a specific predicate in the precision. One could then try to rule out a specific abstraction state for a given property using the predicates attributed to that property and the abstraction formula.

As greatly discussed in section 5.4.2 we were not able to find a connection between state space explosions or other performance impacts onto a multi-property verification and the overlap $\mathtt{overlap}_{\%}$ of property scopes on a per function level, defined in section 5.2.2.2. At least for our examples *fopen_malloc_expl.c* and *fopen_malloc_noexpl.c* even a more fine grained look at statement level would be insufficient. It seems that at least for the example the overlap of the parts of the program where the respective property is *enabled* is crucial. Future work could try to improve on the detection of when a property is *enabled* as stated above and reevaluate RQ1.x using a notion of overlap tailored to that.

7 RELATED WORK

Because property scopes and the presented approach have a lot of resemblance with program slicing we put a focus on related work on that topic.

7.1 CONSTRUCTION OF PROGRAM SLICES

Several approaches use dynamic program slicing as a debugging aid [1, 20, 23] to help the programmer with localizing faulty program statements. This is done by slicing backwards from the location where the error appears, either where a wrong output is received from the program or a crash occurs. This has some resemblance with property scopes but limited to concrete executions.

Comuzzi and Hart [13] present an approach to perform program slicing using logical predicates and by computing weakest preconditions which shows some similarities to the refinement techniques of the predicate analysis described in section 2.3.5 we rely on for scope determination.

7.2 ASSESSMENT OF PROGRAM SLICES

Related to the assessment of property scopes we look at some approaches which perform measurements on program slices and compare several program slices with each other.

Binkley, Gold, and Harman [10] study all static forward and backward slices of a set of C programs by measuring the slice size using different configurations of a slicing tool involving, amongst other things the granularity of the slicing (statements vs. functions) and the inclusion of the calling context.

Meyers and Binkley [21] use program slices to measure the cohesion inside of software modules. They develop various cohesion metrics based on slices including but not limited to overlap (common statements in slices), coverage (length of slices compared to the length of the modules) and parallelism (the number of slices which have more than en number of statements in common). Bieman and Ott [9] measure cohesion on a per procedure level by computing *data slices* at the output of a procedure. A *data slice* is built out of of a combined forward and backward program slice which consists of variable and constant definitions and references called *data tokens*. The *data slice* is then further abstracted and compared.

7.3 PROPERTY SPECIFICATION PATTERNS

Dwyer, Avrunin, and Corbett [15] describe observations about patterns which temporal safety properties commonly adhere to. They divide them into *Order* patterns which talk about the relative order events must occur during execution and *Occurrence* patterns which talk about if an event occurs during execution. Patterns also have a scope where they are enabled determined by a start and a end event.

8 CONCLUSION

First we presented an approximative technique to determine the scope of safety properties inside the abstract state space of a software model checker by observing an existing analysis based on predicate abstraction.

Second we developed valuable ways to visualize property scopes by reducing the the visual representation of the typically extensive state space, providing an overview about the coarse structure of the scopes inside the state space of a multi-property verification and explain how to display the distribution of property scopes inside the program as well as their overlap points in a clear and compact way using arc diagrams.

Third our experimental assessment offers valuable insights into the data coverage, data locality and distribution of property scopes inside the programs of two substantial benchmark sets. Furthermore we found out through experimental evaluation that there exists no direct correlation between overlapping property scopes an the performance on multi-property.

Forth we proposed interesting opportunities for future work especially focusing more on the areas of the state space where properties are enabled.

BIBLIOGRAPHY

[1] Hiralal Agrawal, Richard A. Demillo, and Eugene H. Spafford. "Debugging with dynamic slicing and backtracking". In: *Software: Practice and Experience* 23.6 (1993), pp. 589–616. ISSN: 1097-024X. DOI: 10.1002/spe.4380230603. URL: http://dx.doi.org/10.1002/spe.4380230603.

[2] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. "Detection of Feature Interactions Using Feature-aware Verification". In: *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. ASE '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–375. ISBN: 978-1-4577-1638-6. DOI: 10.1109/ASE.2011.6100075. URL: http://dx.doi.org/10.1109/ASE.2011.6100075.

[3] Sven Apel, Hendrik Speidel, Philipp Wendler, Alexander von Rhein, and Dirk Beyer. *Feature-Aware Verification*. Tech. rep. MIP-1105. University of Passau, Germany, 2011. eprint: arXiv:1110.0021. URL: https://arxiv.org/abs/1110.0021.

[4] Sven Apel, Dirk Beyer, Vitaly Mordan, Vadim Mutilin, and Andreas Stahlbauer. "On-the-fly decomposition of specifications in software model checking". In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM. 2016, pp. 349–361.

[5] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis". In: *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007, Berlin, July 3-7)*. Ed. by W. Damm and H. Hermanns. LNCS 4590. Springer-Verlag, Heidelberg, 2007, pp. 504–518. ISBN: 978-3-540-73367-6.

[6] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. "Program Analysis with Dynamic Precision Adjustment". In: *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008, L'Aquila, September 15-19)*. IEEE Computer Society Press, Los Alamitos (CA), 2008, pp. 29–38. ISBN: 978-1-4244-2187-9.

[7] Dirk Beyer and M. Erkan Keremoglu. "CPAchecker: A Tool for Configurable Software Verification". In: *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Berlin, Heidelberg: Springer Berlin Heidelberg,

2011, pp. 184–190. ISBN: 978-3-642-22110-1. DOI: 10.1007/978-3-642-22110-1_16. URL: http://dx.doi.org/10.1007/978-3-642-22110-1_16.

[8] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. "Predicate Abstraction with Adjustable-Block Encoding". In: *Proceedings of the 10th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2010, Lugano, October 20-23)*. FMCAD, 2010, pp. 189–197. URL: http://www.sosy-lab.org/~dbeyer/cpa-abe/.

[9] James M Bieman and Linda M Ott. "Measuring functional cohesion". In: *IEEE transactions on Software Engineering* 20.8 (1994), pp. 644–657.

[10] David Binkley, Nicolas Gold, and Mark Harman. "An Empirical Study of Static Program Slice Size". In: *ACM Trans. Softw. Eng. Methodol.* 16.2 (Apr. 2007). ISSN: 1049-331X. DOI: 10.1145/1217295.1217297. URL: http://doi.acm.org/10.1145/1217295.1217297.

[11] David Binkley, Sebastian Danicic, Tibor Gyimóthy, Mark Harman, Akos Kiss, and Lahcen Ouarbya. "Formalizing executable dynamic and forward slicing". In: *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*. IEEE. 2004, pp. 43–52.

[12] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. "Counterexample-guided abstraction refinement for symbolic model checking". In: *Journal of the ACM (JACM)* 50.5 (2003), pp. 752–794.

[13] Joseph J. Comuzzi and Johnson M. Hart. "Program slicing using weakest preconditions". In: *FME'96: Industrial Benefit and Advances in Formal Methods: Third International Symposium of Formal Methods Europe Co-Sponsored by IFIP WG 14.3 Oxford, UK, March 18–22, 1996 Proceedings*. Ed. by Marie-Claude Gaudel and James Woodcock. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 557–575. ISBN: 978-3-540-49749-3. DOI: 10.1007/3-540-60973-3_107. URL: http://dx.doi.org/10.1007/3-540-60973-3_107.

[14] William Craig. "Linear reasoning. A new form of the Herbrand-Gentzen theorem". In: *The Journal of Symbolic Logic* 22.03 (1957), pp. 250–268.

[15] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. "Patterns in Property Specifications for Finite-state Verification". In: *Proceedings of the 21st International Conference on Software Engineering*. ICSE '99. Los Angeles, California, USA: ACM, 1999, pp. 411–420. ISBN: 1-58113-074-0. DOI: 10.1145/302405.302672. URL: http://doi.acm.org/10.1145/302405.302672.

[16] Mark Harman and Robert Hierons. "An overview of program slicing". In: *Software Focus* 2.3 (2001), pp. 85–92. ISSN: 1529-7950. DOI: 10.1002/swf.41. URL: http://dx.doi.org/10.1002/swf.41.

[17] Jeffrey Heer, Michael Bostock, and Vadim Ogievetsky. "A Tour Through the Visualization Zoo". In: *Queue* 8.5 (May 2010), 20:20–20:30. ISSN: 1542-7730. DOI: 10.1145/1794514.1805128. URL: http://doi.acm.org/10.1145/1794514.1805128.

[18] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. "Lazy Abstraction". In: *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '02. Portland, Oregon: ACM, 2002, pp. 58–70. ISBN: 1-58113-450-9. DOI: 10.1145/503272.503279. URL: http://doi.acm.org/10.1145/503272.503279.

[19] Orna Kupferman and MosheY Vardi. "Model checking of safety properties". In: *International Conference on Computer Aided Verification*. 1999, pp. 172–183.

[20] Shinji Kusumoto, Akira Nishimatsu, Keisuke Nishie, and Katsuro Inoue. "Experimental Evaluation of Program Slicing for Fault Localization". In: *Empirical Software Engineering* 7.1 (2002), pp. 49–76. ISSN: 1573-7616. DOI: 10.1023/A:1014823126938. URL: http://dx.doi.org/10.1023/A:1014823126938.

[21] Timothy M. Meyers and David Binkley. "An Empirical Study of Slice-based Cohesion and Coupling Metrics". In: *ACM Trans. Softw. Eng. Methodol.* 17.1 (Dec. 2007), 2:1–2:27. ISSN: 1049-331X. DOI: 10.1145/1314493.1314495. URL: http://doi.acm.org/10.1145/1314493.1314495.

[22] Alexander von Rhein. "Verification Tasks for Software Model Checking". MA thesis. Universität Passau, Aug. 2010.

[23] Xiangyu Zhang, Haifeng He, Neelam Gupta, and Rajiv Gupta. "Experimental Evaluation of Using Dynamic Slices for Fault Location". In: *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*. AADEBUG'05. Monterey, California, USA: ACM, 2005, pp. 33–42. ISBN: 1-59593-050-7. DOI: 10.1145/1085130.1085135. URL: http://doi.acm.org/10.1145/1085130.1085135.

ERKLÄRUNG

Hiermit versichere ich, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, March 29, 2017

Peter Dahlberg