UNIVERSITÄT PASSAU

# Abstract Transducers
for Software Analysis and Verification

Dissertation

by **Andreas Stahlbauer**
for the award of a **Doctorate in Natural Sciences (Dr. rer. nat.)**

submitted to the **Faculty of Computer Science and Mathematics**
of the **University of Passau**
in **August 2019**

Advisor and examiner: **Prof. Dr. Sven Apel**
External examiner: **Prof. Willem Visser, PhD**

ABSTRACT

Context

Whenever software faults can endanger human life, property, or the environment, the absence of faults must be ensured with utmost care and the best technologies available. Evidence is needed showing that all requirements are satisfied and that the risk of faults is reduced. One technique to conduct such a *verification task*—composed of the software to verify, the specification to check, and a model of the environment—is software model checking.

Problem

To conduct a verification task with a model checker, different models of the task are constructed. We distinguish between two types of task models: *syntactic task models* and *semantic task models*, which define the respective syntactic structure (control flow) and semantic structure (state transitions, invariants) of the verification task. When constructing such models, we can observe that similar structures and substructures reappear within and among different verification tasks. For example, the same assertions to check can appear in different functions, or the same predicate can be part of different invariants to describe sets of program states. Similarities that appear during the model construction process can be the result of solving similar reasoning problems, often solved using computationally expensive procedures (as typical for model checking), over and over again. Not reusing results of solving similar problems, not having a means for conducting repeated efforts automatically, or not trying to reduce the number of similar reasoning efforts, is a *waste of precious resources*.

Objectives

To address these problems, we present a common conceptual and technical foundation for sharing *syntactic and semantic task artifacts* for reuse, within and among verification runs. Both the syntactic construction of a verification task and the construction of its semantic model—which describes all possible behaviors and states—are covered. We study how commonalities and regularities in the task models can be taken into account to facilitate the process of sharing task artifacts for reuse, and to make the overall verification process more efficient and effective. We introduce *abstract transducers* as the theoretical foundation of this thesis: a type of finite-state transducers with an inherent notion of abstraction for states, the input alphabet, and its output alphabet. Abstracting these transducers allows us to widen both the set of input words for that they produce output and the sets of output words. Abstract transducers are instantiated as task artifact transducers to map from program structures to task artifacts to share. We show that the notion of abstraction provides a means for increasing the scope for that task artifacts are shared for reuse. We present two instances of task artifact transducers: Yarn transducers and precision transducers. We use *Yarn transducers* for providing code to weave into the control-flow structure of a computer program, and present the *Loom analysis* as a means for orchestrating the weaving process. *Precision transducers* provide a means for sharing abstraction precisions for reuse, thus aid in defining the level of abstraction of a semantic task model. For both types of transducers, we provide empirical evidence on their practical applicability, for example, to verify Linux kernel modules, and show that they can help in increasing the verification performance.

PUBLICATIONS

While most of this work consists of unpublished material, we have presented some of the ideas previously as part of the following publications. The authors are ordered *alphabetically*. For each paper, we provide a *summary*, outline the *contributions* of the author of this thesis, describe the *overlap* with the content of this thesis, and outline *inspirations* that we gained for follow-up work. This list is required by the doctoral degree regulations of the Faculty of Computer Science and Mathematics of the University of Passau.

1. D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. "Precision Reuse for Efficient Regression Verification." In: *Proc. ESEC/FSE*. ACM, 2013, pp. 389–399.

   Summary. This paper is on sharing abstraction precisions for reuse among verification runs. It provides evidence on the performance gains that can result from precision reuse in the context of regression verification. The empirical study is conducted on revisions of 62 Linux device drivers that cover more than 5 years of kernel development.

   Contributions. The author of this thesis was the principal investigator for this paper. He coordinated the contributions of the co-authors, designed and conducted all experiments, has created all tables and contributed a significant fraction of the paper's text. Furthermore, he presented the work at the Alpine Verification Meeting in Trento (Italy), at ESEC/FSE in St. Petersburg (Russia), at the Software Engineering Conference in Kiel (Germany), and GrammaTech Inc. in Ithaca (USA).

   Overlap. This thesis shares the idea that abstraction precisions are a precious intermediate verification result that should be reused with this paper. Furthermore, the paper helped to identify potential adverse effects of precision reuse. Nevertheless, this thesis does not share any texts, tables, or figures with that paper. Chapter 5 is an independent work on precision reuse, also independent from the other authors on the paper.

   Inspirations. Follow-up investigations resulted in prototypes for precision synthesis because we learned that many of the refinements could be saved by deriving predicates from the specification. Furthermore, this work helped to identify the potential of precision reuse within a model checker.

2. D. Beyer and A. Stahlbauer. "BDD-based Software Verification - Applications to Event-Condition-Action Systems." In: *STTT* 16.5 (2014), pp. 507–518.

   Summary. We implement a program analysis based on an abstract domain that encodes the state-space of a program in binary decision diagrams (BDDs) to verify safety properties. We compare the performance of this state-space encoding to various other established approaches for a restricted class of programs: event-condition-action systems from the challenge on Rigorous Examination of Reactive Systems (RERS).

Contributions. The author of this thesis implemented the BDD-based program analysis on top of CPAchecker, and participated with this implementation in the RERS challenge. Furthermore, he conducted all experiments, created tables, figures, examples, and plots. He discussed the results of the experiments and also contributed a significant portion of the paper's other text. The author of this thesis presented a preliminary version of this work at ISoLA 2012 in Crete (Greece).

Overlap. This thesis does not share any contributions with the paper.

Inspirations. The programs from the RERS challenge are an interesting class of programs. Our Reactive scenario, which we use in the empirical study in Chapter 4, is inspired by these programs.

3. D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. "Witness Validation and Stepwise Testification across Software Verifiers." In: *Proc. ESEC/FSE.* ACM, 2015, pp. 721–733.

Summary. This paper presents the concept of sharing error-witnesses among different verifiers for validation. This process increases the confidence in whether a given counterexample is a false alarm or not.

Contributions. The work in this paper was conducted in three phases, by three different teams of contributors. The first phase was about choosing and implementing an exchange format, discussions with different tool developers, and first experiments with the full set of SV-COMP programs. The second phase was about establishing the concept in the SV-COMP community, including discussions with tool developers on the value and integration of such an approach. The last phase was about conducting the experiments, fine-tuning the tools, and finishing the paper. The author of this thesis contributed considerably during the first and the last phase. He contributed the concept of step-wise testification, identified the role of abstraction for witness automata, identified and elaborated on the relationship to conditional model checking, designed and conducted the experiments on the effectiveness (reduction of the covered state space) of witness validation, and contributed large amounts of text to the paper. Furthermore, the author of this thesis presented the work, together with another co-author, at ESEC/FSE in Bergamo (Italy).

Overlap. Both the paper and the thesis share the idea of reusing (intermediate) verification results within and among runs of a verifier. Nevertheless, all contributions of the present thesis are independent work, and the paper can be seen as any other related work in the literature.

Inspirations. This thesis generalizes the idea of abstracting finite-state machines to increase their potential for reuse in a verification run.

4. S. Apel, D. Beyer, V. O. Mordan, V. S. Mutilin, and A. Stahlbauer. "On-the-fly Decomposition of Specifications in Software Model Checking." In: *Proc. FSE*. ACM, 2016, pp. 349–361.

Summary. A software model checker checks if a given program satisfies a given specification, which is composed of a set of properties. This work aims at dynamically partitioning a given set of properties such that the overall verification performance is increased. Along with this, we present an approach that interleaves weaving the specification with the program to check and the actual state-space exploration.

Contributions. The author of this thesis was the principal investigator for the paper. He coordinated the work with the co-authors, implemented the approach, designed and conducted all experiments, has created all tables and figures and contributed the majority of the concepts and text. Others contributed both the idea of taking the relevance of properties for the partitioning into account and the set of verification tasks (Linux kernel modules and a specification of Linux kernel API functionality). The author of this thesis presented the work at the FSE conference in Seattle (USA).

Overlap. The thesis presents a considerably refined version of the weaving mechanism, the Loom, along with refined finite-state machines for program specification.

Inspirations. We adopt the idea of using abstraction precisions to specify which properties to consider during the state-space exploration, which results in the notion of abstract precision.

THANK YOU

To my family, friends, colleagues, teachers, mentors, sponsors, advisors, assistants, partners, and all other people that had or have influence on my life: *Thank you* ♡ for supporting, teaching, mentoring, trusting, funding, motivating, calming, relaxing, pushing, and inspiring me.

> “*If I have seen further, it is by standing on the shoulders of giants.*”
>
> — Isaac Newton

CONTENTS

NOTATION

We provide a brief summary of notations that we use to present and formalize our contributions. Less frequently used symbols and notations are introduced locally. *Generally*, we *try* to stick to following *rules*:

$\cdot$	A centered dot $(\cdot)$ is used to abstract away details that are irrelevant for the discussion or explanation.
s_1, a_i, C_k	Subscripts are used to specify specific subsets, instances, or an enumeration of elements of collections; the subscript is dropped if it is clear from the context.
A, B, K	Sets are denoted by uppercase letters, such as the set A.
A^*	The set of all words over an alphabet A is denoted by A^*.
a, b, k	Elements of sets or lists are denoted by lowercase letters, such as the element a.
$\langle s_1, s_2, \dots \rangle$	The elements of sequences, lists, or vectors are enclosed in angle brackets.
(s_1, s_2, s_3)	The components of tuples are enclosed in round brackets.
$\{s_1, s_2, s_3\}$	The elements of sets are enumerated in curly brackets.
$[a, b]$	The closed interval of numbers from a to b (including a and b) is denoted by $[a, b]$, that is, it is enclosed in squared brackets.
$\bar{a} \in A^*$	A symbol s with a bar $\bar{a}$ denotes a vector, sequence, or (ordered) list.
$\widehat{a} \subseteq A$	A symbol a with a hat $\widehat{a}$ denotes a set.
a', a'', a'''	We use primed symbols—that is, symbols that are followed by one or more prime symbol $(')$—to denote related variables. In most of the cases, it denotes a new version of the variable that is the result of an assignment.
$\mathbb{N}$, $\mathbb{N}_0$	The symbol $\mathbb{N}$ denotes the set of natural numbers without zero; the symbol $\mathbb{N}_0$ denotes the natural numbers including zero.
$\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$	We use the common symbols for the set of integers $\mathbb{Z}$, the set of reals $\mathbb{R}$, and the set of Booleans $\mathbb{B}$.
$\cup$, $\cap$, $\subseteq$, $\subset$, $\setminus$	We use the common set-theoretic operators with their common meaning.
$\emptyset$, $\{\}$	The empty set is denoted by $\emptyset$ or by $\{\}$.
2^S	The power set of a set S, that is, the set of all subsets of S, is denoted by 2^S.
$s \in S$, $s \notin S$	A predicate that states that an entity s is element of a set S is denoted by $s \in S$; we use $s \notin S$ to denote the negation of this predicate.
$\{\cdot \mid \cdot\}$	We use set comprehension with its common meaning.
$\lvert A \rvert$	The number of elements in a collection A is denoted by $\lvert A \rvert$.
$A \times B$	The Cartesian product of two sets A and B is denoted by $A \times B$.
$\mathsf{foo} : C \to D$	A function or operator foo that takes an element a from a set A and returns an element b from a set B is declared by $\mathsf{foo} : A \to B$.

SYMBOLS

The following table lists symbols that are *frequently used* in this work, with their meaning and a reference to their definition:

$\vdash$	Syntactic entailment	13
$\models$	Semantic entailment	13
$\equiv$	Semantic equivalence	13
$[\![\cdot]\!]$	Semantic denotation (concretization function)	13
$\langle\!\langle\cdot\rangle\!\rangle$	Abstraction function	15
$\circ$	Concatenation	9
$\rightsquigarrow$	Abstract transition relation	16
$\rightarrow$	Concrete transition relation	10
δ	Control-state transition relation	36
ϵ	Empty word	9
η	Concern dependency relation	72
$\mathcal{E}$	Lattice	13
$\mathbb{H}$	Concern dependency graph	72
π	Abstraction precision	21
Π	Set of abstraction precisions	21
$\mathbb{\Pi}$	Precision lattice	134
C	Set of concrete states	15
CFA	Control-flow automaton	10
D	Abstract domain	15
$\mathbb{D}$	Configurable program analysis	21
E	Set of abstract states	15
$\mathcal{F}$	Set of formulas in predicate logic	13
G	Set of control-flow transitions	10
H	Set of task concerns	72
$\mathcal{L}$	Language	9
L	Set of control locations	10
ℓ	Depth of a lookahead	39
Op	Set of program operations	10
Θ	Set of output symbols (alphabet)	37
S	Set of properties (specification)	11
$\mathcal{P}$	Set of predicates	13
$\overline{P}$	Program (sequence of statements)	10
$\mathbb{P}$	Set of precision transducers	140
Q	Set of control states of an automaton	37
$\mathcal{T}$	Transductions	46
$\mathbb{Y}$	Set of Yarn transducers	93
Y	One Yarn transducer	93
Y_ϵ	The neutral Yarn transducer	93
X	Set of data locations	10

1 INTRODUCTION

Keywords: Software Model Checking, Static Analysis, Formal Methods, Safety Critical, Transducers, Abstraction, Artifacts, Sharing and Reuse

This thesis presents novel concepts and techniques that support automatic and algorithmic reasoning about software and its properties. We introduce abstract transducers as the common conceptual foundation to unify several techniques from the fields of program analysis and verification. We focus on applying abstract transducers for sharing task artifacts for reuse, for example, to compose analysis tasks on a syntactical level, or for creating semantic models of an analysis task in an efficient way. We implement and evaluate our techniques within one framework and illustrate the practical applicability and relevance of these concepts and techniques. Along with this work, bugs in the Linux kernel have been identified, reported, and fixed [36].

Context. Software has taken a critical role in today's world. Our society is dependent on reliable software systems and can be affected in dramatic ways if these—and the hardware the software is running on—do not perform their tasks. A system is called *safety critical* if its malfunction could harm people, equipment, property, or the environment [76].

Safety Critical Systems

Along with the increase of the criticality of software came an increase of the economic pressure to develop complex software systems fast, and bring these systems to the markets early. Regulation authorities have addressed this by establishing *standards* that enforce processes and tools that are appropriate for ensuring the quality of software systems. For example, the standard IEC 62304 for medical device software, DO-178C for avionics, and ISO 26262 for automotive systems. More general requirements on the development and maintenance of safety critical systems were formulated in the standard IEC 61508. Despite standards and regulations, we can read nearly weekly about a new safety critical bug that caused huge damage, for example, a bug in the Airbus A350 that requires a reboot after 149 h of operation[1] or a similar bug that affected Boeing's 787[2].

1 `https://ad.easa.europa.eu/ad/2017-0129R1`
2 `https://federalregister.gov/a/2015-10066`

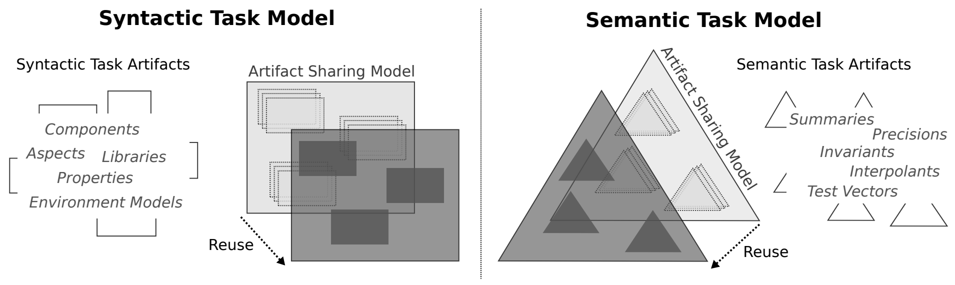


Figure 1: An artifact sharing model provides a means for providing different task artifacts of different kind at different *sharing points*, in different *contexts*, *for reuse*. The goal is to *reduce the overall effort* for reasoning about programs and their properties.

Testing and Verification

Various techniques for ensuring the quality of software systems have been proposed continuously since software engineering and the quality of its products has been identified to be critical [201]. Testing [200] is by far the most established technique to ensure the quality of software systems. The increased criticality of software systems, and progress in research, brings more elaborated techniques for ensuring the quality of software into focus. New techniques for formal verification—model checking and deductive verification—promise an increased practical applicability [162]. Whereas testing can show only the presence of bugs [96], more elaborated verification techniques, such as model checking, can prove their absence. Formal verification techniques are considered to be cost-effective, especially if the price of failures is high [50]. Already ethical considerations should force engineers to apply formal methods for safety-critical software systems [50]. A big success story of the application of formal methods is the verification of device drivers [19], which helped to increase the reliability of operating systems considerably. Formal verification techniques complement testing; these techniques can, in some cases, even replace tests and provide an exhaustive coverage of the state space, which is considered by the standard DO-178C and its extension DO-333. Reliable software and the verifying compiler have been declared as one of the grand challenges in computer science [139, 140].

Grand Challenge

Software Model Checking. This work contributes to getting closer to the ideal of fully verified computer programs—that is, programs for which the absence of bugs, regarding a formal specification, has to be proven—and *focuses* on concepts and techniques for software model checking. *Software model checking* [71, 151] is an automatic and algorithmic approach to explore the entire space of all states and behaviors of a program exhaustively. That is, contrary to testing—which checks only a limited number of paths—all possible paths that executions can take are checked.

Verification Task

Task Models. A software model checker operates on a verification task, which is composed of the program to analyze, an environment model, and the specification. The environment model describes the environment in which the program is supposed to operate in and to interact with; it provides assumptions that should be taken into account in the verification process to not cause false alarms. The specification provides a formal description of properties that the program must satisfy.

To motivate the problems that we address in this work, we introduce the notion of task models. A *task model* provides a formal and unambiguous description of an analysis task. A model checker deals with two types of task models—see Fig. 1 for an illustration: The *syntactic task model* models the syntactic structure of an analysis task, and the *semantic task model* models all semantically possible states and behaviors that are implied by the syntactic model. A *task artifact* is a piece of information that contributes to an analysis task, its construction and solution; it is a reusable entity from which a syntactic or semantic model of the analysis task can be composed of.

Syntactic Task Model

Semantic Task Model

Problem. We address the problem of repeated, similar, computational and manual efforts that arise when constructing models of verification tasks, both of syntactic task models and semantic task models: (1) To construct a semantic task model, a syntactic task model has to be constructed first, which is typically done in different tools, and with different approaches, and in a step before the verifier is invoked. This increases the *complexity of the tool chain*, and with it the *costs in the overall verification process*. (2) Different verification tasks, especially those that result from such a composition process, have *many commonalities*. That is, the verifier has also to come up with *similar solutions for these similar reasoning problems*—which can be the result of expensive computations—for these common parts. Not reusing intermediate verification results that have already been computed for similar problems is a waste of precious resources. (3) Despite *regularities* that can be observed both in the construction of syntactic task models and semantic task models, there is *no common formalism* that aids in both the construction of the syntactic and the semantic task model. We identified three core problems: (1) similar efforts among different tools or tool instances, (2) similar efforts within one verification run (tool instance), and (3) similar problems but different concepts and techniques to solve them.

[10] *Complex Tool Chain*

[11] *Similarities*

[12] *Common Foundation*

Goal. In this work, we aim at providing a fully-integrated approach, and tool, that provides a common conceptual and technical foundation for sharing and reusing task artifacts within and among verification runs, and to provide solid empirical evidence on the practical applicability and performance of this approach. We study how commonalities and regularities in syntactic and semantic task models can be taken into account to make the overall verification process more efficient and effective, and to provide generic mechanisms that aid in sharing of common task artifacts for reuse. We address the syntactic construction of a verification task and the construction of its semantic model—which describes all its possible behaviors and states.

1.1 CONTRIBUTIONS

Two themes are central for this work and are an integral part of most of our contributions: (1) *sharing and reuse* that is (2) *guided by abstract transducers*:

Sharing and Reuse. From the conceptual perspective, we aim at fostering sharing and reuse. The potential for sharing and reuse stems from commonalities between different sub-problems in the verification tasks and their construction, within a verification run and among different verification runs.

We share artifacts that can contribute to many different concerns of a verification task, at different points of a task model, in different contexts. We address the following questions: How to deal with different task artifacts that should independently contribute to solve a reasoning problem? Which adverse effects can sharing and reuse have? How do sharing and reuse contribute to the complexity of the verification task? What are the similarities between reasoning problems that make task artifacts reusable?

Transducer-aided Program Analysis. Automata can describe sets of words. Since also execution traces of programs are words in a specific formal language, we can use automata to describe sets of execution traces. An extension of automata are transducers, that is, automata that produce output on their states or transitions. We use this capability of transducers to provide different task artifacts, at different points of execution traces, that contribute to the composition or the solving of a verification task, that is, we use them as a means for sharing task artifacts for reuse. Transducers can provide different artifacts for reuse in different contexts; this can be important for (1) the soundness of the analysis, and (2) for an efficient analysis process since only artifacts that are useful in the corresponding context should (or can) be reused. We answer the following questions: How to formulate infinitely long output words that describe task artifacts? Is there an approach to summarize these output words in a finite representation? For which types of task artifacts is the approach applicable?

This work provides substantial contributions that have, at the time of writing this thesis, not yet been published in that form. For some of the concepts and techniques, prototypes and pilot studies were presented at major software engineering conferences [8, 36] before. Our work is in line with research [31, 32] on the convergence of static analysis and model checking: We (1) build on established concepts and formalisms from this community, and (2) provide results that advance its state of the art. The typical audience of this work is expected in the fields of *software engineering, programming languages, program analysis, and computer-aided software verification*. We make the following contributions:

Artifact Sharing Model. We introduce the notion of an *artifact sharing model*, which we instantiate based on transducers that provide task artifacts as output (*task artifact transducers*), as the conceptual foundation to describe possible compositions of syntactic task models and the corresponding semantic task models: It describes which task artifacts should be composed to arrive at the final task model. That is, a *family of task models* can be constructed based on an artifact sharing model. The artifact sharing model defines the *context* for that the reuse of artifacts is intended. Different syntactic models—in the form of relations that describe the control flow, and implicitly the data flow, of the programs to analyze—of verification tasks can be composed, for example, based on different properties to check. Different semantic models—in the form of Kripke structures—of verification tasks can be produced, for example, by modeling different details of the syntactic model with different levels of abstraction, that is, with different abstraction precisions. Figure 1 illustrates different application scenarios of artifact sharing models for sharing task artifacts for reuse.

1. *Abstract Transducers.* We introduce *abstract transducers* as the theoretical foundation of this work. Abstract transducers are abstract machines that map between an input language and an output language and can conduct a lookahead. Both their input alphabet and the output alphabet are composed of abstract words, where one abstract word denotes a set of concrete words. An *abstract word domain* provides the mapping between abstract words and concrete words, and means for abstraction (widening). The fact that abstract transducers produce an *intermediate language* has several implications on the design of algorithms that operate on these machines. We use techniques from abstract interpretation and domain theory as the foundation to provide means to compute *abstractions* of abstract transducers, including their output.

2. *Output Closure Abstraction.* Abstract transducers can emit exponentially many and infinitely long words as output, for example, code fragments that take advantage of the full Turing-completeness of a programming language, that is, including loops. We present a technique for computing *finite abstractions of the output* of ϵ-closures with *ϵ-loops*.

3. *Task Artifact Transducers.* We instantiate abstract transducers as *task artifact transducers* and with it as one instance of an artifact sharing model. They are a generic means to provide various artifacts, which contribute to different concerns of an analysis task, in different contexts of an analysis task for reuse. They *foster sharing and reuse* of components of an analysis task and the intermediate verification results that are produced while conducting an analysis. They are the foundation for several concepts and techniques that are presented in this work: We instantiate them as YARN transducers to compose syntactic task models, and as precision transducers to aid in the construction of semantic task models. We use the possibility to compute abstractions of task artifact transducers as a means to *increase the sharing of emitted task artifacts*: Task artifacts become emitted for a larger set of input words.

4. *Generic Transducer Analysis.* To execute abstract transducers—for example, task artifact transducers—as components of a program analysis framework, we provide an *abstract transducer analysis*. It is a generic analysis that tracks the current control state of a transducer and stores its output; we formalize and implement it as a configurable program analysis (CPA). Having this type of analysis at hand aids in providing a fully-integrated approach for program analysis, for example, to compose and construct both syntactic and semantic task models.

Syntactic Task Artifact Sharing and Reuse. With this group of contributions, we demonstrate that *syntactic* program artifacts can be shared for reuse based on abstract transducers. We formalize the syntactical fragments that we share as YARNS. A YARN denotes a set of sequences of program operations that is mapped to a specific task concern. The explicit mapping to a task concern fosters tractability and enables concern-specific state-space abstraction strategies. We present a fully-integrated approach for composing verification tasks by weaving YARN and describe an approach for conducting this composition along with their actual verification.

1. Yarn *and* Yarn *Transducers.* We introduce Yarn as an *abstract* syntactic task artifact, which can be composed into control-transition relations by weaving. We present Yarn *transducers*, an approach that is based on abstract transducers for providing Yarn to weave. A full program, a safety specification, or an environment model can be expressed based on a Yarn transducer.

2. *On-the-fly Weaving.* We present the Loom *analysis*, a program analysis for composing Yarn from different sources—for example, from Yarn transducers—by weaving. Loom introduces the possibility to *delegate the encoding* of information to different analysis components or analysis steps (for more efficient or effective handling), aids in providing *traceability* of program fragments (mapping to concerns), and enables the idea of *on-the-fly weaving*. That is, the construction of the syntactic task model is interleaved with the construction of the semantic model.

3. *Dynamic Control Encoding.* We introduce the concept of *dynamic control encoding*. An automaton analysis can delegate the encoding of the current control state, *on-demand*, to other analysis components by emitting *guards and assignments based on state variables* as Yarn to weave. This Yarn can then be encoded more efficiently by other analyses—with different, possibly symbolic, abstract domains. This approach gives the flexibility to use the full range (hybrid) of encodings of automata states dynamically: from pure explicit state representations to fully weaved, and possibly symbolic, representations.

4. *Empirical Study.* We systematically evaluate the performance of different control-encoding strategies based on a set of scenarios and a set of Linux modules. Our results show that the type of control encoding has a considerable influence on the performance of a verification tool. Depending on the characteristics of a verification problem, a different encoding strategy provides better performance. A detailed sensitivity analysis reveals the dependency of the performance to several parameters of a verification tool, which indicates directions for further studies.

Semantic Task Artifact Sharing and Reuse. This group of contributions aims at sharing semantic tasks artifacts for reuse. A *semantic task artifact* aids in summarizing the semantics of a fragment of a syntactic task model. We share and reuse abstraction precisions as semantic tasks artifacts. An *abstraction precision* determines the set of details of a verification task that the semantic model should describe for reasoning about a specific property. One instance of an abstraction precision is, for example, a set of predicates in predicate logic, which is used for computing predicate abstractions.

1. *Discover-Share-Reuse Scheme.* We provide a *scheme for expressing and comparing* strategies for sharing and reusing abstraction precisions. The scheme is applicable both for sharing and reuse within one verification run, but also for sharing and reuse among different runs of a verifier.

2. *Abstraction Precision.* We *formalize* the notion of abstraction precision and define a *lattice* of abstraction precisions. We define the *scope* of an

abstraction precision and introduce the notion of an *abstract precision* that separates abstraction precisions by *concern*.

3. *Precision Transducers.* We introduce precision transducers as a means to share abstraction precisions for reuse at different points of the control flow of a verification task, in different contexts. A *precision transducer* is an abstract transducer that emits abstraction precisions as its output.

4. *Precision Sharing Strategies.* Precision transducers are a flexible means to share abstraction precisions in precisely defined scopes. That is, different *precision sharing strategies* can be realized, or become realizable, based on precision transducers, including lazy abstraction [136]. These strategies can help to balance the positive and adverse effects of precision sharing (and reuse). We use the concept of precision transducer *abstraction* to increase the abstraction precision sharing scope. These strategies can help, for example, to reduce the sensitivity to changes in the context of regression verification.

5. *Empirical Study.* We conduct an empirical study to demonstrate the *applicability* and effects of different strategies for sharing abstraction precisions for reuse based on precision transducers. A novel sharing strategy that takes advantage of the expressiveness of precision transducers yields promising results. Furthermore, we present first results for precision synthesis based on precision transducers.

Replication Package. Along with this work, we provide a replication package. This package makes our results more convenient to reproduce; it includes the verification tasks of all case studies, the tool configurations, the raw results, and the full source code of our implementation. Parts of the replication package that ship with this thesis have already been evaluated [8, 36] by artifact evaluation committees of major conferences.

1. *Open-Source Implementation.* We instantiate our concepts based on the configurable program analysis (CPA) [31, 32] framework. We provide different CPAs and corresponding operators to run abstract transducers. We provide an open-source (Apache 2.0) implementation—based on CPAchecker—of our approach to the community.

2. *Sensitivity Plots.* To increase the validity of our results, we analyze the sensitivity of our approaches to different parameters, that is, we conduct sensitivity analyses. We provide *sensitivity plots* to obtain an overview of the sensitivity of different analysis configurations regarding a specific parameter. The implementation for creating this type of plot is part of the replication package.

3. *Tasks and Results.* We provide all programs and the corresponding specifications, that is, the verification tasks, as part of the replication package. Other tools and approaches can then quickly be evaluated based on these tasks. We ship the raw results of all experiments to make our results easy to verify.

1.2 OUTLINE

We begin (Chapter 2) by describing the established concepts and techniques on that the contributions of this work build. The chapter that follows (Chapter 3) introduces the concept of abstract transducers and various techniques that increases their general utility for the use in static analysis and software model checking. We then (Chapter 4) introduce the Loom analysis, the concept of Yarn, and Yarn transducers as a means to provide control-transition relations for composition by weaving. In the next chapter (Chapter 5) instantiate abstract transducers as precision transducers, that is, as a means to share abstraction precisions—with different levels of granularity—to various abstraction tasks in an analysis run.

2 BACKGROUND

This chapter introduces existing concepts and techniques for program analysis and software model checking that are the foundation of this work. Please note that definitions of different terms vary within the community. We use the formalisms and definitions that fit best for this work: Readers that are already familiar with the field of program analysis and model checking should take notice of the definitions, formalisms, and notions we use.

Possible sources for further information on basic concepts are the *Handbook of Practical Logic and Automated Reasoning* [130], the *Handbook of Model Checking* [73], and the *Principles of Program Analysis* [204].

2.1 PROGRAM REPRESENTATION

Before we introduce concepts for program analysis, we introduce formalisms and concepts to represent the *syntactic* structure of programs and do not focus on their semantics yet.

2.1.1 Language and Word

We take a language-theoretic approach for formalizing many of the concepts and techniques that we present in this work. The set of all *finite words* over an *alphabet* Σ is denoted by Σ^*, which is a free monoid $\Sigma^* = (\Sigma, \circ, \epsilon)$, also known as Kleene star, where *concatenation* $\circ : \Sigma^* \times \Sigma^* \to \Sigma^*$ is its binary operator and its neutral element ϵ is the *empty word*. A *word* $\bar{\sigma}$ is a sequence $\langle \sigma_1, \ldots, \sigma_n \rangle$ of symbols from the alphabet. The concatenation $W_1 \circ W_2$ of two collections $W_1 \subseteq \Sigma^*$ and $W_2 \subseteq \Sigma^*$ of words is conducted pairwise, and is defined as $W_1 \circ W_2 := \{\bar{w}_1 \circ \bar{w}_2 \mid \bar{w}_1 \in W_1 \land \bar{w}_2 \in W_2\}$. The set of all symbols in a word $\bar{s}$ is denoted by $\Sigma(\bar{s}) \subseteq \Sigma$. The *length* $|\bar{\sigma}|$ of a word $\bar{\sigma}$ is its number of subsequent symbols; the empty word has length $|\epsilon| = |\langle\rangle| = 0$. Given two words $\bar{\sigma} = \langle \sigma_1, \ldots, \sigma_n \rangle$ and $\bar{\tau} = \langle \tau_1, \ldots, \tau_m \rangle$, the concatenation $\bar{\sigma} \circ \bar{\tau}$ results in the word $\bar{c} = \langle \sigma_1, \ldots, \sigma_n, \tau_1, \ldots, \tau_m \rangle$ with length $|\bar{c}| = |\bar{\sigma}| + |\bar{\tau}|$.

13 Word

A word $\bar{\sigma}_a$ is *prefix* of another word $\bar{\sigma}_b$, that is, it is element $(\bar{\sigma}_a, \bar{\sigma}_b) \in \preceq$ of the prefix relation $\preceq$, if there exists a *suffix* $\bar{\sigma}_s \in \Sigma^*$ such that $\bar{\sigma}_b = \bar{\sigma}_a \circ \bar{\sigma}_s$. Given a word $\bar{s} = \langle s_1, \ldots, s_n \rangle$, the *prefix operator* $\mathsf{pre} : \Sigma^* \to \Sigma^*$ returns a new word without the *last symbol* $s_n = \mathsf{last}(\bar{s})$.

14 Prefix

Finitely long words over an alphabet Σ are denoted by Σ^*, the *infinitely long* ones are denoted by Σ^ω. The set of all words is denoted by $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ [185], with the *infinite iteration* $\cdot^\omega$ and the *finite iteration* $\cdot^*$.

15 Finite and Infinite

The set of all words over an alphabet Σ that is described by a structure S and that are considered well-formed regarding certain production rules is called *language* $\mathcal{L}(S) \subseteq \Sigma^\infty$ of S. The *empty word language* $\{\epsilon\}$ consists of the empty word ϵ only, the *empty language* corresponds to the empty set $\{\}$.

16 Language

2.1.2 Program

We formalize programs based on several building blocks: the set of all typed *data locations* X, the set of all *control locations L*, the set of all *program operations Op*, and the set of all *control transitions* $G = L \times Op \times L$.

A data location $x \in X$ corresponds to a specific memory cell on the heap or stack. The set of control locations (*program locations*) corresponds to possible values of the *program counter* register, which is also known as the *instruction pointer*. A control location $l \in L$ is a point in a program where the value of all memory locations can be observed [70]. Control locations help to identify common points in the control flow. We use a *special data location pc* $\in X$ that holds the current position in the control flow: It represents the program counter register. A control location can be *compound*: A composite $(l_p, l_c) \in L$ is composed of a *major control location* $l_p \in L$ and a *minor control location* $l_c \in L$. Please note that we add attributes to control locations along with this work, which might give control locations further semantics.

17 *Control Locations*

18 *Compound Locations*

19 *Program Operations*

The set of program operations $Op = Asu \cup Asg$ consists of assignments *Asg* and assumes *Asu* on data locations; we model the neutral program operation $\mathrm{nop} \in Asu$ as an assumption that always evaluates to *true*. Furthermore, we assume that the operation nop does not influence any temporal properties of the program under analysis, that is, a control-flow transition with the program operation nop is dual to an ϵ-move in a NFA [237]. Each operation $op \in Op$ is represented by an abstract syntax tree.

20 *Neutral Program Operation*

A *program* is a finite sequence $\overline{P}$ of program statements, where one *program statement* is compound of a set of program operations and can—depending on its evaluation result—jump to different control locations. A program is composed (written) to contribute to particular program concerns. The set H denotes all *program concerns*; each control transition can contribute to a set of different concerns.

21 *Program Concerns*

2.1.3 Control Flow Automaton

We represent a program $\overline{P}$ by a *control-flow automaton* [136] $CFA_{\overline{P}} = (L_{\overline{P}}, G_{\overline{P}}, l_0)$, with a set $L_{\overline{P}} \subseteq L$ of control locations, a *control transition relation* $G_{\overline{P}} \subseteq G$ of possible *control flows* (transitions) between the control locations, and the initial control location $l_0 \in L_{\overline{P}}$. The set of L_F of *final control locations* (states) is defined implicitly: It is the set of control locations where the CFA relinquishes its control [186].

22 *Control Path*

A *sequence of control transitions* $\langle g_1, \ldots \rangle \in G^\infty$ is *well-formed* regarding a control transition relation $G_r \subseteq G$ if $g_1 \in G_r$ and for all $i \in \langle 2, \ldots \rangle$, with $g_{i-1} = (\cdot, \cdot, l) \in G_r$ holds true that $g_i = (l, \cdot, \cdot) \in G_r$. A finite sequence of control transitions can also be written as $l_1 \xrightarrow{op_1} l_2 \ldots \xrightarrow{op_n} l_{n+1}$. A *control path*—also referred to as a *path segment*—is a well-formed sequence of control transitions; a *program control path* (or *program trace*) is a control path that starts in the initial control location $l_0 \in L_{\overline{P}}$ of a program and its *CFA*. Please note that we sometimes use the notion of control path in a projected form, that is, either as a sequence $\bar{\sigma} = \langle l_0, \ldots \rangle \in L^*$ of control locations or as a sequence $\bar{\sigma} = \langle op_1, \ldots \rangle \in Op^*$ of program operations.

```
  program = { statement ";" };
statement = declare | assign | condjump;
condjump = "if" assume "goto" stmtno "else" stmtno ;
   declare = ? declaration of arrays with different types and lengths ? ;
    assign = ? assignment to an array cell from a given expression ? ;
   assume = ? an expression that evaluates to the type 'bool' ? ;
   stmtno = ? an unique identifier ∈ ℕ₀ of a statement ? ;
```

Figure 2: The Turing-complete Pseudo Programming Language (PPL) and its pseudo grammar

The *transitive closure* of a set of locations $M \subseteq L$ regarding a transition relation G_r is denoted by $M^+ \subseteq L \times L$, that is, for all pairs $(l_1, l_2) \in M^+$ exists a well-formed control path from l_1 to l_2 on G_r. A set of locations $M \subseteq L$ is said to be *connected* if there exists a control path between every pair of locations in M, that is, for all $l_1, l_2 \in M$, either $(l_1, l_2) \in M^+$ or $(l_2, l_1) \in M^+$. 23 *Closure*

The set F represents *all functions* of a program. Given a function $f \in F$, the set $L(f) \subseteq L$ represents all control locations that belong to the function f, the set $G(f) = \{g \mid (l_1, \cdot, l_2) \in G \land l_2 \in L(f)\}$ represents all transitions that lead to a location in f. Given a control location $l \in L$, the function $F(l) \in F$ represents the function to that location l belongs to. 24 *Function Locations*

2.1.4 Language of a Program

The *language of a program* $\overline{P}$ and its control-flow automaton $CFA_{\overline{P}}$ is the set of all its program control paths $\mathcal{L}(\overline{P}) = \mathcal{L}(CFA_{\overline{P}}) \subseteq G^\infty$, that is, the set of sequences of control transitions that are well-formed regarding the control transition relation $G_{\overline{P}}$. The language $\mathcal{L}_{Op}(\overline{P}) \subseteq Op^\infty$ is a projection of $\mathcal{L}(\overline{P})$ with words over program operations only. The language $\mathcal{L}_L(\overline{P}) \subseteq L^\infty$ is a projection of $\mathcal{L}(\overline{P})$ with words over control locations only.

Figure 2 shows a basic Turing-complete language without functions, to illustrate the basic complexities we have to cope with. You can assume that functions are inlined and recursions are transformed to loops. The number of words of a (imperative) program is determined by its control structure and the transition system that is implied by it. Programming in a Turing complete programming language implies that there can be programs that result in exponentially many and infinitely long program traces.

2.2 PROGRAM SPECIFICATION

Since we aim at checking the correctness of programs, we need a formalism to specify which words are considered to be correct, that is, within the specification and which words violate the specification—and represent incorrect words (runs). A *specification* formulates requirements as a set of properties $\rho \subset S$. The set of all properties is denoted by S. A program can be seen as a refined form of a specification, that is, program and specification have different levels of abstraction. We consider each property to be a program concern, that is, $\mathbb{S} \subset \mathrm{H}$.

From the discussion of programs that are written in Turing complete languages, we have learned that the language $\mathcal{L}(\overline{\mathrm{P}})$ of a program can have infinitely long and exponentially many words, and all of them could be correct. A means for specifying desired properties of such execution sequences is Linear Temporal Logic (LTL) [115, 243], which extends classical logic and is considered to be one natural and unambiguous way of describing the temporal behavior of a computer program [179].

25 LTL

2.2.1 Safety and Liveness

We distinguish between safety properties and liveness properties [178, 187], and do not discuss hyperproperties [74]: A *safety property* specifies that something (bad) must not happen [178]. Checking safety properties is equivalent to checking invariants, that is, we check that the (global) invariant is always satisfied (a violation never happens); assertions that always hold are a typical example for invariants. The violation of safety properties is witnessed by counterexamples of finite length [171]. Please note that also *test goals* [33, 114, 141, 221] are expressible as safety properties—literature also uses the term *never claims* [101] or *trap properties* [114]. Tests goals can be derived based on a coverage criteria [141]. A *liveness property* specifies that something (good) must happen eventually—that is, after a finite number of steps. The violation of a liveness property is witnessed by a counterexample of infinite length; checking liveness requires to prove termination.

26 Safety Properties

27 Test Goals

28 Liveness Properties

2.2.2 Behavior and State

Literature also distinguishes properties based on the extent to which temporal aspects are relevant. For example, if we say that we check the correctness regarding a specification, we also say that we check that there are no *states* or *behaviors* that violate the specification. The Oxford Dictionary defines "to behave" as "act or conduct oneself in a specified way", whereas "the state" is defined as "the particular condition that someone or something is in at a specific time." The model checking community uses [235] the terms *data property* (data-dependent property) and *control property* (control-dependent property) to talk about state and behavior.

A pure *control property* only refers to the control flow, that is, the sequence of program operations or control locations. The control locations in a sequence thereof must appear in a particular temporal order. This corresponds to the expressiveness that is brought by the temporal operators of temporal logic. A pure *data property* does not have a notion of temporal order: It does not refer to any (sequence of) control locations. It makes propositions about data locations and their values for one point in time only.

29 Control Properties

30 Data Properties

Please note that control properties can be converted to data properties and vice versa [61]. Also, hybrid forms of these property types are possible. Dually, safety checking (reachability) can be expressed as global invariant checking [214], and liveness checking can be performed as safety checking [229].

2.3 PROGRAM SEMANTICS AND MODEL

After the previous section has introduced the syntactical program representation, we now discuss how we represent and model the semantics of program executions and their space of possible behaviors and states.

Some general terms and definitions upfront: A proposition in the form of a phrase a is said to be *valid* if it is true for all its possible interpretations. The *denotation function* $[\![a]\!]$ provides the set of all semantically valid interpretations of a phrase a. Two phrases a and b are said to be semantically equivalent $a \equiv b$ if $[\![a]\!] = [\![b]\!]$. A proposition b is a semantic consequence of a proposition a, that is, $a \vDash b$, if $[\![a]\!] \subseteq [\![b]\!]$. We say that proposition b *overapproximates* proposition a if $[\![a]\!] \subseteq [\![b]\!]$. A decision procedure is called to be *sound* if only logically valid propositions are deduced. A decision procedure is said to be *complete* if all logically valid propositions can be deduced.

31 Overapproximation

32 Soundness and Completeness

2.3.1 Predicate Logic

The semantics of programs can be captured in formulas in predicate logic [122, 161]—also known as first-order logic. Predicate logic is defined based on a set $\mathbb{V}$ of variables, a set of logical symbols $\mathbb{X}$, a set of n-ary functions $\mathbb{F}$, with $n \geqslant 0$, a set of m-ary predicates $\mathbb{U}$, with $m > 0$, and a set of rules to derive well-formed formulas $\mathcal{F}$ in predicate logic. We use the set of logical operators $\mathbb{X} = \{\forall, \exists, \wedge, \neg, \vee, \Rightarrow, \Leftrightarrow\}$ with their common meaning. A binary operators $* \in \mathbb{X}$ on formulas forms the relation $* : \mathcal{F} \times \mathcal{F} \to \mathcal{F}$, a unary operator $\star \in \mathbb{X}$ forms the relation $\star : \mathcal{F} \to \mathcal{F}$. We use the symbols $\vartheta, \varphi, \psi, \rho \in \mathcal{F}$ to denote separate formulas. A *predicate* asserts attributes of a set of objects; it is a formula in predicate logic with n free variables. The *vocabulary* of a formula ϑ is the set $\texttt{vocab}(\vartheta)$ of all symbols that identify variables and functions. The set of all predicate formulas $\mathcal{P} \subseteq (\mathcal{F} \cup \{\mathit{true}, \mathit{false}\})$ also includes the formulas *true* and *false*.

33 Vocabulary

34 AllSat

Given a formula $\vartheta \in \mathcal{F}$ and a sequence of Boolean variables $V \in \mathbb{V}^*$, the operator $\mathsf{AllSat} : \mathcal{F} \times 2^{\mathbb{V}} \to \mathbb{B}^*$ returns a truth table, where each row corresponds to a truth assignment to the variables V that make ϑ satisfiable.

2.3.2 Lattices

Lattices are one of the most fundamental structures that this work builds on. They are used, for example, to compute abstractions [80]. Building program analyses based on lattices [80, 155] allows relying on fixed-point theorems for checking convergence in the state-space exploration process [174].

Lattice. We define a (complete) *lattice* [123] as a tuple $\mathcal{E} = (E, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$, with a set of *abstract entities* E and an *partial order relation* $\sqsubseteq \subseteq E \to E$. The relation $\sqsubseteq$ is also called the *inclusion relation* [250], and implies a graph that represents this relation [46]. The operator *meet* (infimum) is a relation $\sqcap : (E \times E) \to E$ that provides the *greatest lower bound* for a given pair $(m_1, m_2) \in E \times E$ of abstract entities. The operator join (supremum) is a relation $\sqcup : E \times E \to E$ that provides the *least upper bound* for a given pair of abstract entities. The *bottom* entity $\bot$ is the *least* in the partial order relation, that

35 Inclusion Relation

is, there exists no other entity $m \sqsubseteq \bot$, with $m \neq \bot$. The *top* entity $\top$ is the *greatest* in the partial order relation, that is, there exists no other entity $\top \sqsubseteq m$, with $m \neq \top$. The operators $\sqcap$ and $\sqcup$ extend to sets naturally: The meet over a set of abstract entities is denoted by $\bigsqcap : 2^E \times E$, and the join by $\bigsqcup : 2^E \times E$, for example, $\bot = \bigsqcap E$ and $\top = \bigsqcup E$. A *join semilattice* is similar to a complete lattice except that no meet is available for all entities in E. A *meet semilattice* is similar to a complete lattice except that no join is available for all entities in E. That is, a *semilattice* either has no meet or no join for all abstract entities. Partially ordered sets (*posets*) can be made semilattices, and semilattices can be made *complete* by adding additional abstract entities with special meaning (supremum or infimum) [113].

36 $\mathsf{fl}(E)$

Flat Lattice. A *flat lattice* [123, 255] is a widely used type of lattice over a set of elements E. We denote such a lattice over E by $\mathsf{fl}(E) = (E, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$. A given pair of elements is the inclusion relation $(e_1, e_2) \in \sqsubseteq$ if and only if the first element $e_1 = \bot$ is the bottom element or if the second element $e_2 = \top$ is the top element of the lattice.

37 $\mathsf{ml}(K, \ddot{V})$

Map Lattice. A *map lattice* $\mathsf{ml}(K, \ddot{V}) = (2^{K \to V}, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ is a lattice of elements that are maps, that is, the elements are functions that map from a set K of keys to a set V of values; the values of this map are elements of another lattice $\ddot{V} = (V, \sqsubseteq_V, \sqcap_V, \sqcup_V, \top_V, \bot_V)$. Such a lattice is also known as *function lattice* [16, 99]. The inclusion relation $\sqsubseteq$ has element $(m_1, m_2) \in \sqsubseteq$ if and only if $\forall k \in K : m_1(k)_\bot \sqsubseteq_V m_2(k)_\bot$. In the following, we rely on the function $m(k)_\bot = m(k)$ if $(k, \cdot) \in m$ otherwise $\bot_V$ which returns the value for a given key k from a map m, and the bottom element of the value lattice if no entry for the key is present. The meet $\sqcap$ is defined by $\sqcap(m_1, m_2) = \{(k, v_1 \sqcap_V v_2) \mid (k, v_1) \in m_1 \wedge v_2 = m_2(k)_\bot\}$, the join $\sqcup$ is defined by $\sqcup(m_1, m_2) = \{(k, m_1(k)_\bot \sqcup_V m_2(k)_\bot) \mid k \in \mathsf{keys}(m_1) \cup \mathsf{keys}(m_2)\}$, the top element $\top$ is defined by $\top = \{(k, \top_V) \mid k \in K\}$, and the bottom element $\bot$ is defined by $\bot = \{(k, \bot_V) \mid k \in K\}$.

38 $m(k)_\bot$

39 *Operator* $\bigsqcup_\to$

We define an *image-join operator* $\bigsqcup_\to : 2^{K \times E} \to 2^{K \to E}$: Given a map $M \subseteq K \times E$, with a set of keys K, and a set of lattice elements E, the operator joins all tuples $(k, e) \in M$ with the same key k into one tuple with a value that aggregates all value elements e, that is, $\bigsqcup_\to M = \{\ (k, \bigsqcup\{e \mid e \in \{(k, e) \in M\}\}) \mid k \in \{k \mid (k, \cdot) \in M\}\ \}$.

40 $\mathsf{pw}(\ddot{E})$

Powerset Lattice. A powerset lattice that describes a Hoare powertheory [4, 103, 131]—over a given lattice $\ddot{E} = (E, \sqsubseteq_E, \sqcap_E, \sqcup_E, \top_E, \bot_E)$—is denoted by $\mathsf{pw}(\ddot{E}) = (2^E, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$, where the set of elements is constituted by the set of all subsets 2^E of the set E. The inclusion relation $\sqsubseteq$ has the element $(E_1, E_2) \in \sqsubseteq$ if and only if $\forall e_1 \in E_1 \exists e_2 \in E_2 : e_1 \sqsubseteq_E e_2$. The join $\sqcup(E_1, E_2) = E_1 \cup E_2$ is the union, and the meet $\sqcap(E_1, E_2) = E_1 \cap E_2$ is the intersection of two given sets $E_1, E_2 \subseteq E$. The bottom element $\bot = \emptyset$ is the empty set, and the top element $\top = E$ is the set with all elements.

Prefix Lattice. It is well known [2, 53, 251] that a (sequential) program can be characterized by a prefix-closed set of sequences of program operations. A set W of words is called *prefix closed* if for each word $\bar{\sigma} \in W$ also every single prefix of the word is included in the set W. That is, a set W is prefix closed if it is a downset regarding a partial order of words that describes whether or not one word is prefix of another word. The prefix lattice describes a corresponding partial order of words: The *prefix lattice* $\mathsf{pr}(\Sigma^\infty) = (\Sigma^\infty, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ describes the relationship between words Σ^∞ over an alphabet Σ. A word is a sequence $\bar{\sigma} = \langle \sigma_1, \ldots \rangle \in \Sigma^\infty$ of symbols from the alphabet Σ. Two words are in the inclusion relation $(\bar{\sigma}_1, \bar{\sigma}_2) \in \sqsubseteq$, with $\sqsubseteq \subseteq \Sigma^\infty \times \Sigma^\infty$, if and only if word $\bar{\sigma}_1$ is prefix of word $\bar{\sigma}_2$. A word $\bar{a}$ is *prefix* of a word $\bar{b}$ if and only if $\bar{a} = \bar{b}$ or $\forall_{i=1\ldots n} b_i = a_i$, where a_i is the ith letter of the corresponding word, and n is the length $|\bar{a}|$ of the word $\bar{a}$. The definitions of the join (least upper bound) $\sqcup : \Sigma^\infty \times \Sigma^\infty \to \Sigma^\infty$, and the meet (greatest lower bound) follow from the relation $\sqsubseteq$. The bottom symbol $\bot$ of the lattice corresponds to the empty word ϵ. The symbol $\top$ is added to complete the lattice such that for all $\bar{\sigma} \in \Sigma^\infty$ holds that $(\bar{\sigma}, \top) \in \sqsubseteq$.

41 $\mathsf{pr}(\Sigma^\infty)$

Boolean Algebra. Any complemented distributive lattice is isomorphic to a Boolean algebra [146], which also follows from the Stone duality [240]; one example for such lattices are powerset lattices. Lattices generalize Boolean algebras by not requiring complement and distributivity in the first hand.

2.3.3 Abstraction

We reason about the states of a program on a computer and distinguish between concrete states and abstract states. One *concrete state* represents the full state of a computer at one point (CPU cycle) in time—a concrete state can be seen as one vector of bits. One *abstract state* describes a set of concrete states—a set of concrete states is sometimes called *region*.

Abstract Domain. The *abstract domain* [31] defines mechanisms for mapping between concrete and abstract entities and for reasoning about the relationship between abstract entities. The abstract domain concept we build on was introduced in the context of abstract interpretation [80, 81, 104] and is the foundation for several program analysis frameworks [31, 82, 83, 155, 224, 227]. An abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ is an algebraic structure that consists of the set of concrete elements C, a (semi-)lattice $\mathcal{E} = (E, \top, \bot, \sqsubseteq, \sqcup, \sqcap)$ on the set E of abstract elements, a denotation function $[\![\cdot]\!] : E \to 2^C$, and an abstraction function $\langle\!\langle\cdot\rangle\!\rangle : 2^C \to E$. The *denotation* (concretization) function $[\![\cdot]\!] : E \to 2^C$ maps from an abstract state $e \in E$ to a set $C \subseteq C$ of concrete elements. The denotation $[\![e]\!] : E \to 2^C$ of an abstract element e is the set of all its possible interpretations—as known from denotational semantics [4]. The *abstraction* function $\langle\!\langle\cdot\rangle\!\rangle : 2^C \to E$ maps from a set of concrete elements to an abstract state, that is, it is a surjection that provides a *symbolic representation*. The abstraction $\langle\!\langle C_k \rangle\!\rangle$ of a set of concrete elements $C_k \subseteq C$ results in an abstract element e, with $[\![e]\!] = C_k$. Please note that constructing a symbolic (abstract) representation does not imply any loss of information.

Two elements are called *semantically equal*, that is, $e_1 \equiv e_2$, if and only if $[\![e_1]\!] = [\![e_2]\!]$ in the same universe. One element *semantically implies* another element, that is, $e_1 \vDash e_2$, if and only if $[\![e_1]\!] \subseteq [\![e_2]\!]$.

Widening. The term abstraction is typically associated with a loss of information, that is, the number of interpretations of a given entity is increased. We reflect this in the abstraction function $\langle\!\langle\cdot\rangle\!\rangle^\pi : E \to E$ *with widening*. The widening is performed based on a given *abstraction precision* $\pi \in \Pi$ [202],

42 *Abstraction Precision*

which defines the set of details (facts) to keep or to discard. The result of a widening $e' = \langle\!\langle e \rangle\!\rangle^{\pi}$ in a new abstract element e' with $[\![e]\!] \subseteq [\![e']\!]$.

Composite Domain. A list of abstract domains $D_1, \ldots, D_n$ can be combined [31] to form a *composite domain* $D_\times$. A composite domain operates on *composite elements* $E_\times \subseteq E_1 \times \ldots \times E_n$, which are elements of a *product lattice*, and form tuples of abstract elements from the component domains [31].

2.3.4 Concrete and Abstract Semantics

We define the semantics of a program operationally and rely on the formalization that is provided with predicate transformers [95]—as used in predicate abstraction [122]. One formula $\vartheta \in \mathcal{F}$ denotes a set $[\![\vartheta]\!]$ of concrete program states. The *strongest postcondition* is an operator $\mathsf{SP} : \mathcal{F} \times Op \rightarrow \mathcal{F}$. That is, the result of a call $\mathsf{SP}_{op}(\vartheta)$, also written as $\mathsf{SP}(\vartheta, op)$, results in a new formula ϑ' that satisfies the Hoare triple $\{\vartheta\}\ op\ \{\vartheta'\}$ [138]. The operator naturally extends to sequences of program operations such as $\bar{s} = \langle op_1, \ldots, op_n \rangle \in Op^*$, that is, $\mathsf{SP}_{\bar{s}}(\vartheta) = \mathsf{SP}(op_n, (\ldots, (\mathsf{SP}(op_1, \vartheta))$.

43 *Strongest Postcondition*

Since we model the state space of a program based on an abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$, we use the term *abstract state* for an abstract element that describes a set of concrete (program) states. An abstract state $e \in E$ is an element in the corresponding lattice $\mathcal{E}$ of abstract states E. The set C corresponds to the set of *concrete states*. We overload the strongest postcondition operator to operate on abstract states $\mathsf{SP} : E \times Op \rightarrow E$, and on concrete states $\mathsf{SP} : C \times Op \rightarrow 2^C$.

Each concrete state $c \in C$ is a map of data locations X to corresponding data values with the respective type—we restrict the discussion to integers for now. We assume that each operation is atomic regarding its execution on a computer. The current status of a computer can be observed in discrete points in time.

2.3.5 Transition Systems

The semantics of a program, with all its states and behaviors, can be captured in a transition system, for example, a Kripke structure [54, 69]. A software model-checking procedure creates a reachability graph to prove a safety property, A control-flow automaton *CFA*, in contrast, describes a program on a syntactical level, that is, it has a language $\mathcal{L}(CFA)$ that consists of all syntactically well-formed program control paths.

Abstract Reachability Graph. An *abstract reachability graph* (ARG) is a graph $ARG = (R, e_0, \rightsquigarrow)$, with a set of reached abstract states $R \subseteq E$, an initial abstract state $e_0 \in R$, and an abstract transfer relation $\rightsquigarrow \subseteq E \times E$. That is, each node corresponds to an abstract state $e \in R$, and an edge between two nodes e and e' represents the transition $(e, e') \in \rightsquigarrow$ between two abstract states—which we also write as $e \rightsquigarrow e'$. The ARG is implicitly parameterized with an abstract domain D that provides the mapping between abstract states and concrete states. The *size of an abstract reachability graph*—also referred to as the *size of the abstract model*, or the *size of the abstract state space*—is the number of abstract states $|R|$. We called an abstract reachability graph to be *labeled* if each transition $(e, e') \in \rightsquigarrow$ between abstract states is labeled with a program operation $op \in Op$ of a corresponding control flow transition $g = (l, op, l') \in G$. The labeling is made explicit by the labeled transfer relation $\overset{Op}{\rightsquigarrow} \subseteq E \times Op \times E$.

Compared to a Kripke structure [54, 69, 173], an ARG does not have a notion of state labeling, but such a labeling can be performed, such that the ARG is *lifted* to become a full or partial [249] Kripke structure [55, 249].

Program Path. An *abstract program path* is a sequence $\bar{e} = \langle e_0, \ldots, e_n \rangle \in E^{\infty}$ of abstract states that starts in the initial state $e_0 \in E$, and is well-formed regarding the transfer relation $\rightsquigarrow$ of the corresponding abstract reachability graph; we also write it as with $e_0 \rightsquigarrow \ldots \rightsquigarrow e_n$. One abstract program path represents a set of concrete program paths $[\![e_0]\!] \rightsquigarrow \ldots \rightsquigarrow [\![e_n]\!]$. A *concrete program path* is a path $c_0 \rightarrow \ldots \rightarrow c_n$, with $c_i \in [\![e_i]\!]$. An abstract program path is *feasible* if each abstract state along the path has a concrete counterpart, that is, $\forall_{i \in [0,n]} \exists \, c_i \in [\![e_i]\!]$, otherwise it is called *infeasible*. A concrete program path is also called a *test vector*.

44 *Abstract Program Path*

45 *Concrete Program Path*

Inductive Invariants. Given a transition system in the form of an ARG $(R, e_0, \rightsquigarrow)$. A formula $\vartheta \in \mathcal{F}$ is an *inductive invariant* for this transition system if (1) then invariant holds in the initial state e_0, that is, $[\![e_0]\!] \subseteq [\![\vartheta]\!]$, and (2) it holds for every state e' that is transitively reachable on the transition relation [106, 210], that is, $[\![e']\!] \subseteq [\![\vartheta]\!]$. A sequence of abstract states $\bar{e} = \langle e_1, \ldots, e_n \rangle$ satisfies invariant ϑ if and only if $\forall_{e_i \in \bar{e}} [\![e_i]\!] \subseteq [\![\vartheta]\!]$.

2.4 MODEL CHECKING PROGRAMS

Software verification is considered a grand challenge of computer science [139]. Several different approaches exist to verify the correctness of a program concerning a given specification. We choose software model checking for this purpose. *Model checking* is an automatic, algorithmic, and exhaustive analysis of the states and behaviors of systems that can be described by state-transition systems [73]. Given a system S and a specification φ, a model checker constructs a model K that overapproximates all possible states and behaviors of S and checks whether this model satisfies the specification φ or not, that is, $K \vDash \varphi$, where the symbol $\vDash$ represents the semantic entailment. A *software model checker* takes as input a program $\overline{P}$ (the system to verify) and a specification and outputs whether this specification is satisfied or not—in an algorithmic, automatic, and exhaustive fashion.

46 *Software Model Checker*

2.4.1 Problem Characterization

The applicability of model checking faces theoretical and practical limitations. First and foremost, Gödel's incompleteness theorem [121] states that there cannot be a complete and sound solution for all possible model checking problems (verification tasks). In practice, we face the *state space explosion problem*, which describes the fact that the size—the number of states and transitions—of a Kripke structure can be exponential several times, regarding different factors. Each factor represents a sub-problem with high (exponential) costs on its own.

47 *Control Explosion*

First, there is the *path explosion problem* or *control explosion problem*, that is, with each branching in the control flow, the size of the state-space potentially doubles and is therefore exponential in the number of branchings—there can be exponentially many words in the language $\mathcal{L}(\overline{P})$ of the program $\overline{P}$.

48 *Unwinding Explosion*

The next dimension is the *unwinding explosion problem* that can result in infinitely long words in the language $\mathcal{L}(\overline{P})$ of the program, which arise from unwinding loops and recursion. *Loop invariants* [111, 167] are needed to construct a finite abstraction of this possibly infinite behavior. In the case of liveness properties, also termination has to be proven.

49 *Data Explosion*

Another dimension, the *data explosion problem*, arises from the data in the system's memory. Given a system with $|X| = n$ data locations, each data location is mapped to a domain D; the domain defines the number of different abstract representations of different values for a data location. The number $|C|$ of concrete states therefore raises exponentially in the number of data locations, where the factors are determined by the domains: $D_1 \times \ldots \times D_n$. Please note that also costs of some decision procedures that are used for symbolic state-space representation can be quadratic or exponential [177] in the number of variables regarding time or space.

50 *Interleaving Explosion*

The next dimension, the *interleaving explosion problem*, arises from all possible interleavings of code fragments (components, processes, or threads) in the context of the analysis of concurrent systems or compositional reasoning. The number of states in the Kripke structure can be exponential in the number of processes [211] or the number of components [117] to interleave.

51 *Configuration Explosion*

Another dimension, the *configuration explosion problem*, is defined based on the different compile-time or run-time configurations that a program can have [6, 222]. The number of variants a system can have is exponential in the number of configuration options.

2.4.2 Counterexample-Guided Abstraction Refinement

52 *Abstraction*

The most effective approach to cope with the state-space explosion is abstraction, that is, to remove (abstract from) all details of the system that are irrelevant for the reasoning task at hand. Or dually, model only those details

53 *Abstraction Precision*

that are relevant to solve the task. The set of details of a system to model is called the *abstraction precision* [32, 202], which can be, for example, a set of predicates, or a set of data locations.

The most prominent approach to derive an abstract model that is sufficiently precise to prove that a program adheres to a given specification is *counterexample-guided abstraction refinement* (CEGAR) [67]. A scheme of CEGAR is illustrated in Fig. 3. The abstraction process starts with an empty abstraction precision $\pi_0 = \emptyset$, which is used to construct a first abstract model (abstract reachability graph) of the program. The abstract model is then checked for abstract states that violate the specification. The program is safe if no violating state is found because the model checking procedure ensured that the abstract model overapproximates all possible states and behaviors that the concrete system can have. In case a violating state is found, a

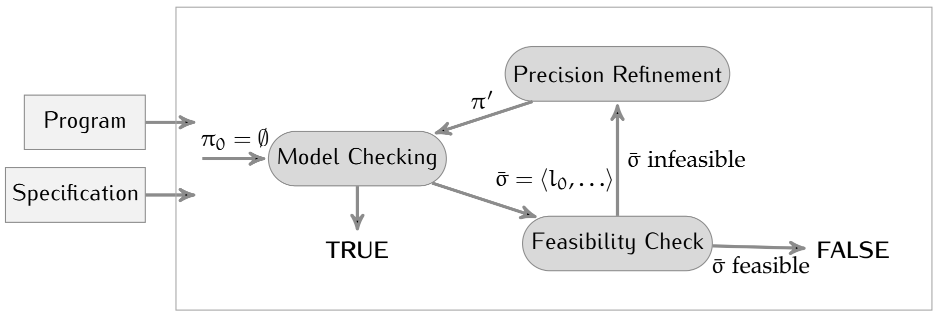


Figure 3: Counterexample-Guided Abstraction Refinement (CEGAR)

counterexample is constructed as a witness for the violation; for safety properties, such a counterexample is a finite program path $\bar{\sigma} = \langle l_0, \ldots \rangle$ [170]. This path might be infeasible in the concrete program, that is, $\mathsf{SP}_{\bar{\sigma}}(\mathit{true}) \equiv \mathit{false}$, and was only reachable in the abstract model due to a lack of precision; we call such a counterexample *spurious*. To eliminate spurious counterexamples, and to prove the absence of violations, the abstraction precision—and with it, the abstract model—has to be refined. Additional details to model are identified based on the counterexample and added to a new abstraction precision π' such that it does not re-appear in the next iteration of the CEGAR loop. Different techniques are available to identify the details to rule out a spurious counterexample, for example, Craig interpolation [84].

54 *Spurious Counterexamples*

55 *Precision Refinement*

2.4.3 Craig Interpolation

We use Craig interpolation [84, 190] to derive precision refinements from spurious counterexamples. It is a technique to get a (more) abstract explanation ϕ for the unsatisfiability of a formula ϑ in predicate logic and helps to localize different facts to track alongside a counterexample [137].

We use McMillans formulization [190] of the Craig interpolation theorem [84]: Given a formula $\vartheta \equiv \vartheta^- \wedge \vartheta^+$, a formula ϕ is a Craig interpolant if and only if

$$\begin{aligned} & \vartheta^- \wedge \vartheta^+ \ \mathit{unsat} \\ \text{and } & \vartheta^- \Rightarrow \phi \\ \text{and } & \phi \wedge \vartheta^+ \ \mathit{unsat} \\ \text{and } & \mathsf{vocab}(\phi) \subseteq \mathsf{vocab}(\vartheta^-) \cap \mathsf{vocab}(\vartheta^+). \end{aligned}$$

A Craig interpolant can be obtained in polynomial time [192] from the proof of an SMT solver if the background theories have the interpolation property.

2.4.4 Predicate Abstraction

Predicate abstraction [122] is an approach to summarize (abstract or widen) a given formula $\vartheta \in \mathcal{F}$ in predicate logic based on a set of predicates $\pi \subset \mathcal{P}$. It is an essential [19, 28, 41, 177] means to derive a finite abstract model of a program. The formula that should be summarized by predicate abstraction represents a set $C_\varphi = [\![\varphi]\!]$ of concrete program states. A program analysis typically conducts such an abstraction at specific points [35] in the control flow—the *abstraction locations*, for example, loop heads [28, 35]. The resulting abstraction is a Boolean combination of the given set of predicates π, that is, it is an abstraction with widening $\langle\!\langle\cdot\rangle\!\rangle : \mathcal{F} \times 2^{\mathcal{P}} \to \mathcal{F}$, which we also write as $\langle\!\langle\cdot\rangle\!\rangle^\pi : \mathcal{F} \to \mathcal{F}$. We distinguish between Boolean predicate abstraction and Cartesian predicate abstraction [21, 28]:

Boolean Predicate Abstraction. A *Boolean predicate abstraction* $\langle\!\langle\vartheta\rangle\!\rangle^\pi_{\mathbb{B}}$ of a given formula $\vartheta \in \mathcal{F}$ is the *strongest* Boolean combination $\psi \in \mathcal{F}$ of predicates $\pi \subset \mathcal{P}$ that is entailed by ϑ, that is, $\vartheta \Rightarrow \psi$ [177]. The resulting formula ψ represents a set of concrete states such that $[\![\varphi]\!] \subseteq [\![\psi]\!]$.

The predicates π are ordered (are assigned an index), which results in the list $\bar{\rho} = \langle \rho_1, \ldots, \rho_n\rangle \in \mathcal{P}^*$ of predicates. We introduce a propositional variable $v_i \in \mathcal{F}$ and get $\bar{v} = \langle v_1, \ldots, v_n\rangle \in \mathcal{F}^*$. We solve the following AllSat [177] problem, which results in a truth table $R \subseteq \mathbb{B}^{|\mathcal{P}|}$ with at most $2^{|\pi|}$ entries:

$$R = \mathsf{AllSat}(\vartheta \wedge \bigwedge_{i \in 1..|\bar{\rho}|} v_i \Leftrightarrow \rho_i, \bar{v})$$

A row in the result table R is a tuple $\bar{r} = \langle r_1, \ldots, r_n\rangle \in \mathbb{B}^{|\pi|}$. For each row $\bar{r} \in R$ we conjunct all predicates for that the propositional variable v_i was evaluated to *true*. The resulting abstraction formula ψ is the disjunction of all these conjunctions:

$$\psi = \bigvee_{\bar{r} \in R} \bigwedge_{\rho_i \in \bar{\rho}} \{\rho_i \mid r_i \equiv \mathit{true}\}$$

The AllSat procedure is implemented on top of an incremental SMT solver. A new blocking clause is pushed after each satisfying assignment. The result is typically [177] stored in a BDD to arrive at a compact representation and allow for fast coverage checks.

Example 1. Given a formula $\vartheta \equiv x = 2 \wedge ((b = x \wedge a > 0 \wedge i = 64) \vee (b = x - 1 \wedge i > 128 \wedge a = 0) \vee (i = 64))$ to abstract using Boolean predicate abstraction with the list of predicates $\bar{\rho} = \langle \rho_1, \rho_2, \rho_3\rangle = \langle b = 1, x = 2, i \leqslant 90\rangle$. We introduce a list of propositional variables $\bar{v} = \langle v_1, v_2, v_3\rangle$, where each of them has a one-to-one correspondence to the predicates in $\bar{\rho}$. The AllSat call returns the assignments $R = \{\langle 0,1,1\rangle, \langle 1,1,0\rangle, \langle 1,1,1\rangle\}$, which results in the Boolean predicate abstraction $\rho_2 \wedge (\rho_1 \vee \rho_3)$.

Cartesian Predicate Abstraction. Another approach for predicate abstraction is *Cartesian predicate abstraction* $\langle\!\langle\vartheta\rangle\!\rangle^\pi_{\mathbb{C}}$ of a given formula $\vartheta \in \mathcal{F}$. It is the conjunction $\psi \in \mathcal{F}$ of all predicates $\in \pi$ that entail ϑ, that is, the abstraction ψ results from:

$$\psi = \bigwedge_{\rho_i \in \pi} \{\rho_i \mid \vartheta \Rightarrow \rho_i\}.$$

This approach does not take dependencies between predicates into account and results in coarse-grained abstractions.

Example 2. Given the formula ϑ and the abstraction precision π from Example 1, the Cartesian predicate abstraction $\langle\!\langle\vartheta\rangle\!\rangle^\pi_{\mathbb{C}}$ results in the formula $x = 2$.

2.5 CONFIGURABLE PROGRAM ANALYSIS

Configurable program analysis (CPA) [31, 32] is a concept, formalism, and framework [34] for formalizing and implementing program analyses. It provides the foundation to combine techniques from different directions of research for reasoning about programs, for example, combinations of techniques from abstract interpretation [80], data-flow analysis [155], symbolic execution [161], and techniques from model checking [71, 73].

56 CPAchecker

The framework was first provided within the tool Blast [41] and was further refined and extended in the tool CPAchecker [34], on which our implementation is based. We slightly modify different operators of the framework, for example, by adding some operators, adding parameters to existing operators, or removing some parameters if they are not relevant to present the contributions of this work. We indicate the modifications where appropriate.

57 ⚠

58 Abstraction Precision

Please note that we build on configurable program analysis *with dynamic precision adjustment* [32]: Different operators and algorithms are extended with an abstraction precision. An *abstraction precision* $\pi \in \Pi$ is a set of program facts to track, and determines the level of abstraction of the abstract model to construct; the set of all abstraction precisions is denoted by Π. An elaborated formal definition of abstraction precisions is provided in Sect. 5.2.

59 Verification Engine

We use the term *verification engine* to denote all components of a verification tool that contribute to the verification process. The set of components includes an SMT solver, a BDD library, different algorithms that wrap the reachability algorithm, for example, a CEGAR loop [67].

2.5.1 CPA Algorithm

60 waitlist *and* reached

The *CPA Algorithm* [8, 31, 32] is the heart of the model-checking procedure that we use. It is a reachability (semi-)algorithm that operates based on a worklist called waitlist, which is a subset of the set of abstract states reached that is considered reached at a given point in time. The algorithm starts the state space exploration starting from an initial worklist waitlist $= W_0$ of abstract states and a set of initially reached states reached $= R_0$, and terminates either if the worklist is empty, that is waitlist $= \emptyset$, or if a target state has been found. A *target state* is an abstract state whose absence or presence—depending on the type of property to check—has to be witnessed by the analysis. All operators, except the operator choose, that are used within the CPA algorithm, are defined within the CPA $\mathbb{D}$—which is an implicit parameter of the algorithm. The operator choose determines the state space traversal strategy, that is, given the set of frontier states waitlist, which state to compute successors for next. The CPA algorithm is typically wrapped by another algorithm, for example, a CEGAR loop. The pairing of abstract states with an abstraction precision was introduced to allow for lazy abstraction [136] and dynamic precision adjustment [32].

61 Target State

62 Operator choose

Algorithm 1 $\mathsf{CPAalg}_{\mathbb{D}}(R_0, W_0)$, adopted from [31, 32]

```
Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec}, \mathsf{target})$,
       a set $R_0 \subseteq E \times \Pi$ of abstract states with precision,
       a subset $W_0 \subseteq R_0$ of frontier abstract states with precision,
       where $E$ denotes the elements of the lattice of $D$
Output: reached abstract states with precision,
       remaining frontier abstract states with precision
Variables: a set $\mathsf{reached} \subseteq E \times \Pi$, a set $\mathsf{waitlist} \subseteq E \times \Pi$
 1: $\mathsf{reached} := R_0$; $\mathsf{waitlist} := W_0$
 2: while $\mathsf{waitlist} \neq \emptyset$ do
 3:   $(e, \pi) := \mathsf{choose}(\mathsf{waitlist})$
 4:   for each $e'$ with $e \rightsquigarrow (e', \pi)$ do
 5:     $(\widehat{e}, \widehat{\pi}) := \mathsf{prec}(e', \pi, \mathsf{reached})$
 6:     for each $(e'', \pi'') \in \mathsf{reached}$ do
 7:       $e_{new} := \mathsf{merge}(\widehat{e}, e'', \widehat{\pi})$
 8:       if $e_{new} \neq e''$ then
 9:         $\mathsf{waitlist} := \big(\mathsf{waitlist} \cup \{(e_{new}, \widehat{\pi})\}\big) \setminus \{(e'', \pi'')\}$
10:         $\mathsf{reached} := \big(\mathsf{reached} \cup \{(e_{new}, \widehat{\pi})\}\big) \setminus \{(e'', \pi'')\}$
11:     if $\neg\, \mathsf{stop}(\widehat{e}, \{e \mid (e, \cdot) \in \mathsf{reached}\})$ then
12:       $\mathsf{waitlist} := \mathsf{waitlist} \cup \{(\widehat{e}, \widehat{\pi})\}$
13:       $\mathsf{reached} := \mathsf{reached} \cup \{(\widehat{e}, \widehat{\pi})\}$
14:       if $\mathsf{target}(\widehat{e}) \neq \emptyset$ then
15:         return $(\mathsf{reached}, \mathsf{waitlist})$
16: return $(\mathsf{reached}, \emptyset)$
```

2.5.2 CPA

The central building block of the CPA framework is the *configurable program analysis* (CPA) which defines, for example, how the concrete states and behaviors of the program under analysis are abstracted, to which extend information from different states is combined, and provide means to check whether a fixed-point has been reached or not.

A *configurable program analysis* (CPA) with dynamic precision adjustment [32] is formalized as a tuple $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec}, \mathsf{target})$ of analysis components (operators):

Abstract Domain D. The abstract domain $D = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ defines the mapping between sets of concrete states and abstract states. The set of concrete elements C corresponds to the concrete states; the set of abstract elements E of the lattice $\mathcal{E}$ corresponds to the abstract states. See Sect. 2.3.3 for detailed definitions and discussions.

Transfer Relation $\rightsquigarrow$. The *transfer relation* $\rightsquigarrow\ \subseteq E \times E \times \Pi$—sometimes called *accessibility relation*—defines the predecessor–successor relation of abstract states, that is, it is the central component that defines the shape of the abstract reachability graph of the program to analyze. The relation is total and must overapproximate the reachable states to not miss potential specification violations. Please note that the general notion of a CPA is independent of the control-flow automaton of the program (or system) to analyze. For analyses that encode the semantics of a control-flow automaton, the *labeled transfer relation* $\overset{g}{\rightsquigarrow}\ \subseteq E \times G \times E \times \Pi$ has been introduced [31], which defines the set of successor states based on a given control-flow transition $g \in G$. For a forwards analysis, the strongest postcondition SP is used to compute the set of successor states for a given abstract state—see Sect. 2.3.4.

OPERATOR stop. The *coverage check* operator $\mathsf{stop} : E \times 2^E \to \mathbb{B}$ determines [31] if a given abstract state is already covered sufficiently by the current set of reached states. A call $\mathsf{stop}(e, R)$ gets as argument the abstract state $e \in E$ to check coverage for, the set of reached states R that might already cover the newly reached state. To be sound—for example, for model checking—we require that $[\![e]\!] \subseteq \bigcup_{r \in R} [\![r]\!]$. Literature defines two standard stop operators [31]: $\mathsf{stop}^{\mathsf{sep}}(e, R) = (\exists r \in R : e \sqsubseteq r)$ and $\mathsf{stop}^{\mathsf{join}}(e, R) = (e \sqsubseteq \bigsqcup_{r \in R} r)$.

OPERATOR merge. The *merge operator* $\mathsf{merge} : E \times E \times \Pi \to E$ determines how and to which extent information of a pair of abstract states should get combined to a new (more abstract) abstract state. It controls if two abstract states should be combined or if they should be explored separately and separate the state space. The default is to always separate two different abstract states. The semantics of the operator also follow from its usage in the CPA algorithm: Given a call $\mathsf{merge}(e, r, \pi)$, the result is computed by widening the second argument r based on the first argument e—that is, the operator is not commutative. To be sound, the operator must ensure that $[\![r]\!] \subseteq [\![\mathsf{merge}(e, r, \pi)]\!]$.

63 $\mathsf{merge}^{\mathsf{join}}$

In a path-insensitive data-flow analysis, the merge operator combines two abstract states based on the join $\sqcup$ of the lattice that is defined along with the abstract domain. This behavior is implemented in the operator $\mathsf{merge}^{\mathsf{join}}(e, r, \pi) = (e \sqcup r)$ [31]. Another merge operator is $\mathsf{merge}^{\mathsf{sep}}$ that always keeps information of two abstract states separated: $\mathsf{merge}^{\mathsf{sep}}(e, r, \pi) = r$, which means that the second argument is not widened [31].

64 $\mathsf{merge}^{\mathsf{sep}}$

OPERATOR prec. The *precision adjustment operator* $\mathsf{prec} : E \times \Pi \times 2^{E \times \Pi} \to E \times \Pi$ [32] can provide a new abstract state with an adjusted abstraction precision. Given a call $\mathsf{prec}(e, \pi, R)$, the operators returns a pair (e', π'), with $[\![e]\!] \subseteq [\![e']\!]$. That is, it adjusts the abstraction precision of the given state e, resulting in a widened state e' and a new abstraction precision π'. An abstraction is, for example, computed based on the abstraction operator $\langle\!\langle \cdot \rangle\!\rangle^{\pi}$ with widening based on the given abstraction precision π. Please note that the precision adjustment operator does not add any requirements on the given abstraction precision π and the returned abstraction precision π'—while such a requirement can help to ensure the progress of the analysis.

OPERATOR target. The *target operator* $\mathsf{target} : E \to 2^{\mathbb{S}}$ determines if a given abstract state is a target state of the analysis process, for example, a state that must not be reachable in the concrete system (safety property). A call $\mathsf{target}(e)$ returns a set of properties $\rho \subseteq \mathbb{S}$ for that the given abstract state might have to be ruled out. The operator returns the empty set $\emptyset$ if the state is not a target state.

2.5.3 Composite CPA

In the framework of configurable program analysis [31] different analyses (CPAs) can be responsible for tracking different facts about the system under analysis. We need a mechanism for composing different analyses, their abstract domains and semilattices, and operators. This mechanism is provided by the *Composite CPA* [31]. A composite CPA $\mathbb{D}_\times$ is composed of a sequence of *component analyses* $\langle \mathbb{D}_1, \dots, \mathbb{D}_n \rangle$. The *composite domain* $D_\times$ is the direct product of the abstract domains of the component analysis domains $\langle D_1, \dots, D_n \rangle$, that is, it operates on *abstract composite states* $E_\times = E_1 \times \dots \times E_n$, which are elements of the *product lattice* $\mathcal{E}_\times = \mathcal{E}_1 \times \dots \times \mathcal{E}_n$.

Strengthening. Whenever an analysis runs as a component analysis within a composite analysis, it can implement an additional strengthening operator $\downarrow$. The operator is invoked for each component analysis $c_1, \dots, c_n$ separately, after all of them have computed a successor state based on their transfer relation, which resulted in the composite state $e'_\times = (e_1, \dots, e_n)$. Compared to the original formalism [31, 32], we have extended the strengthening operator for our needs. The remaining details about this analysis can be found in the literature [31, 32].

Operator $\downarrow$. The *strengthening operator* $\downarrow : E_\times \times E_\times \times E_{c_i} \rightarrow E_{c_i}$ can be implemented in a component analysis to use information from other abstract states in the composite state to strengthen the component state. A call $\downarrow_i(e_\times, e'_\times, e_{c_i})$ takes as argument the predecessor composite state $e_\times \in E_\times$, the successor composite state $e'_\times \in E_\times$, and the abstract component successor state $e_{c_i} \in E_i$, and returns a strengthened successor state $e_{c'_i} \in E_i$. The strengthening process can take information from $e_\times$ and $e'_\times$ into account.

2.5.4 Predicate CPA

We use the predicate abstraction with adjustable block encoding [35] to abstract the heap and stack of the program under analysis. The functionality of this analysis is implemented in the *Predicate CPA* $\mathbb{D}_\mathcal{P}$. We now describe those details of the analysis that are relevant for this work. Please refer to the literature [35] for a full description of this analysis.

Abstract Domain $D_\mathcal{P}$. The abstract domain $D_\mathcal{P} = (C, \mathcal{E}, [\![\cdot]\!])$ is defined based on a semilattice $\mathcal{E}$ on the set of abstract states $E = \mathcal{F} \times \mathcal{F} \times L$. An abstract predicate state $e = (\varphi, \psi, l^\psi) \in E$ consists of a *block formula* φ, an *abstraction formula* ψ (summary), and an *abstraction location* $l^\psi \in L$. The abstraction formula is typically stored in a BDD [177], which allows fast entailment checks on the level of propositional variables, and simplifies the formulas.

Details on the semilattice $\mathcal{E}$ can be found in the literature [35]. Since we use the Predicate CPA (and this abstract domain) in a composite analysis along with an analysis that keeps care of tracking the current position in the transition relation of the system under analysis, we do not model the current control location as part of this domain—as done in existing work [35].

Operator $\mathsf{prec}_\mathcal{P}$. We use the precision adjustment operator $\mathsf{prec}_\mathcal{P} : E \times \Pi \times 2^{E \times \Pi} \rightarrow E \times \Pi$ to compute predicate abstractions, at specific control locations—called the abstraction locations.

65 Block Operator

The *block operator* $\mathsf{blk} : E \times L \times \mathbb{B}$ is used to determine the points in the state space for that an abstraction (summary) should be computed based on a given abstraction precision (set of predicates). The block operator blk_L returns *true* iff the given control location is a loop head, the block operator blk_{LF} returns *true* iff the given control location is a function call or a loop head, and the operator blk_{SBE} returns *true* always, that is, fosters a single-block encoding [21, 28]. Cartesian predicate abstraction is used to compute the abstraction in case single-block encoding [21] is enabled; otherwise, Boolean predicate abstraction [177] is used. Please note that the original analysis [35] computes abstraction in the transfer relation of the analysis.

Given an abstract state $e = (\varphi, \psi, l^{\psi})$ and a set of predicate π for a control location $l \in L$, the operator $\mathsf{prec}_{\mathcal{P}}$ returns (e, π) if $\mathsf{blk}(e, l) = \mathit{false}$, otherwise, it returns $((\mathit{true}, \langle\langle \psi \wedge \varphi \rangle\rangle^{\pi}, l), \pi)$. Depending on the block operator, a different abstraction function $\langle\langle \cdot \rangle\rangle^{\pi}$ with widening is used to conduct either a Cartesian predicate abstraction or a Boolean predicate abstraction.

3 ABSTRACT TRANSDUCERS

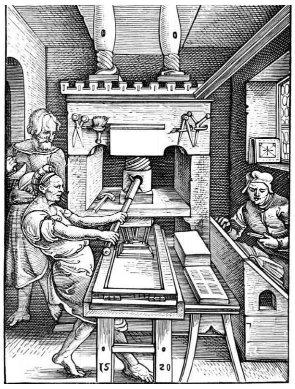


Figure 4: Printing press: A means to reproduce and share information

Keywords: Abstract Transducers, Transducer Abstraction, Sharing and Reuse, Task Artifacts, Epsilon Closure, Closure Abstraction

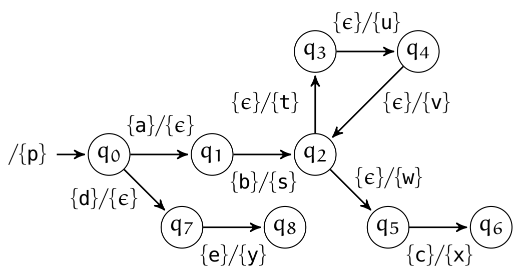
In this chapter, we present a new type of abstract machine, which we use as the generic conceptual and technical foundation for sharing task artifacts for reuse and reproduction, within and among verification runs, both for constructing and composing syntactic and semantic task models—see the motivation of this thesis in Sect. 1. Having such a generic concept—and corresponding techniques—for sharing at hand can reduce the complexity of analysis tools, and the toolchain it is used in, considerably; it reduces the number of concepts to deal with, and allows to build on generic and well-tested implementations.

66 *Common Foundation Needed*

Over time, several techniques for sharing task artifacts, of different types, for reuse have been presented. These techniques are used, for example, to store and share reachability graphs [30, 33, 135], summaries for distinct functions [233], sets of predicates [122], cross-cutting concerns—which are shared as aspect—to weave [160], or specifications that are applicable among program variants, which are stored in separate specification files for reuse [22, 39]. These techniques use different concepts, mechanisms, and data structures to depict the process of sharing. All of them have to deal with the structure of computer programs, and changes thereof in case they evolve and reuse from other revisions or variants is intended. Not only the complexity of a tool is increased by implementing all these techniques without a generic foundation, but also common ideas are hidden behind different formalisms: Gaining a deep understanding of the fundamental ideas is hindered. Without reducing the differences between concepts, techniques, and their implementations to the necessary minimum, the number of threats that are imposed on the internal validity of empirical studies is increased.

67 *Transducers*

We propose a form of transducers as generic means for sharing artifacts of different types for reuse. In general, a *transducer* is a mechanism for transforming information between different information carriers, for example, be-



Input	Output
ϵ	p
a	p
de	py
ab	ps(tuv)*w
abc	ps(tuv)*wx

Figure 5: Each transition of an abstract transducer is annotated with an abstract input word $\mathfrak{v}$ and an abstract output word $\mathfrak{w}$. An abstract input word denotes $[\![\mathfrak{v}]\!] \subseteq \Sigma^*$ a set of words over an input alphabet Σ, an abstract output word denotes $[\![\mathfrak{w}]\!] \subseteq \Theta^\infty$ a set of words over an output alphabet Θ; the illustration shows the sets of concrete words. Please note that already the activation of the initial state q_0 emits an abstract output word—in this case: the set {p} of concrete words. We use the abstract epsilon word $\mathfrak{v}_\epsilon$, with $[\![\mathfrak{v}_\epsilon]\!] = \{\epsilon\}$, with the semantics that is known from ϵ-NFAs [143, 237]. Transitions with the abstract input word $\mathfrak{v}_\epsilon$ are called ϵ-moves and can result in output words of (here countable) infinite length, which result from ϵ-loops—for example, the loop $q_2 \to q_3 \to q_4 \to q_2$. The table on the right shows—for the transducer on the left—a set of input words and corresponding output words in the form of regular expressions.

tween different forms of energy [1, 209]; it might carry information on its own, which it can emit under specific conditions. It can be used as a mechanism for *reproducing and sharing information*, possibly on a different form of information carrier. Examples for transducers in the real world include microphones [188], which transduce between sound (mechanical energy) and electrical energy, or a printing press, which transduces between a printing plate and paper—one form of sharing a large amount of complex information with a broad audience. 68 *Finite State Machines* The transducers that we propose for sharing artifacts for reuse are finite-state machines (automata) and map between words that are constructed based on different alphabets. Automata are a fundamental computational model for computer science; an automaton typically defines or classifies a set of words. Finite-state transducers are an extension of automata that also produce outputs—on their states or on transitions between states; a transducer maps between words from different alphabets. For our application, the input words describe execution traces of verification tasks, that is, the input alphabet consists of control-flow transitions or program operations; the output words are formed based on the artifacts to share. Figure 5 illustrates the general working principle of the type of transducers we propose. By using such transducers as means for sharing artifacts for reuse, we *gain precise control* over the sharing process: We can precisely specify at which points and in which context (path prefix), of the control flow of a program, certain artifacts should be shared for reuse.

69 *Abstract Transducers* We present *abstract transducers* as a new type of abstract machines that operate on an abstract input alphabet and an abstract output alphabet, and that have an inherent notion of abstraction. Both the input alphabet and the output alphabet are described based on abstract domains, which enables different forms of abstracting these transducers and allows for different forms of *symbolic* representations. An abstract representation of words is essential for

(1) creating *finite abstractions* of possibly exponentially many and infinitely long output words, and (2) abstraction of a transducer allows to increase the *sharing* of its outputs, that is, one output becomes applicable to a wider set of input words. Different abstract domains, with corresponding lattices, have been proposed to represent and abstract states and behaviors of systems and their relationships [80]. An abstract domain provides means to map between abstract and concrete entities—it defines a Galois connection. Combining abstract domains and finite-state transducers results in a generic formalism that (1) provides a *unified view* on different types of automata and transducers, and (2) enables *new applications* in different areas, for example, in program analysis and verification.

70 *Transducers and Abstract Domains*

71 *Addressed Problems*

72 *Alphabets*

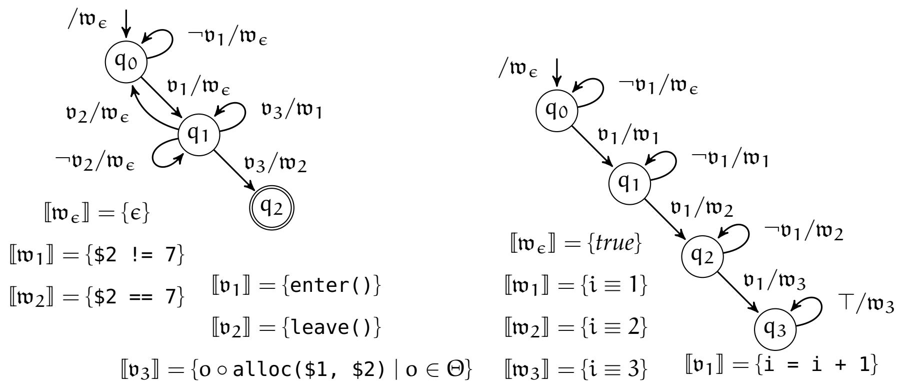
Abstract transducers address several problems: (1) In case alphabets consist of many, possibly exponentially many, symbols, traditional automata concepts with single concrete symbols per transitions provide limited efficiency. Automata that employ a symbolic alphabet—where one symbol from the alphabet denotes a set of concrete symbols—solve this issue [206, 244]. Having a symbolic representation of alphabet symbols makes approaches for abstracting (or widening) finite-state machines—such as relational abstraction or alphabet abstraction [58, 217]—applicable. We use abstract domains, as known from abstract interpretation, for constructing symbolic representations, and mapping between concrete and symbolic alphabets. This way, we can choose from a large variety of abstract domains to provide different symbolic and explicit mechanisms for representing data, for example, binary decision diagrams [56], predicates [21, 122], or polyhedra [236]. Abstraction is also essential for output words, which are produced by transducers, and has not yet received attention by researchers. (2) We allow the transducers to have ϵ-moves that are annotated with outputs, which can lead to output words of *infinite length*; here, a symbolic representation of sets of output words, based on corresponding abstract domains for the output alphabet, can help to provide a finite representation that represents or even overapproximates sets of exponentially many and infinitely long words. By having a means for abstracting both the input alphabet and the output alphabet, we can implement further, more elaborated techniques with various applications. We abstract our transducers to *increase the sharing* of the output they emit. An abstract transducer might have been constructed to produce its output for a specific set of input words that can be found in a specific analysis task, that is, (3) the reuse of the output can be limited to a specific set of analysis tasks, while the output would also be applicable to a broader set of tasks. Sharing is increased if a given output word becomes produced for a larger set of input words—that is, we take advantage of the nondeterminism that abstraction introduces [14]. The alphabets from which these words can be composed of can (in general) consist of arbitrarily complex entities (symbols), for example, tuples of concrete letters as used for multi-track automata [58]. (4) Nevertheless, also for these complex symbols, a means of abstraction is needed. Constructing complex alphabets, and words thereof, based on abstract product domains [79] addresses this issue.

73 *ϵ-Moves*

74 *Means for Sharing*

75 *Task Artifact Transducers*

We instantiate abstract transducers as task artifact transducers. A *task artifact transducer* is an abstract transducer that maps between a set of control paths of a given program to analyze and a set of task artifacts, which are


$/\mathfrak{w}_\epsilon$
$\neg\mathfrak{v}_1/\mathfrak{w}_\epsilon$
q_0
$\mathfrak{v}_1/\mathfrak{w}_\epsilon$
$\mathfrak{v}_2/\mathfrak{w}_\epsilon$
$\mathfrak{v}_3/\mathfrak{w}_1$
q_1
$\neg\mathfrak{v}_2/\mathfrak{w}_\epsilon$
$\mathfrak{v}_3/\mathfrak{w}_2$
q_2
$[\![\mathfrak{w}_\epsilon]\!] = \{\epsilon\}$
$[\![\mathfrak{w}_1]\!] = \{\texttt{\$2 != 7}\}$
$[\![\mathfrak{w}_2]\!] = \{\texttt{\$2 == 7}\}$
$[\![\mathfrak{v}_1]\!] = \{\texttt{enter()}\}$
$[\![\mathfrak{v}_2]\!] = \{\texttt{leave()}\}$
$[\![\mathfrak{v}_3]\!] = \{o \circ \texttt{alloc(\$1, \$2)} \mid o \in \Theta\}$
$/\mathfrak{w}_\epsilon$
$\neg\mathfrak{v}_1/\mathfrak{w}_\epsilon$
q_0
$\mathfrak{v}_1/\mathfrak{w}_1$
$\neg\mathfrak{v}_1/\mathfrak{w}_1$
q_1
$\mathfrak{v}_1/\mathfrak{w}_2$
$\neg\mathfrak{v}_1/\mathfrak{w}_2$
q_2
$\mathfrak{v}_1/\mathfrak{w}_3$
$\top/\mathfrak{w}_3$
q_3
$[\![\mathfrak{w}_\epsilon]\!] = \{true\}$
$[\![\mathfrak{w}_1]\!] = \{i \equiv 1\}$
$[\![\mathfrak{w}_2]\!] = \{i \equiv 2\}$
$[\![\mathfrak{w}_3]\!] = \{i \equiv 3\}$
$[\![\mathfrak{v}_1]\!] = \{\texttt{i = i + 1}\}$

(a) YARN Transducer **(b)** Precision Transducer

Figure 6: Two types of abstract transducers are illustrated: A YARN transducer that emits code to weave, that is, it corresponds to an aspect in AOP, and a precision transducer, which is a means to share candidate invariants, or predicates, for (re-)use in predicate abstraction. Please note that the abstract input word $\mathfrak{v}_3$ describes a lookahead, that is, it contains a word $\bar{\sigma} \in [\![\mathfrak{v}]\!]$ with $|\bar{\sigma}| > 1$. The lookahead matches if the input that remains to be consumed after word $\mathfrak{v}_3$ matched starts with `alloc($1,$2)`, where `$1` and `$2` are parameters that bind the arguments that are given to the function `alloc` to internal variables, which are then used to produce the concrete output for the abstract output words $\mathfrak{w}_1$ and $\mathfrak{w}_2$.

intended to be shared for reuse. We use task artifact transducers as a *generic means to provide information that contributes to an analysis task and its solution*. Task artifact transducers are an artifact sharing model—see Chapter 1 for the idea of sharing models. These task artifact transducers aid in various analysis tasks for that task artifacts, for example, intermediate verification results, have to be provided at specific points and in specific contexts in the control flow. We use them both to construct the transition relation of the analysis task itself, and for constructing a state-space abstraction with a finite number of abstract states in an efficient and effective manner, that is, for sharing syntactic and semantic task artifacts. Syntactic task artifacts include, for example, components, aspects, or assertions to check; semantic task artifacts include, for example, function summaries, invariants, or Craig interpolants.

76 *Application Scenarios*

The chapters that follow do instantiate the generic concept of abstract transducers and use them as the foundation for concrete analysis techniques: YARN transducers and the corresponding YARN analysis (Chapter 4), precision transducers and a precision transducer analysis (Chapter 5). Figure. 6 provides examples for these types of transducers. A YARN *transducer* can express aspects—source code, or labeled transition systems (LTSs) in general, to emit at specific points—to weave into a control-flow graph (or an LTS in general). Such aspects can, for example, provide the environment model or a specification. It must be possible to emit code to weave before any of the transitions that are processed as input: An initial transducer output is needed. For soundness, operations such as epsilon-elimination, union, or reduction must keep the semantics—including their temporal relationships, also concurrency—of these aspects. A *precision transducer* is annotated with sets of predicates (candidate invariants) to emit for reuse in different contexts

77 *YARN Transducer*

of the transition system to construct (for example, a Kripke structure) in an analysis process. The shared predicates can be used to compute predicate abstractions (as used for software model checking [20, 122]), the number of CEGAR [36, 67] iterations can be reduced by abstracting these transducers, which increases sharing (the same predicate can be emitted in more contexts). Such precision transducers can also express the predicate sharing strategy of lazy abstraction [136].

78 Precision Transducer

This chapter presents the following contributions:

- We introduce *abstract transducers* as a generic and unifying type of abstract machines that use *abstract word domains* to characterize both the input alphabet and the output alphabet, and that have an inherent notion of abstraction.
- We present techniques for computing *finite abstractions of the output* of ϵ-closures with ϵ*-loops* that are possible in transducers that allow ϵ-moves. These techniques allow to produce finite outputs from transducers with outputs that describe exponentially large sets of potentially infinitely long words, and they aid in eliminating the ϵ-moves.
- We present an *abstract transducer analysis* as a generic configurable program analysis for running different types of abstract transducers.
- We instantiate abstract transducers as *task artifact transducers* to have a generic means to share various artifacts that contribute to different concerns of an analysis task. Task artifact transducers foster sharing and reuse of components of an analysis task and the intermediate analysis (reasoning) results that are produced while conducting an analysis.

3.1 ABSTRACT WORDS

Before we present abstract transducers, we describe concepts to cope with sets of possibly exponentially many and infinitely long words symbolically. A word can *express temporal or causal relationships* between the letters of the word. We introduce concepts and techniques to deal with sets of words on an abstract level.

3.1.1 Hierarchy of Characters, Words, and Languages

We now discuss established terms that are relevant in the context of the terms that we introduce in the following sections. This helps to understand our terminology choices.

Both the input alphabet and the output alphabet of an abstract transducer is characterized based on an abstract domain. *Abstract domains* are a generic means for abstraction and provide various operations for manipulating and comparing abstract elements (entities) [80], and for mapping between *concrete and abstract elements*—see Sect. 2.3.3 for a formal definition.

79 Abstract Domain

Elements from a set Σ can be combined to form (possibly infinitely long) sequences $\bar{\sigma} \in \Sigma^\infty$ of those elements. We use the term *word* to denote sequences of elements that can be formed from other words by concatenation. Words are elements of a free monoid (semigroup) for that concatenation is the binary and associative operator, and the empty word (empty sequence) is the identity (neutral) element. A *language* is a set of words—and typically well-formed regarding some production rules.

80 *Words and Languages*

81 *Concrete and Abstract*

In a generic abstract domain, one *abstract element* maps to a set of *concrete elements*, which is reflected by the denotation (concretization) function $[\![\cdot]\!]$. That is, we can deduce that one *abstract word* represents a set of *concrete words*, and an *abstract language* maps to a set of *concrete languages*.

82 *Temporal*

A word, as mentioned earlier, establishes a *temporal relationship* between all its characters; each character has a semantic denotation on its own, that is, it maps to a set of entities. The expressiveness of words compared their characters is dual to the expressiveness of linear temporal logic to propositional logic: A formula in propositional logic (interpreted for a specific universe) denotes a set of entities, whereas a formula in linear temporal logic denotes sequences of sets of entities (over time). A *set of words*, that is, a *language*, provides sufficient expressiveness to describe a set of forks in words over time, for example, to describe a set of *concurrent* program executions, or for matching trees or (more general) graphs.

83 *Linear Time*

84 *Branching Time*

85 *Abstract Words and Languages*

That is, an abstract word, which maps to a set of concrete words, provides an abstraction with sufficient expressiveness to describe sets of linear-time concerns, and an abstract language, which represents a set of sets of words, provides expressiveness to describe sets of concerns that are expressible in branching-time logic. In the following, we restrict the discussion and presentation to abstract words and keep abstract languages for future work.

86 ⚠

3.1.2 Abstract Word Domain

The foundation of abstract transducers is formed by the abstract word domain, a lattice-based abstract domain [80, 104] for mapping between abstract words and concrete words.

Definition 1: Abstract Word

An *abstract word* $\mathfrak{v} \in \mathfrak{I}$ is a symbolic representation of a set $\subseteq \Sigma^\infty$ of concrete words over a concrete alphabet Σ, where the set $\mathfrak{I}$ denotes all abstract words.

The relationship between an abstract word and the set of concrete words it represents, along with a means for abstraction, is defined by the abstract word domain:

Definition 2: Abstract Word Domain

An *abstract word domain* is an abstract domain $D_W = (\mathsf{pw}(\dddot{W}), \dddot{\mathfrak{I}}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ that has abstract words $\mathfrak{I}$ as its abstract elements. The relationship between abstract words is defined based on the *abstract word lattice* $\ddot{\mathfrak{I}} = (\mathfrak{I}, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$. One abstract word $\mathfrak{v}$ maps to a set of concrete words $[\![\mathfrak{v}]\!] \subseteq W$, which is defined by the denotation function $[\![\cdot]\!] : \mathfrak{I} \to 2^W$. The lattice of concrete words $\ddot{W}$ defines the relationship between elements from the set of concrete words W. Sets of concrete words are formed based on a powerset lattice $\mathsf{pw}(\ddot{W})$. The abstraction function $\langle\!\langle\cdot\rangle\!\rangle : 2^W \to \mathfrak{I}$ transforms a given set of concrete words $\widehat{w} \subseteq W$ into an abstract word $\mathfrak{v}$, that is, $\mathfrak{v} = \langle\!\langle\widehat{w}\rangle\!\rangle$. The *abstract epsilon word* $\mathfrak{v}_\epsilon$ maps $[\![\mathfrak{v}_\epsilon]\!] = \{\epsilon\}$ to the set with the empty word ϵ only. The *bottom element* $\bot$, or also *abstract bottom word*, of the abstract word lattice denotes an abstract word that maps to the empty set of concrete words, that is, $[\![\bot]\!] = \emptyset$.

87 Role of Abstraction

The abstraction mechanism that is provided by the abstract word domain is important for (1) constructing finite abstractions of collections with exponentially many or infinitely long words; it can be used to (2) check whether or not the analysis process ran into a fixed point, and (3) for increasing the sharing of the output that we produce based on abstract transducers.

88 Word Coverage Problem

A problem that we have to deal with is the *word coverage problem*, that is, the question of whether or not a given abstract word $\mathfrak{v}_a$ is covered by another abstract word $\mathfrak{L}_b$, that is, if $\mathfrak{v}_a \sqsubseteq \mathfrak{v}_b$, where $\sqsubseteq$ is the inclusion relation of the abstract word lattice. The actual matching process, that is, the check for coverage can be implemented based on quotienting:

89 Quotienting

The abstract word domain must provide the possibility to compute left quotients [57] (Brzowzowski derivates) to match abstract words.

Definition 3: Left Quotient

The *left quotient* [57] $\mathfrak{v}^{\mathfrak{w}} : \mathfrak{I} \times \mathfrak{I} \to \mathfrak{I}$ of an abstract word $\mathfrak{v} \in \mathfrak{I}$ regarding an abstract word $\mathfrak{w} \in \mathfrak{I}$ is defined as $\mathfrak{v}^{\mathfrak{w}} = \langle\!\langle\{\bar{s} \mid \bar{p} \circ \bar{s} \in [\![\mathfrak{v}]\!] \wedge \bar{p} \in [\![\mathfrak{w}]\!]\}\rangle\!\rangle$. It denotes suffixes of $\mathfrak{v}$ for that $\mathfrak{w}$ contains prefixes.

Another fundamental operation when dealing with words is their concatenation, which is the binary operator of the free monoid Σ^* that describes the set of words over an alphabet Σ. We extend this operator to abstract words, and with it to sets of words:

Definition 4: Concatenation

The *concatenation* of a pair of abstract words $\mathfrak{v}_1 \circ \mathfrak{v}_2$ results in an abstract word $\mathfrak{v}_\circ$ that denotes $[\![\mathfrak{v}_\circ]\!]$ the concatenation of all concrete finite words from the abstract word $\mathfrak{v}_1$ with all (finite and infinite) concrete words from the abstract word $\mathfrak{v}_2$. The concatenation $\bar{\sigma}_1 \circ \bar{\sigma}_2$ of an infinite word $\bar{\sigma}_1$ with another word $\bar{\sigma}_2$ results in the infinite word $\bar{\sigma}_1$. That is $[\![\mathfrak{v}_1 \circ \mathfrak{v}_2]\!] = \{\bar{\sigma}_1 \circ \bar{\sigma}_2 \mid \bar{\sigma}_1 \in [\![\mathfrak{v}_1]\!] \wedge \bar{\sigma}_2 \in [\![\mathfrak{v}_2]\!]\}$.

To deal with abstract words, the notion of head and tail is important:

Definition 5: Head

Given an abstract word $\mathfrak{v}$, the function $\mathsf{head}(\mathfrak{v}) : \mathfrak{I} \to \mathfrak{I}$ denotes the *head of an abstract word*: The resulting abstract word represents the set of prefixes with length one, or formally $[\![\mathsf{head}(\mathfrak{v})]\!] = \{\,\bar{h} \mid \bar{h} \circ \bar{\sigma} \in [\![\mathfrak{v}]\!] \wedge |\bar{h}| = 1\,\}$.

Definition 6: Tail

The *tail of an abstract word* is provided by the function $\mathsf{tail}(\mathfrak{v}) : \mathfrak{I} \to \mathfrak{I}$. A call $\mathfrak{v}' = \mathsf{tail}(\mathfrak{L})$ returns a new abstract word $\mathfrak{v}'$ that represents the set of postfixes that follow after the head. That is, $[\![\mathsf{tail}(\mathfrak{v})]\!] = \{\,\bar{\sigma} \mid \bar{h} \circ \bar{\sigma} \in [\![\mathfrak{v}]\!] \wedge |\bar{h}| = 1\,\}$, which equals $\mathsf{tail}(\mathfrak{v}) = \mathfrak{v}^{\mathsf{head}(\mathfrak{v})}$.

3.1.3 Boolean Algebra

90 *Boolean Algebra*

On several occasions, when reasoning about abstract words and their relationship, we need the full expressive power of a Boolean algebra. We can build on the duality between Boolean algebras, regular languages, and complemented and distributive lattices, which follows from the Stone duality [213, 240]. The abstract word lattice is dual to a Boolean algebra if and only if its meet $\sqcap$ and join $\sqcup$ are distributive over each other and if each element in the lattice has a complement within the lattice. One example of a lattice that is dual to a Boolean algebra is the powerset lattice and another one the lattice of regular languages [52, 116]. Both lattices can describe sets of words and can thus be instantiated as an abstract word lattice of an abstract word domain. Given a lattice of regular expressions, the join $\sqcup$ corresponds to the language union, the meet $\sqcap$ to the language intersection, and the operator $\sqsubseteq$ describes the language inclusion; the language is complemented since the complement of a regular expression is still regular.

Definition 7: Abstract Word Complement

Given an abstract word $\mathfrak{v}$, its complement word $\neg\mathfrak{v}$ defines a set of concrete words such that $\forall \mathfrak{v} \in \mathfrak{I} : \neg\mathfrak{v} \sqcap \mathfrak{v} = \bot$ and $\forall \mathfrak{v} \in \mathfrak{I} : \neg\mathfrak{v} \sqcup \mathfrak{v} = \top$, with $\forall \bar{a} \in [\![\mathfrak{v}]\!] : \forall \bar{b} \in [\![\neg\mathfrak{v}]\!] : \bar{a} \sqcap_c \bar{b} = \bot_c$, where $\sqcap_c$ and $\bot_c$ are components of the concrete word lattice.

In case an abstract word lattice is dual to a Boolean algebra, the abstract words and their composition, can also be described using Boolean operators, which have their duals in lattice theory: The join $\sqcup$ corresponds to the logical disjunction $\bigvee$, the meet $\sqcap$ corresponds to the conjunction $\bigwedge$, and the complement corresponds to the logical negation $\neg$. A Boolean formula ϑ is equivalent to an abstract word $\mathfrak{v}$ if and only if $[\![\vartheta]\!] = [\![\mathfrak{v}]\!]$.

3.1.4 Parameterized Words

An abstract word can be *parameterized* with a finite set of parameters $\beta \subseteq \mathcal{B}$. A parameterized abstract word can take two roles: It (1) can *capture* (bind) values to the parameters during a matching process for a given input, and (2) values for the arguments can get passed explicitly (and act as a *template*). We use the term *instantiation* to denote the process of deriving an abstract word $\mathfrak{v}'$ from an abstract word $\mathfrak{v}$ by assigning values to the parameters, with $[\![\mathfrak{v}']\!] \subset [\![\mathfrak{v}]\!]$. Examples for different types of templates words include invariant templates [166, 238]. The values that have been bound to the parameters of an abstract word are provided by the operator $\mathsf{bounded} : \mathfrak{I} \to 2^{\mathcal{B} \to \mathcal{V}}$. We can bind values to parameters of an abstract word and derive a new abstract word with the operator $\mathsf{bind} : \mathfrak{I} \times 2^{\mathcal{B} \to \mathcal{V}} \to \mathfrak{I}$. Binding of values to parameters (variable binding) was extensively studied in the past, for example, for rewriting systems [125, 205], and regular expressions [110].

3.2 ABSTRACT TRANSDUCERS

This work introduces abstract transducers, a type of abstract machines that map between abstract input words and abstract output words. Compared to established transducer concepts, intermediate languages are central (we still have a notion of accepted language): Informally speaking, the *intermediate input language* is the set of words for which the transducer can perform state transitions, and the set of words that are produced as output along these transitions is called the *intermediate output language*.

91 *Intermediate Language*

A *transducer* is an established concept for transforming a signal $s \in S$ of one type S, to another signal $d \in D$ of a type D [1, 209]. The term *transduction* denotes the process of transforming between two signals. Transductions might result in a *loss of precision*. A transducer can carry information on its own and encode it into the output signal. Compared to a compiler, a transducer does not guarantee to preserve the semantics between words of the input language and words of the output language.

92 *Transducers.*

To produce the intermediate output language, an abstract transducer operates *prescient*, that is, it can take a lookahead into account to decide whether to conduct a state transition or not—and with it produce an output. Words from the intermediate output language are intended to be used immediately, that is, as soon as they are produced while executing the transducer, which has several implications on the design on the algorithms that execute abstract transducers and that manipulate them—for example, to eliminate ϵ-moves.

93 *Prescient*

Both the input language and their output language are abstract and defined based on abstract word domains. One abstract word maps to a set of concrete words; the abstract domain provides means for mapping between these representations. This abstraction functionality enriches the possibilities to compute abstractions (widenings) of abstract transducers, which we use as a *means of increasing the scope of sharing*: one output language is mapped to a larger input language.

Each transition of an abstract transducer is annotated with an abstract input word and an abstract output word—which corresponds to symbols of the input alphabet and the output alphabet of traditional transducers. Consuming and producing abstract words instead of single concrete letters has several advantages that increase the generality of our approach: (1) it can be used for lookahead-matching, that is, instead of describing the input symbol to consume, also a sequence of symbols that must follow can be described, (2) the abstract epsilon word $\mathfrak{v}_\epsilon$, with $[\![\mathfrak{v}_\epsilon]\!] = \{\epsilon\}$, can be used to model the behavior of an ϵ-NFA [237] with a corresponding ϵ-closure and to model automata that do not produce outputs at all, and (3) relying on abstract words allows to produce and cope with output words of infinite length, which can be the result of ϵ-loops.

Formally, we define an abstract transducer as:

Definition 8: Abstract Transducer

An *abstract transducer* $\mathsf{T} \in \mathbb{T}$ is defined by following tuple:

$$\mathsf{T} = (Q, D_{in}, D_{out}, \iota_0, F, \delta)$$

- Control States Q. The finite set Q defines the *control states* in which the transducer can be in.
- Abstract Input Domain D_{in}. The *abstract input domain* is an abstract word domain that maps between abstract words $\mathfrak{I}$ and concrete words over the concrete input alphabet Σ. It provides a denotation function $[\![\cdot]\!]_{in} : \mathfrak{I} \to 2^{\Sigma^*}$ to map between an abstract word and a set of concrete (and finite) words. We assume the lattice of abstract words to be distributive and complemented, that is, to be dual [240] to a *Boolean algebra*. An abstract domain with lattice-valued regular expression [195] would be an example of an abstract input domain.
- Abstract Output Domain D_{out}. The *abstract output domain* is an abstract word domain that defines the abstract output words $\mathfrak{W}$ and their relationship. Its denotation function $[\![\cdot]\!]_{out} : \mathfrak{W} \to 2^{\Theta^\infty}$ maps between an abstract output word and the corresponding set of concrete output words over the concrete output alphabet Θ. An instance of an abstract output domain could, for example, use antichains [3] for word inclusion checks.
- Initial Transducer State $\iota_0 \in 2^{Q \to \mathfrak{W}}$. The (non-empty) map ι_0 characterizes the *initial transducer state*. The pairing of control states with outputs is needed, since already the transitions that leave the initial state can be ϵ-moves that are annotated with an output, and it must be possible to eliminate those moves without affecting the semantics of the transducer.
- Final Control States $F \subseteq Q$. The set F defines the final (accepting) control states. This set can be empty, for example, if the transducer is not intended to operate as a classical acceptor, that is, if the focus is on the intermediate languages.
- Transition Relation $\delta \subseteq \Delta$. The *transition relation* defines the set of transducer transitions that are possible between the different control states. Given a *transducer transition* $(q, \mathfrak{v}, q', \mathfrak{w}) \in \delta$, with $\Delta = Q \times \mathfrak{I} \times Q \times \mathfrak{W}$, both the abstract *transition input word* $\mathfrak{v}$ and the abstract *transition output word* $\mathfrak{w}$ can be the abstract epsilon word, which is used to implement the functionality of an ϵ-NFA. The abstract input word $\mathfrak{v}$ must never be the abstract bottom word, that is, $[\![\mathfrak{v}]\!]_{\bar{\sigma}} \neq \emptyset$. Having the empty word as output explicitly signals that the matching process must stop for the given abstract input word—nevertheless, there can be another transition from the same state q that has an intersecting abstract input word which can cancel out this effect.

The set of all transducers is denoted by $\mathbb{T}$, with the subset $\mathbb{T}_{D_{in} \times D_{out}} \subseteq \mathbb{T}$ of transducers that transduce from words from an abstract input domain D_{in} to those from an abstract output domain D_{out}.

94 *Special Control States* The set of control states Q of an abstract transducer implicitly contains two special states that are entered under certain conditions or are used by algorithms that operate on abstract transducers:

Definition 9: Trap State

The *trap state* or *inactivity signaling state* is a special control state q_π that can be entered to signal that the analysis should continue from that point on, but the transducer will no more contribute to the analysis process. We assume that this state is implicitly present for each transducer, that is, $q_\pi \in Q$ and $(q_\pi, \top, q_\pi, \mathfrak{w}_\epsilon) \in \delta$, with $[\![\mathfrak{w}_\epsilon]\!]_{\text{out}} = \{\epsilon\}$.

The trap state is entered if no more transitions to move are left, but the analysis should still continue from that point on. This state is important for configurations of analyses that track automata or transducers with a non-stuttering semantics, that is, that do not stay in the same state if no transition matches. We define another, similar, control state:

Definition 10: Bottom Control State

A *bottom control state* or *unreachable control state* is a special control state $q_\perp \in Q$ that has no leaving transitions and is not an accepting state, that is, $(q_\perp, \cdot, \cdot, \cdot) \notin \delta$ and $q_\perp \notin F$, while we assume this state to be present for all transducers implicitly in their set of control states Q.

The core of an abstract transducer is its transition relation, which defines the possible transitions between control states and the output to produce on these transitions. The result of a state transition is a new transducer state:

Definition 11: Transducer State

A *transducer state* $\iota \in J$, with $J = 2^{Q \to \mathfrak{W}}$, is map $\iota : Q \to \mathfrak{W}$ from control states to abstract output words. Typically, a transducer state is the result of running the abstract transducer for a given input, starting in the initial transducer state $\iota_0 \in J$.

We formalize abstract transducers as Mealy-style [193] finite-state machines. Nevertheless, also a Moore-style [196] representation is possible:

Definition 12: Moore-style Abstract Transducer

A *Moore-style abstract transducer* is an abstract transducer that emits its outputs not on transitions between control states but *active control states*. That is, it is defined by the tuple

$$T^{\text{Moore}} = (Q, D_{\text{in}}, D_{\text{out}}, Q_0, F, \delta, \lambda).$$

This form of abstract transducer has a control transition relation $\delta \subseteq Q \times \mathfrak{I} \times Q$ and uses a state-output labeling function $\lambda : Q \to \mathfrak{W}$ to map abstract output words to control states. Furthermore, this style of abstract transducer has a set Q_0 of initial control states.

A Moore-style abstract transducer allows to represent an abstract reachability graph easily. For this work, we prefer the Mealy-style formalization of abstract transducers because they require fewer states and are fit well for sharing syntactic task artifacts (program fragments for weaving).

After we have defined the components of an abstract transducer, we continue in following subsections with the description of their semantics.

3.2.1 Lookaheads and Graph Matching

Annotating a transition of an abstract transducer with an abstract input word that maps to at least one concrete word that is longer than one letter, specifies a lookahead. The possibility of conducting lookaheads is essential if a transition should produce a particular output only if the remaining word to process has a specific word as its prefix. Consider the following example:

Example 3. Assume that the transducer is in control state $q \in Q$. Given a concrete input word $\bar{\sigma} = \langle \sigma_1, \dots, \sigma_n \rangle \in \Sigma^*$, a transducer transition $(q, \mathfrak{v}, q', \mathfrak{w}) \in \delta$, with $[\![\mathfrak{v}]\!]_{in} = \{\langle x, \text{'e'}, \text{'d'} \rangle \mid x \in \Sigma\}$ will only match if $\sigma_2 = \text{'e'} \wedge \sigma_3 = \text{'d'}$ and will then produce the output $\mathfrak{w}$.

We characterize the lookahead of a transducer transition by a number:

Definition 13: Transition Lookahead

The *lookahead* $\ell(\tau) \in \mathbb{N}_0$ of a transition $\tau = (q, \mathfrak{v}, \cdot, \cdot) \in \delta$ is $\ell(\tau) = 0$ if the input language is either the abstract epsilon word or the abstract bottom word, otherwise it is defined as $\ell(\tau) = \max\{|\bar{\sigma}| \mid \bar{\sigma} \in [\![\mathfrak{v}]\!]_{in}\} - 1$.

The lookahead of an abstract transducer is defined by the maximal lookahead that is conducted on one of its transitions, that is:

Definition 14: Transducer Lookahead

The *lookahead of an abstract transducer* $\ell(T) \in \mathbb{N}_0$ is the maximal lookahead of any of its transition. That is, $\ell(T) = \max\{\ell(\tau) \mid \tau \in \delta\}$, where δ is the transition relation of transducer T.

One can execute an abstract transducer on a rooted and directed graph instead of a particular input word—one word corresponds to a list or a sequence of letters. Each edge of the graph that we match is labeled with a letter. Words are formed by concatenating all letters on the graph edges that get traversed during the matching process, starting from the root node of the graph. Figure 9 provides an intuition of the matching process. In this work, we restrict the graph matching process to disjunctive tree matching:

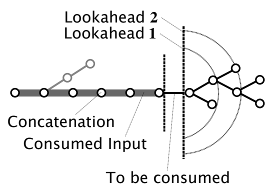


Figure 7: Matching

Definition 15: Disjunctive Tree Matching

A tree matching procedure is called to be *disjunctive* if not several input branches that follow from a particular point on have to satisfy specific criteria. That is, if only *one* of the input words that follow (on that the lookahead is conducted), must satisfy a given criterion.

To allow for matching based on the full expressiveness of regular tree expressions (several of the input words might have to satisfy a specific criterion), the abstract transducer's abstract input domain has to be lifted from an abstract word domain to an abstract *language* domain—see Sect. 3.1.1. We keep abstract transducers with abstract input domains based on abstract languages for future work.

95 *Role of Abstract Languages*

3.2.2 Epsilon Closure

An established practice [143, 237] in automata theory and its application is to use automata with transitions that are annotated with an empty-word symbol ϵ. This was, first and foremost, introduced as a convenience feature to describe automata and its transition relation in a more concise fashion. Abstract transducers allow to annotate transitions with the abstract epsilon word $\mathfrak{v}_\epsilon$ to provide similar semantics and convenience:

Definition 16: ϵ-Move

An ϵ-move (or ϵ-*transition*) is an automaton transition (or transducer transition) $(q, \mathfrak{v}, q', \mathfrak{w}) \in \delta$ that is annotated with the abstract epsilon word $\mathfrak{v}_\epsilon$ as its input, that is, $[\![\mathfrak{v}]\!]_{\text{in}} = [\![\mathfrak{v}_\epsilon]\!]_{\text{in}} = \{\epsilon\}$.

Some algorithms might not be able to deal with transducers that have ϵ-moves—or they might be more sophisticated in their presence—but only with those transducers from that all ϵ-moves were eliminated. We define abstract transducers without ϵ-moves as:

Definition 17: Input-ϵ-Free

An abstract transducer is said to be *input-ϵ-free* if it does not have any transition based on an ϵ-move, that is, $(\cdot, \mathfrak{v}_\epsilon, \cdot, \cdot) \notin \delta$, with $[\![\mathfrak{v}_\epsilon]\!]_{\text{in}} = \{\epsilon\}$.

The presence of ϵ-moves can lead to loops thereof, which is vital for expressing complex outputs, for example, to describe the control-flow of Turing-complete programs—assuming that each move emits a program operation to conduct as output.

Definition 18: ϵ-Loop

An ϵ-loop is any sequence of ϵ-moves that starts in a control state q_k and could include this control state q_k infinitely often in such a sequence. More formally, an ϵ-loop is a sequence $\bar{\tau} = \langle \tau_1, \dots, \tau_n \rangle \in \Delta^\infty$ of ϵ-moves that is well-founded in the transition relation δ and there exists a transducer transition $\tau_i = (q, \cdot, \cdot, \cdot) \in \bar{\tau}$ for which the source state q is precisely the destination state q' of a transducer transition $\tau_j = (\cdot, \cdot, q', \cdot) \in \bar{\tau}$, with $i \leqslant j$.

From the definition of ϵ-moves follows the definition of the ϵ-closure [237]. Intuitively speaking the ϵ-closure of a control state q is the set of control states that become instantly and simultaneously (parallel) active if state q becomes active.

Definition 19: Epsilon Closure

The *epsilon closure* $\mathsf{epsclosure} : Q \to 2^Q$ of a state $q \in Q$ is the set $\mathsf{epsclosure}(q) \subseteq Q$ of states that can get reached transitively from state q by only following ϵ-moves [237]. The bottom state $q_\perp$ is added if the epsilon closure includes an ϵ-loop from which no control state is reachable with no ϵ-move leaving.

The transition relation of an abstract transducer can contain sequences $\{(q_1, \mathfrak{v}_\epsilon, q_2, \mathfrak{w}_1), (q_2, \mathfrak{v}_\epsilon, q_3, \mathfrak{w}_2)\} \subseteq \delta$ of ϵ-moves but not each control state that is reached within such a sequence might have non-ϵ-moves leaving in the transition relation. We therefore introduce the notion of closure termination states:

Definition 20: Closure Termination States

The *closure termination states* $\mathsf{closureterm} : Q \to 2^Q$ of a given state q are both the states (1) in the epsilon closure $\mathsf{epsclosure}(q)$ from which no ϵ-move leaves and (2) states within the closure that are accepting, that is, $\mathsf{closureterm}(q) = \{q' \mid q' \in \mathsf{epsclosure}(q) \wedge (q', \mathfrak{v}_\epsilon, \cdot, \cdot) \notin \delta\} \cup (\mathsf{epsclosure}(q) \cap F)$.

Each transition between states from an epsilon closure can be mapped to a set of closure termination states:

Definition 21: Termination State Mapping

The *termination state mapping* is a map $\Delta_\Omega : \Delta \to 2^Q$ that maps a given transducer transition to the set of closure termination states that are reachable. Given a control state $q \in Q$, the result is the empty set $\emptyset$ if no ϵ-move leaves state q; it is the bottom state $q_\perp$ if there is not any other termination state.

Since also each transition within an epsilon closure can produce an output, we introduce the notion of *concrete language on termination*. This notion reflects with which output words the different closure termination states can be reached:

Definition 22: Concrete Language on Termination

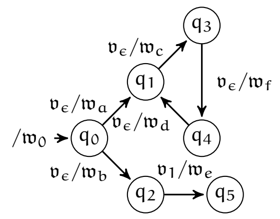
The *concrete language on termination* $\Omega : Q \times Q \to 2^{\Theta^\infty}$ for a given pair (q, q_Ω) describes the concrete output language (a set of concrete words) that can be produced starting in control state q and that terminates with a closure termination state $q_\Omega \in \mathsf{closureterm}(q)$.
More formally, let $\widehat{\tau} = \{\bar{\tau}_1, \dots\} \subseteq \Delta^\infty$ be the set of all well-founded sequences of transducer transitions between control state q and the termination state q_Ω, with $\bar{\tau}_i = \langle \tau_1, \dots \rangle$ and $\tau_i = (q, \mathfrak{v}_i, q', \mathfrak{w}_i) \in \delta$. The concrete output language $[\![\bar{\tau}_i]\!]$ of a sequence $\bar{\tau}_i$ is the concatenation $[\![\mathfrak{w}_1]\!]_{\mathrm{out}} \circ \dots$ of the concretizations of all abstract output words $\mathfrak{w}_i$ that are emitted along it. That is, the concrete output language $\Omega(q, q_\Omega)$ is the union $\bigcup_{\bar{\tau}_i \in \widehat{\tau}} [\![\bar{\tau}_i]\!]$.

Definition 23: Concrete Closure Language

The *concrete closure language* $\Omega(q) \subseteq \Theta^\infty$ of a given control state q and its ϵ-closure is the set of concrete output words that is produced while making transitions along the ϵ-moves between states in the closure. More precisely, it is the join of concrete languages on termination, that is, $\Omega(q) = \bigcup \{\bar{\sigma} \in \Omega(q, q_\Omega) \mid q_\Omega \in \mathsf{closureterm}(q)\}$.

In our applications of abstract transducers, we use the (anonymous) states and transitions in the epsilon closure as a tool for expressing relational outputs. Please note, that also ϵ-moves that lead to a *dead-end* are relevant and *must not be eliminated*—what is done for some applications [87]—because the output might be relevant for the analysis task, and the soundness of the produced result, for which the transducer is executed.

Example 4. Figure 8 illustrates an example transducer: The ϵ-closure of control state q_0 is the set $\mathsf{epsclosure}(q_0) = \{q_0, q_1, q_2, q_3, q_4, q_\perp\}$, for state q_2, the closure $\mathsf{epsclosure}(q_2) = \{q_2\}$ does not contain additional states. State q_0 has the set of closure termination states $\mathsf{closureterm}(q_0) = \{q_2, q_\perp\}$, and state q_1 has $\mathsf{closureterm}(q_1) = \{q_\perp\}$, that is, no other termination state is reachable. The transitions between states $\{q_1, q_3, q_4\}$ form an ϵ-loop.



q_3
$\mathfrak{v}_\epsilon/\mathfrak{w}_c$
q_1
$\mathfrak{v}_\epsilon/\mathfrak{w}_f$
$\mathfrak{v}_\epsilon/\mathfrak{w}_a$
$\mathfrak{v}_\epsilon/\mathfrak{w}_d$
$/\mathfrak{w}_0$
q_0
q_4
$\mathfrak{v}_\epsilon/\mathfrak{w}_b$
$\mathfrak{v}_1/\mathfrak{w}_e$
q_2
q_5

Figure 8: With ϵ-loop

Given a control state $q \in Q$, the semantics of ϵ-moves implies that with reaching state q, actually all states in $Q_t = \mathsf{closureterm}(q)$ are reached immediately. That is, also all output on the transitions from q to a state in Q_t is produced immediately, resulting in—possibly exponentially many and infinitely long—words $\subseteq \Theta^\infty$ over the output alphabet Θ.

3.2.3 Output Closure

Previous section describes the epsilon closure of abstract transducers; in contrast to established transducer concepts, we also address ϵ-moves that are annotated with non-empty outputs, and use them as tool to express complex output languages, with possibly exponentially many and infinitely long words, in a convenient fashion. When executing or reducing (minimizing) abstract transducers, means for collecting, aggregating, and possibly abstracting the output on these transitions are needed. Given a control state $q \in Q$, the goal of this summarization process is to provide an abstract output word $\mathfrak{w}_\Omega \in \mathfrak{W}$ for each of its closure termination states $q_\Omega \in \mathsf{closureterm}(q)$ that overapproximates the concrete closure language, that is, $\Omega(q, q_\Omega) \subseteq [\![\mathfrak{w}_\Omega]\!]_{\text{out}}$—which *can* lead to a loss of information. The computation of this closure is done in a corresponding operator:

Definition 24: Abstract Output Closure

The *abstract output closure* of a given control state $q \in Q$ is a finite overapproximation of the concrete closure language of each of its closure termination states; it is a map of closure termination states of q to abstract output words, which summarizes the corresponding closure output languages: $\mathsf{abstclosure} : (Q \times \mathfrak{W}) \to 2^{Q \to \mathfrak{W}}$. A call $\mathsf{abstclosure}(q, \mathfrak{w}_0)$, with an initial abstract output word $\mathfrak{w}_0$, returns a map $\{(q_t, \mathfrak{w}_t) \mid q_t \in \mathsf{closureterm}(q) \wedge [\![\mathfrak{w}_t]\!]_{\text{out}} \subseteq [\![\mathfrak{w}_0]\!]_{\text{out}} \circ \Omega(q, q_t)\}$.

We extend the abstract output closure operator $\mathsf{abstclosure}$ to sets:

Definition 25: Abstract Output Closure

The abstract output closure of a given set of transducer states $\widehat{\mathsf{abstclosure}} : 2^{Q \times \mathfrak{W}} \to 2^{Q \to \mathfrak{W}}$ is defined as $\widehat{\mathsf{abstclosure}}(S) = \bigsqcup_{\to} \bigcup \{(q_\Omega, \mathfrak{w}_\Omega) \mid (q, \mathfrak{w}_0) \in S \wedge (q_\Omega, \mathfrak{w}_\Omega) \in \mathsf{abstclosure}(q, \mathfrak{w}_0)\}$.

Actual implementations of an abstract output closure operator can be provided, for example, based on abstract interpretation, or based on techniques from automata theory. Even transducers can be used [218] to compute abstractions of languages, in our case, the concrete output languages that are produced in the ϵ-closure. We give two examples of implementations:

Joining Closure. The first abstract output closure operator $\mathsf{abstclosure}_\sqcup$ joins all abstract output words that can be found on transitions in the epsilon closure from control state q that are mapped to the same closure termination state. Let us assume that there is an operation $\mathsf{closuretrans} : Q \times Q \to 2^\Delta$ that, given a pair of control states $q, q_\Omega \in Q$, returns all transitions from the transition relation δ that are in the epsilon closure $\mathsf{epsclosure}(q)$ and are mapped to a closure termination state q_Ω. Then, we can define the closure operator as follows: $\mathsf{abstclosure}_\sqcup(q, \mathfrak{w}_0) = \{(q_\Omega, \mathfrak{w}_0 \sqcup_{\text{out}} \bigsqcup_{\text{out}}\{\mathfrak{w} \mid (\cdot, \cdot, \cdot, \mathfrak{w}) \in \mathsf{closuretrans}(q, q_\Omega)\}) \mid q_\Omega \in \mathsf{closureterm}(q)\}$. This operator produces an overapproximation of the concrete output language. The resulting abstraction does neither preserve information on the flow nor is path information kept.

Regular Closure. Another example of an output closure operator is $\mathsf{abstclosure}_\infty$. Here, we assume that the abstract output words can be described based on an abstract domain of ∞-regular languages [185], with a corresponding lattice thereof. Rules for transforming automata into regular expressions can be applied [185]: The result for the transducer in Fig. 8 is $\mathsf{abstclosure}_\infty(q_0, \mathfrak{w}_0) = \{(q_2, \mathfrak{w}_0 \circ \mathfrak{w}_b), (q_\perp, \mathfrak{w}_0 \circ \mathfrak{w}_a \circ (\mathfrak{w}_c \circ \mathfrak{w}_f \circ \mathfrak{w}_d)^\omega)\}$. This type of output closure is lossless. Nevertheless, not all applications require this level of detail.

3.2.4 Runs

We now define runs of abstract transducers and illustrate how they are conducted for given inputs. All runs of an abstract transducer start from the initial transducer state and follow the transition relation:

Definition 26: Abstract Transducer Run

A *run* of an abstract transducer on a concrete input word $\bar{\sigma} = \langle \sigma_1, \dots, \sigma_n \rangle \in \Sigma^*$ and a lookahead $\widehat{\sigma} \subseteq \Sigma^*$ is a sequence of transducer state transitions $\iota_0 \xrightarrow{\mathfrak{v}_1/\mathfrak{w}_1} \dots \xrightarrow{\mathfrak{v}_n/\mathfrak{w}_n} \iota_n$, also denoted by $\langle \iota_0, \dots, \iota_n \rangle$ in case the actual transducer transitions are irrelevant for the discussion. A run always starts in the initial transducer state $\iota_0 \in J$, is well-founded in the transition relation δ, and all transitions along the input match, that is, the quotienting $(\langle\!\langle \{\langle \sigma_i, \dots, \sigma_n \rangle\} \circ \widehat{\sigma} \rangle\!\rangle_{\text{in}})^{\mathfrak{v}_i} \neq \bot$ does not result in the abstract bottom word.

Before we continue to define feasible and accepting runs of an abstract transducer, we define the abstract output of a run:

Definition 27: Abstract Run Output

The *abstract output* of a run $\iota_0 \xrightarrow{\mathfrak{v}_1/\mathfrak{w}_1} \dots \xrightarrow{\mathfrak{v}_n/\mathfrak{w}_n} \iota_n$ is the concatenation of the subsequent abstract output words $\mathfrak{w}_\circ = \mathfrak{w}_0 \circ \mathfrak{w}_1 \circ \dots \circ \mathfrak{w}_n$. The abstract output word $\mathfrak{w}_0$ is one abstract output word from the initial transducer state, that is, there exists a pair $(\cdot, \mathfrak{w}_0) \in \iota_0$.

The output of an abstract transducer run is essential for the definition of feasible transducer runs:

Definition 28: Feasible Run

A run is called *feasible* if and only if its abstract output $\mathfrak{w}_\circ$ is not the bottom element $\bot$, that is, if and only if $[\![\mathfrak{w}_\circ]\!]_{\text{out}} \neq \emptyset$. The set of all concrete inputs (with lookaheads) that result in a feasible run on an abstract transducer T defines the function $\mathsf{feasible}_T : \Sigma^* \times 2^{\Sigma^*} \to \mathbb{B}$.

Abstract transducers can also operate as acceptors and define a set of inputs to be accepted. We first define the notion of an accepting run and define the accepted input language later:

Definition 29: Accepting Run

A run $\langle \iota_0, \dots, \iota_n \rangle$ is called to be *accepting* if it is feasible and its last transducer state contains an accepting (final) control state, that is, if and only if $(q_n, \mathfrak{w}_n) \in \iota_n$, with $q_n \in F$ and $\mathfrak{w}_n \neq \bot$.

In general, an abstract transducer is a nondeterministic automaton, nevertheless it can be deterministic if it satisfies following criterion:

Definition 30: Deterministic Abstract Transducer

We call an abstract transducer *deterministic* if and only if it does not allow a run $\bar{\iota} = \langle \iota_0, \dots \iota_n \rangle$ with a transducer state $\iota_i \in \bar{\iota}$ that consists of more than one element, that is, $\forall \iota_i \in \bar{\iota} : |\iota_i| \leqslant 1$.

We now continue with an operational perspective on the runs of an abstract transducer. Given a *concrete input word* $\bar{\sigma} \in \Sigma^*$ based on the concrete input alphabet Σ and a set $\widehat{\sigma} \subseteq \Sigma^*$ of words that can follow to this word (used for the lookahead), which output does the transducer produce and does processing the word terminate in an accepting control state? Since a concrete input word can be represented as an abstract word, and we consider this the more general case, we describe runs based on abstract input words: A given concrete input word $\bar{\sigma} \in \Sigma^*$ can be transformed to an abstract input word by applying the abstraction operator such that we end up in an abstract word $\mathfrak{v} = \langle\!\langle \{\bar{\sigma}\} \rangle\!\rangle_{\text{in}}$, with $[\![\mathfrak{v}]\!]_{\text{in}} = \{\bar{\sigma}\}$.

Definition 31: Run

The function $\mathsf{run}_T : Q \times \mathfrak{W} \times \mathfrak{I} \times \mathfrak{I} \to 2^{Q \to \mathfrak{W}}$ conducts a run starting from a control state $q \in Q$, an initial abstract output word $\mathfrak{w} \in \mathfrak{W}$, an abstract input word $\mathfrak{v} \in \mathfrak{I}$, with $\mathfrak{v} \neq \bot$, and an abstract word $\mathfrak{v}_\ell \in \mathfrak{I}$ that describes the lookahead that must be satisfied:

$$\mathsf{run}_T(q, \mathfrak{w}, \mathfrak{v}, \mathfrak{v}_\ell) = \begin{cases} \{\, (q, \mathfrak{w}) \,\} & \text{if } \mathfrak{v} = \mathfrak{v}_\epsilon \\ \bigsqcup_{\rightarrow} \bigcup \{\, \mathsf{run}_T(q'', \mathfrak{w} \circ \mathfrak{w}'', \mathsf{tail}(\mathfrak{v}), \mathfrak{v}_\ell) \mid & \\ \quad (q, \mathfrak{v}_\tau, q', \mathfrak{w}') \in \delta & \\ \quad \wedge\ (q'', \mathfrak{w}'') \in \mathsf{abstclosure}(q', \mathfrak{w}') & \\ \quad \wedge\ (\mathfrak{v} \circ \mathfrak{v}_\ell)^{\mathfrak{v}_\tau} \neq \bot & \\ \quad \wedge\ (\mathsf{head}(\mathfrak{v}))^{\mathsf{head}(\mathfrak{v}_\tau)} \neq \bot \,\} & \text{otherwise} \end{cases}$$

The function run terminates its recursion if the abstract input word is the bottom element. The recursive call to run is done for the tail of the abstract input word—which ensures termination—in case a transition that leaves the given control state q matched the input.

We extend this function to $\widehat{\mathsf{run}}_T : 2^{Q \to \mathfrak{W}} \times \mathfrak{I} \times \mathfrak{I} \to 2^{Q \to \mathfrak{W}}$, which starts from a transducer state, and we define it as follows:

$$\widehat{\mathsf{run}}_T(\iota, \mathfrak{v}, \mathfrak{v}_\ell) = \bigsqcup_{\rightarrow} \bigcup_{(q, \mathfrak{w}) \in \iota} \mathsf{run}_T(q, \mathfrak{w}, \mathfrak{v}, \mathfrak{v}_\ell)$$

The transducer state to start from is omitted if it is the abstract transducer's initial transducer state ι_0, that is, $\widehat{\mathsf{run}}_T(\mathfrak{v}, \mathfrak{v}_\ell) = \widehat{\mathsf{run}}_T(\iota_0, \mathfrak{v}, \mathfrak{v}_\ell)$. Given a concrete input word $\bar{\sigma} \in \Sigma^*$ and a corresponding set of concrete words $\widehat{\sigma} \subseteq \Sigma^*$ for the lookahead, we write $\widehat{\mathsf{run}}_T(\bar{\sigma}, \widehat{\sigma})$ as an abbreviation for $\widehat{\mathsf{run}}_T(\iota_0, \bar{\sigma}, \widehat{\sigma})$, which is as an abbreviation for $\widehat{\mathsf{run}}_T(\iota_0, \langle\!\langle \{\bar{\sigma}\} \rangle\!\rangle_{\text{in}}, \langle\!\langle \widehat{\sigma} \rangle\!\rangle_{\text{in}})$.

96
Runs on Concrete Inputs

3.2.5 Languages and Transductions

Contrary to other types of finite state transducers [88] our abstract transducers distinguish between two types of input languages: the intermediate input language and the accepted input language.

Definition 32: Intermediate Input Language

The *intermediate input language* $\mathcal{L}_{in}(\mathsf{T}) \subseteq \Sigma^* \times 2^{\Sigma^*}$ of an abstract transducer T is the set of concrete input words for that the transducer can conduct feasible runs starting from the initial transducer state ι_0:

$$\mathcal{L}_{in}(\mathsf{T}) = \{\, (\bar{\sigma}, \widehat{\sigma}) \mid \mathsf{feasible}_{\mathsf{T}}(\bar{\sigma}, \widehat{\sigma}) \wedge \bar{\sigma} \in \Sigma^* \wedge \widehat{\sigma} \subseteq \Sigma^* \,\}.$$

It follows that each prefix $\bar{\sigma}_p \preceq \bar{\sigma}$ of each word $\bar{\sigma} \in \mathcal{L}_{in}(\mathsf{T})$ is also element of the intermediate input language, that is, $\bar{\sigma}_p \in \mathcal{L}_{in}(\mathsf{T})$.

The accepted input language reflects the established notion of input language, which is based on the set of words that can reach a final control state:

Definition 33: Accepted Input Language

The *accepted input language* $\mathcal{L}_{acc} \subseteq \mathcal{L}_{in}$ is the subset of the intermediate input language for which an accepting control state $q \in F$ is reached:

$$\mathcal{L}_{acc}(\mathsf{T}) = \{\, (\bar{\sigma}, \widehat{\sigma}) \in \mathcal{L}_{in}(\mathsf{T}) \mid (q, \cdot) \in \widehat{\mathsf{run}}_{\mathsf{T}}(\bar{\sigma}, \widehat{\sigma}) \wedge q \in F \,\}.$$

Beside the accepted input language, another characteristic of an abstract transducer is its set of transductions and its set of accepting transductions:

Definition 34: Transductions

The set of *transductions* $\mathcal{T}(\mathsf{T}) \subseteq \Sigma^* \times 2^{\Sigma^*} \times 2^{\Theta^\infty}$ of an abstract transducer T characterizes both its concrete input language and the outputs that are produced for them. One element $(\bar{\sigma}, \bar{\Sigma}_\ell, \bar{\Theta}) \in \mathcal{T}(\mathsf{T})$ from this set is a tuple that consists of a word prefix $\bar{\sigma}$ that is consumed by a run of the transducer, a set of concrete words $\bar{\Theta} \subseteq \Sigma^*$ to conduct the lookahead on and that remains to be consumed by the next transitions of the transducer, and the set of concrete output words $\bar{\Theta} \subseteq \Theta^\infty$ that are emitted with the consumption of word $\bar{\sigma}$—see the definition of $\widehat{\mathsf{run}}_{\mathsf{T}}$ for more details:

$$\begin{aligned}\mathcal{T}(\mathsf{T}) = \bigcup \{\, & (\bar{\sigma}, \widehat{\sigma}, [\![\mathfrak{w}]\!]_{\mathrm{out}}) \mid (\bar{\sigma}, \widehat{\sigma}) \in \mathcal{L}_{in}(\mathsf{T}) \\ & \wedge (q, \mathfrak{w}) \in \widehat{\mathsf{run}}_{\mathsf{T}}(\bar{\sigma}, \widehat{\sigma}) \,\}.\end{aligned}$$

Definition 35: Accepting Transductions

The set of *accepting transductions* $\mathcal{T}_{\text{acc}}(\mathsf{T}) \subseteq \mathcal{T}(\mathsf{T})$ is the subset of the transductions of a given abstract transducer T that are produced by accepting runs:

$$\begin{aligned}\mathcal{T}(\mathsf{T}) = \bigcup \{\, (\bar{\sigma}, \widehat{\sigma}, [\![\mathfrak{w}]\!]_{\text{out}}) \mid (\bar{\sigma}, \widehat{\sigma}) \in \mathcal{L}_{in}(\mathsf{T}) \\ \wedge (q, \mathfrak{w}) \in \widehat{\text{run}}_{\mathsf{T}}(\bar{\sigma}, \widehat{\sigma}) \\ \wedge\, q \in F \,\}.\end{aligned}$$

The number of accepting transductions is greater or equal than the number of accepted input words, that is, $|\mathcal{L}_{acc}(\mathsf{T})| \leqslant |\mathcal{T}_{\text{acc}}(\mathsf{T})|$, because there can be independent concrete output languages for one concrete input $(\bar{\sigma}, \widehat{\sigma}) \in \Sigma^* \times 2^{\Sigma^*}$.

In combination, the set of transductions and the set of accepted transductions determine if two abstract transducers are equivalent to each other:

Definition 36: Equality

Two abstract transducers $\mathsf{T}_1, \mathsf{T}_2 \in \mathbb{T}$ are called *equivalent*, $\mathsf{T}_1 \equiv \mathsf{T}_2$, if and only if both have the same set of transductions and the same set of accepting transductions, that is, if and only if $\mathcal{T}(\mathsf{T}_1) = \mathcal{T}(\mathsf{T}_2)$ and $\mathcal{T}_{\text{acc}}(\mathsf{T}_1) = \mathcal{T}_{\text{acc}}(\mathsf{T}_2)$.

Based on the notion of equality, we can define different operations, for example, reduction or ϵ-elimination. We start by defining a more fundamental one: The union of two abstract transducers. The union is constructed similar to the union of ϵ-NFAs, with the exception that no ϵ-moves are added; we take advantage of the fact that the initial transducer state is a set:

Definition 37: Union

Given two abstract transducers $\mathsf{T}_1, \mathsf{T}_2 \in \mathbb{T}_{D_{\text{in}} \times D_{\text{out}}}$ that both have the same abstract input domain D_{in} and the same abstract output domain D_{out}, such that $\mathsf{T}_1 = (Q_1, D_{\text{in}}, D_{\text{out}}, \iota_{0\,1}, F_1, \delta_1)$ and $\mathsf{T}_2 = (Q_2, D_{\text{in}}, D_{\text{out}}, \iota_{0\,2}, F_2, \delta_2)$. The *union* $\cup : \mathbb{T} \times \mathbb{T} \to \mathbb{T}$ of two abstract transducers results in a new abstract transducer $\mathsf{T}_\cup = \mathsf{T}_1 \cup \mathsf{T}_2$ that maintains exactly both the union of the set of transductions and the set of accepting transductions, that is, $\mathcal{T}(\mathsf{T}_\cup) = \mathcal{T}(\mathsf{T}_1) \cup \mathcal{T}(\mathsf{T}_2)$ and $\mathcal{T}_{\text{acc}}(\mathsf{T}_\cup) = \mathcal{T}_{\text{acc}}(\mathsf{T}_1) \cup \mathcal{T}_{\text{acc}}(\mathsf{T}_2)$. We define the union as $\mathsf{T}_\cup = \cup(\mathsf{T}_1, \mathsf{T}_2) = (Q_1 \cup Q_2, D_{\text{in}}, D_{\text{out}}, \iota_{0\,1} \cup \iota_{0\,2}, F_1 \cup F_2, \delta_1 \cup \delta_2)$.

3.2.6 Elimination of ϵ-Moves

Since ϵ-moves are considered to be a convenience feature, eliminating them without losing any output must be possible—that is, without altering the semantics of the transducer. The ϵ-closure can allow sequences of state transitions of infinite length, that is, a means to encode this infinite information

Algorithm 2 $\mathsf{elim}(T_\epsilon)$

```
Input: Abstract transducer $T_\epsilon = (Q, D_{in}, D_{out}, \iota_0, F, \delta) \in \mathbb{T}$
Output: Abstract transducer $T \in \mathbb{T}$, with $T_\epsilon \equiv T$
     // Sentinel transitions for the initial transducer state, with $\mathfrak{v}_\epsilon \not\equiv \mathfrak{v}_{start}$
  1: $\delta_\epsilon = \delta \cup \{ (q_s, \mathfrak{v}_{start}, q, \mathfrak{w}) \mid (q, \mathfrak{w}) \in \iota_0 \}$
     // Shortcut ε-moves to their termination states
  2: $\delta' = \{ (q, \mathfrak{v}, q'', \mathfrak{w}'') \mid \tau = (q, \mathfrak{v}, q', \mathfrak{w}) \in \delta_\epsilon$
     $\wedge\, \mathfrak{v} \neq \mathfrak{v}_\epsilon \wedge (q'', \mathfrak{w}'') \in \mathsf{abstclosure}(\{(q', \mathfrak{w})\}) \}$
     // Reconstruct a new initial transducer state
  3: $\iota_0{}' = \{ (q, \mathfrak{w}) \mid (\cdot, \mathfrak{v}, q, \mathfrak{w}) \in \delta \wedge \mathfrak{v} = \mathfrak{v}_{start} \}$
     // Reassemble the components to a new abstract transducer
  4: return $(Q, D_{in}, D_{out}, \iota_0{}', F, \delta')$
```

into one (finite) output symbol is needed. An algorithm for computing abstract output closures provides such a means.

For the design of an ϵ-elimination algorithm, it is important to note that all states in the ϵ-closure of a control state become active when it is entered. This implies that then also the output that is produced along with these ϵ-moves must be emitted: Existing algorithms for ϵ-elimination are not applicable to abstract transducers. An algorithm for eliminating ϵ-moves from an abstract transducer T_ϵ must ensure that the resulting transducer T is equivalent $T_\epsilon \equiv T$. Please note that stuttering transitions must be made explicit and must be considered to allow a sound elimination of ϵ-moves.

Algorithm 2 is our approach for eliminating ϵ-moves from an abstract transducer. The algorithm constructs a new transition relation, from which all ϵ-moves are removed by adding shortcut transition to the closure termination states and concatenating the corresponding closure output language.

Proposition 1. Given an abstract transducer T_ϵ, all its ϵ-moves *can* be eliminated without affecting its semantics, that is, without affecting either the set of transductions or the set of accepting transductions. The abstract transducer T_ϵ can be transformed into an input-ϵ-free transducer T, with $T_\epsilon \equiv T$.

Proof. We prove the proposition by providing an algorithm that conducts this transformation while maintaining the set of transductions and the set of accepted transductions: Given an abstract transducer T_ϵ that has ϵ-moves, Algorithm 2—which we implicitly parameterize with the output closure operator $\mathsf{abstclosure}_\infty$—produces an abstract transducer T that is input-ϵ-free and satisfies $\mathcal{T}(T_\epsilon) = \mathcal{T}(T)$ and $\mathcal{T}_{acc}(T_\epsilon) = \mathcal{T}_{acc}(T)$. (1) The transition relation δ', and with it the resulting transducer T, is input-ϵ-free because only non-ϵ-moves are added to the transition relation. (2) The set of closure termination states, for which $\mathsf{abstclosure}_\infty$ provides a pairing with the corresponding output closure language, contains all accepting states (Definition 3.2.2), that is, all moves to accepting states are maintained, and with it the set of accepting transductions. (3) The set of transductions is maintained: The output from the epsilon closures, that is, the closure termination languages, are concatenated to the transitions to the closure termination states. □

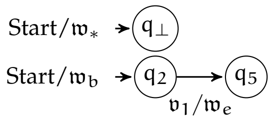


Figure 9: Without ϵ-moves

Example 5. Given the transducer in Fig. 8, Algorithm 2 proceeds as follows: First, we extend the transition relation with sentinels and get $\delta_\epsilon = \{(q_0, \mathfrak{v}_\epsilon, q_1, \mathfrak{w}_a), (q_1, \mathfrak{v}_\epsilon, q_3, \mathfrak{w}_c), (q_3, \mathfrak{v}_\epsilon, q_4, \mathfrak{w}_f), (q_4, \mathfrak{v}_\epsilon, q_1, \mathfrak{w}_d), (q_0, \mathfrak{v}_\epsilon, q_2, \mathfrak{w}_b), (q_2, \mathfrak{v}_\epsilon, q_5, \mathfrak{w}_e), (q_s, \mathfrak{v}_{start}, q_0, \mathfrak{w}_0)\}$. In the next step, ϵ-moves are left out by adding transitions to the closure termination states and concatenating the corresponding closure output languages; the result is a new transition relation $\delta' = \{(q_s, \mathfrak{v}_{start}, q_\perp, \mathfrak{w}_*), (q_s, \mathfrak{v}_{start}, q_2, \mathfrak{w}_b), (q_2, \mathfrak{v}_1, q_5, \mathfrak{w}_e)\}$, with $\mathfrak{w}_* = \mathfrak{w}_a \circ (\mathfrak{w}_c \circ \mathfrak{w}_f \circ \mathfrak{w}_d)^\omega$. Then, the initial transducer state is reconstructed from the relation δ' and we get $\iota_0{}' = \{(q_\perp, \mathfrak{w}_*), (q_2, \mathfrak{w}_b)\}$. Finally, the transducer is re-assembled and we get the transducer shown in Fig. 9.

3.2.7 Determinization

A typical operation when dealing with finite state machines is the transformation of a nondeterministic automaton into a deterministic one. This is *not possible* for abstract transducers in general: The control-flow structure of the state-transitions within the ϵ-closure describes different information flows—that is, sets of output words that reach different closure termination states—as its semantics, which is not the case for classical automata and transducers. For example, a state-space splitting might be intended based on the information of the emitted output—different outputs for the same input that lead to different control states. That is, different closure termination states, which can be accepting states, can have associated different closure termination languages; this separation must be maintained—which is also reflected in our definition of transducer equivalence.

Proposition 2. Not every nondeterministic abstract transducer T can be transformed into an equivalent deterministic transducer T_d, with $T \equiv T_d$.

Proof. We prove the proposition by counterexample—assuming that all abstract transducers can be determinized. Given an abstract transducer T with the set of initial transducer states $\iota_0 = \{(q_1, \mathfrak{w}_1), (q_2, \mathfrak{w}_2)\}$ and the relation $\delta = \{(q_1, \mathfrak{v}_1, q_3, \mathfrak{w}_3), (q_2, \mathfrak{v}_2, q_4, \mathfrak{w}_4)\}$, with $\langle a \rangle \in [\![\mathfrak{v}_1]\!]_{in}$ and $\langle b \rangle \in [\![\mathfrak{v}_2]\!]_{in}$, it has the set of transductions $\mathcal{T}(T) = \{(\epsilon, \{\epsilon\}, [\![\mathfrak{w}_1]\!]_{out}), (\epsilon, \{\epsilon\}, [\![\mathfrak{w}_2]\!]_{out}), (\langle a \rangle, \{\epsilon\}, [\![\mathfrak{w}_1 \circ \mathfrak{w}_3]\!]_{out}), (\langle b \rangle, \{\epsilon\}, [\![\mathfrak{w}_2 \circ \mathfrak{w}_4]\!]_{out})\}$. A determinized version would have an initial transducer state with only one element, that is, the initial transducer state can be either $\iota_{01} = \{(q_0, \mathfrak{w}_1 \sqcup \mathfrak{w}_2)\}$ of a transducer T_1 or $\iota_{02} = \{(q_0, \epsilon)\}$ of a transducer T_2. Both are wrong since transducer T intended an initial state space splitting with different output languages. Transducer T_2 does not have the transduction $(\epsilon, \{\epsilon\}, [\![\mathfrak{w}_1 \sqcup \mathfrak{w}_2]\!]_{out}) \in \mathcal{T}(T_2)$. The transductions of T_1 are not equal to those of T, since $\mathcal{T}(T_1) = \{(\epsilon, \{\epsilon\}, [\![\mathfrak{w}_1 \sqcup \mathfrak{w}_2]\!]_{out}), (\langle a \rangle, \{\epsilon\}, [\![(\mathfrak{w}_1 \sqcup \mathfrak{w}_2) \circ \mathfrak{w}_3]\!]_{out}), (\langle b \rangle, \{\epsilon\}, [\![(\mathfrak{w}_1 \sqcup \mathfrak{w}_2) \circ \mathfrak{w}_4]\!]_{out})\} \neq \mathcal{T}(T)$. □

Proposition 3. An abstract transducer needs a set of initial transducer states to allow for an elimination of ϵ-moves. That is, a set of initial transducer states with $|\iota_0| = 1$ is not sufficient for all ϵ-input-free transducers while maintaining their semantics.

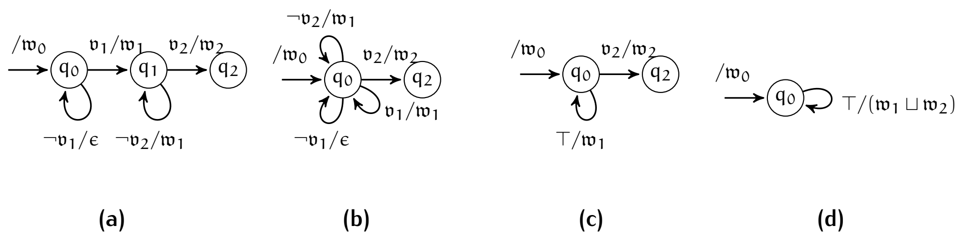

$/\mathfrak{w}_0$
$\mathfrak{v}_1/\mathfrak{w}_1$
$\mathfrak{v}_2/\mathfrak{w}_2$
q_0
q_1
q_2
$\neg\mathfrak{v}_1/\epsilon$
$\neg\mathfrak{v}_2/\mathfrak{w}_1$
(a)
$\neg\mathfrak{v}_2/\mathfrak{w}_1$
$/\mathfrak{w}_0$
$\mathfrak{v}_2/\mathfrak{w}_2$
q_0
q_2
$\mathfrak{v}_1/\mathfrak{w}_1$
$\neg\mathfrak{v}_1/\epsilon$
(b)
$/\mathfrak{w}_0$
$\mathfrak{v}_2/\mathfrak{w}_2$
q_0
q_2
$\top/\mathfrak{w}_1$
(c)
$/\mathfrak{w}_0$
q_0
$\top/(\mathfrak{w}_1 \sqcup \mathfrak{w}_2)$
(d)

Figure 10: Examples for different types of abstractions. Abstractions are applied step-wise from left to right: (a) we start with the unabstracted transducer, (b) we conduct a state abstraction by merging states q_0 and q_1, (c) we abstract the input alphabet, (d) we abstract the output alphabet.

Proof. Implication of the proof for proposition 2. □

3.3 TRANSDUCER ABSTRACTION

Abstracting (widening) an abstract transducer is a means to provide its output for a larger set of input words, that is, a *mechanism to increase sharing and with it the potential of reuse*. That is, we explicitly rely on the fact that abstracting an automaton can widen its input language, and introduces non-determinism [14]. We discuss different types of abstractions that are relevant for this work—Fig. 10 provides examples for abstractions. Approaches for abstracting classical automata and symbolic automata have been presented in the past [58, 217], which can also be adopted for abstract transducers.

Given an abstract transducer T, the abstraction operator $\langle\!\langle \cdot \rangle\!\rangle^{\pi} : \mathbb{T} \to \mathbb{T}$ with widening—with the abstraction precision π as an implicit parameter that determines the level of abstraction to achieve—has to guarantee that the resulting abstraction overapproximates both the set of transductions and the set of accepting transductions:

Definition 38: Overapproximation

An abstract transducer T_1 *overapproximates* another abstract transducer T_2, which we denote by $T_2 \models T_1$, if and only if T_1 overapproximates both the set of transductions and the set of accepting transductions of transducer T_2, that is, $T_2 \models T_1$ if and only if $\forall(\bar{\sigma}_2, \widehat{\sigma}_2, \widehat{\theta}_2) \in \mathcal{T}_{\text{acc}}(T_2) : \exists(\bar{\sigma}_1, \widehat{\sigma}_1, \widehat{\theta}_1) \in \mathcal{T}_{\text{acc}}(T_1) : \bar{\sigma}_1 = \bar{\sigma}_2 \wedge \widehat{\sigma}_1 = \widehat{\sigma}_2 \wedge \widehat{\theta}_2 \subseteq_C \widehat{\theta}_1$ and $\forall(\bar{\sigma}_2, \widehat{\sigma}_2, \widehat{\theta}_2) \in \mathcal{T}(T_2) : \exists(\bar{\sigma}_1, \widehat{\sigma}_1, \widehat{\theta}_1) \in \mathcal{T}(T_1) : \bar{\sigma}_1 = \bar{\sigma}_2 \wedge \widehat{\sigma}_1 = \widehat{\sigma}_2 \wedge \widehat{\theta}_2 \subseteq_C \widehat{\theta}_1$. The relation $\subseteq_C$ denotes the inclusion relation of the concrete language lattice of the output language domain.

The classical approach to abstract an automaton is *state abstraction*, that is, to merge several control states into one [212]. Please note that this approach

Algorithm 3 qmerge(T, Q_m)

```
Input: Abstract transducer $T = (Q, D_{in}, D_{out}, \iota_0, F, \delta)$,
       set $Q_m$ of states to merge
Output: Abstract transducer $T'$, with $T \models T'$
Variables: Control state $q_m$ that is not in the set $Q$ of transducer $T$
    // Define the abstraction $\alpha$
 1: $\alpha = \{ (q, q') \mid q \in Q \land q' = q_m \text{ if } q \in Q_m \text{ else } q' = q \}$
    // New set of control states
 2: $Q' = (Q \setminus Q_m) \cup \{q_m\}$
    // New set of accepting states
 3: $F' = \{ \alpha(q) \mid q \in F \}$
    // New initial transducer state
 4: $\iota_0{}' = \bigsqcup_{\rightarrow} \{ (\alpha(q), M) \mid (q, M) \in \iota_0 \}$
    // New transition relation
 5: $\delta' = \{ (\alpha(q), \mathfrak{v}, \alpha(q'), \mathfrak{w}) \mid (q, \mathfrak{v}, q', \mathfrak{w}) \in \delta \}$
    // Compose the resulting transducer
 6: return $(Q', D_{in}, D_{out}, \iota_0{}', F', \delta')$
```

can also be used for abstracting output closures, which is the case if control states within an ϵ-closure are merged:

Definition 39: Control State Merge

A *state merge* for a given abstract transducer T is conducted by merging a set of its control states $Q_m \subseteq Q$ into one new state q_m, and results in a new abstract transducer T_m. We denote this process by the operator qmerge : $\mathbb{T} \times 2^Q \rightarrow \mathbb{T}$, that is, $T_m = \text{qmerge}(T, Q_m)$. The actual definition of operator qmerge is given by Algorithm 3.

Proposition 4. Given an abstract transducer T, a transformation $T' = \text{qmerge}(T, Q_m)$ results in a new abstract transducer T', with $T \models T'$, that is, transducer T' overapproximates transducer T.

Proof. We have to show that (1) each input $(\bar{\sigma}, \widehat{\sigma}) \in \Sigma^* \times 2^{\Sigma^*}$ that leads to a feasible run $\iota = \widehat{\text{run}}_T(\bar{\sigma}, \widehat{\sigma})$ on T also leads to a feasible run $\iota' = \widehat{\text{run}}_{T'}(\bar{\sigma}, \widehat{\sigma})$ on transducer T', and for each element $(q, \mathfrak{w}) \in \iota$ there exists an element $(q', \mathfrak{w}') \in \iota'$, with $[\![\mathfrak{w}]\!] \subseteq [\![\mathfrak{w}']\!]$. Furthermore, we have to show that (2) each input that leads to an accepting run on transducer T also lead to an accepting run on transducer T'. Given a run $\bar{\iota} = \langle \iota_0, \dots, \iota_n \rangle$ that is feasible on transducer T for a given input $(\bar{\sigma}, \widehat{\sigma}) \in \Sigma^* \times 2^{\Sigma^*}$. The same input will also produce a feasible run $\bar{\iota}' = \langle \iota_0{}', \dots, \iota_n' \rangle$ on transducer T'. For each $\iota_i \in \bar{\iota}$ with $(q_1, \cdot) \in \iota_i$ or $(q_2, \cdot) \in \iota_i$, the corresponding transducer state $\iota_i' \in \bar{\iota}'$ will contain the merged control state q_m with a corresponding abstract output word, that is, $(q_m, \mathfrak{w}) \in \iota_i'$. The definition of qmerge ensures that all transitions from either control state q_1 or q_2 are also possible from control state q_m: All transitions in δ from or to a control state in Q_m are replaced by corresponding transitions from or to control state q_m. In case a control state to merge is included in the initial transducer state ι_0, it is replaced by control state q_m in the initial transducer state $\iota_0{}'$ of transducer T'. The non-deterministic nature of abstract transducers ensures that all transitions that match will also be taken: One control state can have a set of successor states for a given input. The transformation of the set of accepting control states F to the set F' ensures that if one of the states to merge was an accepting state, also state q_m will become an accepting state; states in F that are not included in Q_m stay accepting states in F'. That is, all transitions—and runs on them—that were possible from or to control states in the set Q_m are still possible (lead to feasible or accepting runs) in the new abstract transducer T', but now start or end in control state q_m. □

Please note that abstracting abstract transducers by merging control states does neither affect the number of transitions nor their labeling—both the input symbols and the output symbols on transitions stay the same, but output languages of epsilon closures can change.

Definition 40: State Abstraction

The state abstraction $\langle\!\langle T \rangle\!\rangle_Q^\pi$ of an abstract transducer T results in a new abstract transducer T' that is computed based on an abstraction precision π, with $T \models T'$. The abstraction precision determines which states to keep separated and which to combine into one state—which represents the corresponding equivalence class. The abstraction precision $\pi = \langle Q_1, \ldots, Q_n \rangle$ defines a list of disjoint sets of control states that should be combined. A state abstraction is conducted as follows:

$$\langle\!\langle T \rangle\!\rangle_Q^\pi = \begin{cases} \mathsf{qmerge}(T, Q_1) & \text{if } \pi = \langle Q_1 \rangle \\ \mathsf{qmerge}(\langle\!\langle T \rangle\!\rangle_Q^{\langle Q_2, \ldots, Q_n \rangle}, Q_1) & \text{if } |\pi| > 1 \text{ and } \pi = \langle Q_1, Q_2, \ldots \rangle \end{cases}$$

An abstraction approach that influences the abstract input words of the transitions is input alphabet abstraction, which is the process of changing the abstract input word $\mathfrak{v}$ of a transducer transition $\tau = (q, \mathfrak{v}, q', \mathfrak{w}) \in \delta$ to an new abstract input word $\mathfrak{v}'$, with $[\![\mathfrak{v}]\!]_{\text{in}} \subseteq [\![\mathfrak{v}']\!]_{\text{in}}$:

Definition 41: Input Alphabet Abstraction

An *input alphabet abstraction* $\langle\!\langle T \rangle\!\rangle_I^\pi$ of an abstract transducer T results in a new abstract transducer where some of the abstract input words of its control transitions were widened based on the given abstraction precision $\pi \in \Pi_I$. The abstraction precision π for input alphabet abstraction maps an abstraction precision π_{in} that is applicable to the abstract input domain to each of the transducer's control transitions, that is, it is a left-total function $\pi : \Delta \rightarrow \Pi_{\text{in}}$. The result is an abstract transducer with a widened transition relation:

$$\delta' = \{\, (q, \langle\!\langle \mathfrak{v} \rangle\!\rangle_{\text{in}}^{\pi_{\text{in}}}, q', \mathfrak{w}) \mid \tau = (q, \mathfrak{v}, q', \mathfrak{w}) \in \delta \wedge (\tau, \pi_{\text{in}}) \in \pi \,\}.$$

Along with this work, we introduce an output alphabet abstraction, which adjusts the abstract output words of transitions. It denotes the process of changing the abstract output word $\mathfrak{w}$ of a transducer transition $\tau = (q, \mathfrak{v}, q', \mathfrak{w}) \in \delta$ to an new abstract output word $\mathfrak{w}'$, with $[\![\mathfrak{w}]\!]_{\mathrm{out}} \subseteq [\![\mathfrak{w}']\!]_{\mathrm{out}}$:

Definition 42: Output Alphabet Abstraction

An *output alphabet abstraction* $\langle\!\langle T \rangle\!\rangle^{\pi}_{O}$ of an abstract transducer T results in a new transducer where some of the abstract output words of its control transitions were widened based on the given abstraction precision $\pi \in \Pi_O$. The precision π for output alphabet abstraction maps an abstraction precision π_{out} that is applicable to the output domain to each of the transducer's transitions, that is, it is a left-total function $\pi : \Delta \to \Pi_{\mathrm{out}}$. The result is an abstract transducer with a widened transition relation:

$$\delta' = \{\, (q, \mathfrak{v}, q', \langle\!\langle \mathfrak{w} \rangle\!\rangle^{\pi_{\mathrm{out}}}_{\mathrm{out}}) \mid \tau = (q, \mathfrak{v}, q', \mathfrak{w}) \in \delta \wedge (\tau, \pi_{\mathrm{out}}) \in \pi \,\}.$$

Please note that also the computation of the abstract output closure—see Sect. 3.2.3—yields a form of output alphabet abstraction.

3.4 TRANSDUCER REDUCTION

Besides abstraction techniques, also techniques for the reduction of abstract transducers are important. Such techniques help to reduce the number of control states, the number of control transitions, and the degree of non-determinism of a given abstract transducer. That is, they help to reduce the costs of using and running abstract transducers for particular inputs, for example, to conduct a verification task. Minimization is related to reduction but aims at ending up in finite state machines with a minimal number of states—an optimum.

The number of control states of an abstract transducer is critical for the performance of its use in an analysis procedure. Since a minimization is too expensive [47, 85, 154], we propose to adopt reduction techniques as known for NFAs to reduce the size and the degree of non-determinism of abstract transducers—a low degree of non-determinism is critical for efficient execution of non-deterministic finite state machines [164].

Abstract transducers can be reduced by merging control states, or their transitions, as long as the set of transductions and the set of accepting transductions is preserved. Please note that we assume, if not stated otherwise, that ϵ-moves were removed *before* applying the reduction techniques that we describe here.

Definition 43: Operator reduce

The (generic) reduction operator $\mathtt{reduce} : \mathbb{T} \to \mathbb{T}$ reduces a given abstract transducer T. Instances of this operator have to guarantee to produce an equivalent abstract transducer, that is, $T \equiv \mathtt{reduce}(T)$.

Reduction by State Merging

Before we continue to outline an algorithm for reducing abstract transducers by merging control states, we provide more definitions:

Definition 44: Control State Equality

Two control states $q_1, q_2 \in Q$ of an abstract transducer T are called *equivalent* to each other, that is, $q_1 \equiv q_2$, if and only if they can be merged without affecting the transducer's set of transductions nor its set of accepting transductions, that is, if and only if $T \equiv \mathtt{qmerge}(T, \{q_1, q_2\})$.

Based on the definition of control state equality, we define the equality of abstract transducer states:

Definition 45: Transducer State Equality

Two transducer states $\iota_1, \iota_2 \in \mathcal{J}$ are called *equivalent* if and only if they describe equivalent pairs of control states and abstract output words, that is, if and only if $\forall (q, \mathfrak{w}) \in \iota_1 : \exists (q', \mathfrak{w}') \in \iota_2 : q \equiv q' \wedge \mathfrak{w} \equiv \mathfrak{w}'$ and $\forall (q, \mathfrak{w}) \in \iota_2 : \exists (q', \mathfrak{w}') \in \iota_1 : q \equiv q' \wedge \mathfrak{w} \equiv \mathfrak{w}'$.

To determine whether merging two control states maintains the set of transductions, the notion of left transductions is essential:

Definition 46: Left Transductions

The set of *left transductions* $\overleftarrow{\mathcal{T}}(T, q) \subseteq \Sigma^* \times 2^{\Sigma^*} \times 2^{\Theta^\infty}$ to a given control state $q \in Q$, which belongs to a particular abstract transducer $T \in \mathbb{T}$, is the set of all transductions that can be produced on paths that start in the initial transducer state ι_0 and that *reach* the given control state q with a feasible run:

$$\begin{aligned}\overleftarrow{\mathcal{T}}(T, q) = \bigcup \{\, & (\bar{\sigma}, \widehat{\sigma}, [\![\mathfrak{w}']\!]_{\mathrm{out}}) \\ & \mid (q, \mathfrak{w}') \in \widehat{\mathsf{run}}_T(\bar{\sigma}, \widehat{\sigma}) \\ & \wedge \bar{\sigma} \in \Sigma^* \wedge \widehat{\sigma} \subseteq \Sigma^* \wedge \mathfrak{w}' \neq \bot \,\}.\end{aligned}$$

Proposition 5. A transformation $T' = \mathtt{qmerge}(T, \{q_1, q_2\})$ maintains both the set of transductions and the set of accepting transductions if the left-transductions of the control states q_1 and q_2 are equal, that is, $T' \equiv \mathtt{qmerge}(T, \{q_1, q_2\})$ if $\overleftarrow{\mathcal{T}}(T, q_1) = \overleftarrow{\mathcal{T}}(T, q_2)$.

Proof. Control state q_1 is reachable by runs that correspond to the set of left transductions $\overleftarrow{\mathcal{T}}(T, q_1)$ and control state q_2 by runs that correspond to the set of left transductions $\overleftarrow{\mathcal{T}}(T, q_2)$. The proposition states that if we merge control states q_1 and q_2, with $\overleftarrow{\mathcal{T}}(T, q_1) = \overleftarrow{\mathcal{T}}(T, q_2)$ into a new state q_m of a new transducer $T' = \mathtt{qmerge}(T, \{q_1, q_2\})$ then this transducer is equivalent $T' \equiv T$ to the original one. (1) First, we show that control state q_m is reachable by all feasible runs that can also reach control state q_1 or q_2, and that there is no feasible run that can reach q_m but neither state q_1 or q_2. That is, we show that $\overleftarrow{\mathfrak{T}}(T', q_m) = \overleftarrow{\mathfrak{T}}(T, q_1) = \overleftarrow{\mathfrak{T}}(T, q_2)$: The operation qmerge ensures that all transitions that entered either state q_1 or state q_2 also enter state q_m; that is, all feasible runs that reached q_1 or q_2 now reach state q_m and since q_m is a new state it is only reachable by these runs. (2) Next, we show that all runs that are feasible from control state q_1 or q_2 are also feasible from control state q_m, and there is no feasible run from state q_m that is not feasible from control state q_1 or q_2: The construction process of q_m ensures that all transitions that leave states q_1 or q_2 also leave state q_m, and no other transitions get added to leave this state; that is, all feasible runs that start in control state q_m are also feasible runs if they start in control state q_1 or q_2. (3) Finally, we have to show that all runs that are accepting from control state q_1 or q_2 are also accepting from state q_m, and there is no accepting run from control state q_m that is not accepting from state q_1 or q_2: The operator qmerge merges states q_1 and q_2 into a state q_m, which becomes an accepting control state if also state q_1 or state q_2 is an accepting control state. That is, the inputs $\{(\bar{\sigma}, \widehat{\sigma}) \mid (\bar{\sigma}, \widehat{\sigma}, \cdot) \in \overleftarrow{\mathfrak{T}}(T, q_1)$ become elements of set of accepting transductions of transducer T' if they were also accepted by transducer T. All inputs that get accepted by runs starting from control state q_1 or state q_2, get also accepted by runs that start from control state q_m. □

Statements about the result of manipulating an abstract transducer by merging control states are also possible based on the notion of right transductions:

Definition 47: Right Transductions

The set of *right transductions* $\overrightarrow{\mathfrak{T}}(T, q, \mathfrak{w}_0) \subseteq \Sigma^* \times 2^{\Sigma^*} \times 2^{\Theta^\infty}$ of a given control state $q \in Q$, which belongs to a specific abstract transducer $T \in \mathbb{T}$, with initial abstract output word $\mathfrak{w}_0$, is the set of all transductions that can be produced on the *feasible runs* that start from the given transducer state $(q, \mathfrak{w}_0)$:

$$\begin{aligned}\overrightarrow{\mathfrak{T}}(T, q, \mathfrak{w}_0) = \bigcup \{\, & (\bar{\sigma}, \widehat{\sigma}, [\![\mathfrak{w}']\!]_{\text{out}}) \\ & \mid (\cdot, \mathfrak{w}') \in \widehat{\mathsf{run}}_T(\{(q, \mathfrak{w}_0)\}, \bar{\sigma}, \widehat{\sigma}) \\ & \wedge \bar{\sigma} \in \Sigma^* \wedge \widehat{\sigma} \subseteq \Sigma^* \wedge \mathfrak{w}' \neq \bot \,\}.\end{aligned}$$

Definition 48: Right Accepted Language

The *right accepted language* of a given abstract transducer T for a given control state q is the set of pairs $(\bar{\sigma}, \widehat{\sigma}) \in \Sigma^* \times 2^{\Sigma^*}$ that lead to an accepting run if started from the given control state q:

$$\begin{aligned}\overrightarrow{\mathcal{L}_{acc}}(T, q) = \{\, & (\bar{\sigma}, \widehat{\sigma}) \in \mathcal{L}_{in}(T) \\ & \mid (q', \cdot) \in \widehat{\mathsf{run}}_T(\{(q, \mathfrak{w}_\epsilon)\}, \bar{\sigma}, \widehat{\sigma}) \\ & \wedge q' \in F \,\}.\end{aligned}$$

Proposition 6. Merging two control states $q_1, q_2 \in Q$ of an abstract transducer T, which results in a new abstract transducer, maintains the set of transductions if their sets of *right transductions* are equal, that is, $\mathcal{T}(T) = \mathcal{T}(\mathsf{qmerge}(T,\{q_1,q_2\}))$ if $\overrightarrow{\mathcal{T}}(T,q_1) = \overrightarrow{\mathcal{T}}(T,q_2)$. Please note that we do not make a proposition about the set of *accepted* transductions here.

Proof. Let the set of left transductions of two control states q_1 and q_2 be different to each other, that is, $\overleftarrow{\mathcal{T}}(T,q_1) \neq \overleftarrow{\mathcal{T}}(T,q_2)$. From proposition 4 and the corresponding proof we known that a merge of control states q_1 and q_2 leads to an overapproximation, that is, $T \models \mathsf{qmerge}(T,\{q_1,q_2\})$. It remains to be shown that the set of transductions is preserved if the right-transductions of two control states to merge are actually equal: $\mathcal{T}(T) = \mathcal{T}(\mathsf{qmerge}(T,\{q_1,q_2\}))$ if $\overrightarrow{\mathcal{T}}(T,q_1) = \overrightarrow{\mathcal{T}}(T,q_2)$, that is, that the merge does not add additional transductions. To add additional transductions it would be necessary that the set of right-transductions of control state q_m overapproximates the union of the right-transductions of control states q_1 and q_2. Nevertheless, since $\overrightarrow{\mathcal{T}}(T,q_1)$ is equivalent to $\overrightarrow{\mathcal{T}}(T,q_2)$ also $\overrightarrow{\mathcal{T}}(T,q_m)$ does not add additional right transductions, that is, $\overrightarrow{\mathcal{T}}(T,q_1) = \overrightarrow{\mathcal{T}}(T,q_2) = \overrightarrow{\mathcal{T}}(T,q_m)$. □

Proposition 7. Merging two control states $q_1, q_2 \in Q$ of an abstract transducer T, which results in a new transducer, does *not* maintain the set of *accepting transductions* if their sets of left transductions are not equal to each other. That is, $\mathcal{T}_{\text{acc}}(T) \neq \mathcal{T}_{\text{acc}}(\mathsf{qmerge}(T,\{q_1,q_2\}))$ if $\overleftarrow{\mathcal{T}}(T,q_1) \neq \overleftarrow{\mathcal{T}}(T,q_2)$.

Proof. Let q_1 and q_2 be two control states of an abstract transducer T, with $q_1 \in F$ and $q_2 \notin F$. Merging these states by $\mathsf{qmerge}(T,\{q_1,q_2\})$ results in a new transducer T' with a control state q_m into that q_1 and q_2 have been merged, and that became an accepting control state $q_m \in F'$. In case the left transductions $\overleftarrow{\mathcal{T}}(T,q_1)$ and $\overleftarrow{\mathcal{T}}(T,q_2)$ are different to each other, different inputs can reach states q_1 and q_2. Both inputs that reached q_1 or q_2 can reach the control state q_m, and all these inputs now result in accepting runs since $q_m \in F'$, that is, also runs for inputs that reached q_2 and that were not accepting before now reach the accepting control state q_m, resulting in an overapproximation of the set of accepting transductions. □

Definition 49: Left Equivalent

The *left equivalence relation* $\equiv_L \subseteq Q \times Q$ describes the pairs of control states that are equivalent to each other and that have the same set of left-transductions—it is a subset of control state equivalence relation. That is, $(q_1,q_2) \in \equiv_L$ if $q_1 \equiv q_2$ and $\overleftarrow{\mathcal{T}}(T,q_1) = \overleftarrow{\mathcal{T}}(T,q_2)$.

Proposition 8. Given an input-ϵ-free abstract transducer T, a set of two control states $Q_m = \{q_1,q_2\} \subseteq Q$ of transducer T satisfy $\overleftarrow{\mathcal{T}}(q_1) = \overleftarrow{\mathcal{T}}(q_2)$ if $\forall (q,\mathfrak{v},q',\mathfrak{w}) \in \mathsf{entering}(Q_m) : \forall (q'',\mathfrak{v}',q',\mathfrak{w}') \in \mathsf{entering}(Q_m) : \mathfrak{v} \equiv \mathfrak{v}' \land \mathfrak{w} \equiv \mathfrak{w}' \land \overleftarrow{\mathcal{T}}(q) = \overleftarrow{\mathcal{T}}(q'')$. We use the auxiliary function $\mathsf{entering}(Q) = \{(q,\mathfrak{v},q',\mathfrak{w}) \in \delta \mid q' \in Q\}$.

Proof. Given the set of all control states $Q_p \subseteq Q$ from these control states in $Q_m = \{q_1, q_2\}$ are directly reachable, that is, $Q_p = \{q \mid (q, \cdot, q', \cdot) \in \delta \wedge q' \in Q_m\}$. If all states in the set Q_p have the same set of left-transductions, then only transitions from control states in the set Q_p to those control states in the set Q_m can affect whether or not the sets of left-transductions of states in Q_m are not equal to each other. □

Definition 50: Operator $\mathsf{reduce}_{\mathsf{Left}}$

The reduction operator $\mathsf{reduce}_{\mathsf{Left}} : \mathbb{T} \to \mathbb{T}$ reduces a given abstract transducer T by merging all control-states that are left-equivalent to another. The transformation satisfies $\mathsf{reduce}_{\mathsf{Left}}(T) \equiv T$.

Existing algorithms [85, 86] for reducing automata and symbolic transducers are not applicable because the set of transductions is not taken into account in the definition of equivalence.

3.5 ABSTRACT TRANSDUCER ANALYSIS

We now present a generic and configurable program analysis that *executes an abstract transducer*. This abstract transducer analysis keeps track of the *current transducer state* while processing the input. The analysis can be configured, for example, to determine the extent to which the transducer states should be tracked in a path sensitive manner—path sensitivity might be needed for particular analysis purposes only. Thus, we can mitigate the state-space explosion problem in some cases. The transducer analysis is the foundation for several analyses that we describe in this work, for example, for the Yarn transducer analysis, and the precision transducer analysis.

3.5.1 Abstract Transducer CPA

Our abstract transducer analysis is built on the concept of configurable program analysis (CPA) [31, 32]. The abstract transducer CPA

$$\mathbb{D}_T = (D_T, \rightsquigarrow_T, \downarrow_T, \mathsf{merge}_T, \mathsf{stop}_T, \mathsf{prec}_T, \mathsf{target}_T)$$

tracks a set of states of a given abstract transducer $T = (Q, D_{in}, D_{out}, \iota_0, F, \delta)$. The CPAs behavior is configured by using different variants of its operators. For example, varying the operator merge_T can configure the analysis to operate path sensitive, or only context sensitive and flow sensitive [31]. We rely on the strengthening operator $\downarrow_T$ for instantiating parameterized outputs. Other program analyses, which run in parallel to the abstract transducer analysis, can read and use the output words for different purposes. The abstract transducer analysis $\mathbb{D}_T$ is composed of the following components:

Abstract Domain D_T. The abstract domain $D_T = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ is defined based on a *map lattice* $\mathcal{E} = (J, \top, \bot, \sqsubseteq, \sqcup, \sqcap)$, with $J = 2^{Q \to \mathfrak{W}}$, where each element $\iota \in J$ of the lattice is an abstract transducer state—see Sect. 37 for more details on map lattices. One transducer state $\iota = \{(q, \mathfrak{w}), \ldots\} \in J$ is a mapping $\iota : Q \to \mathfrak{W}$ from control states to abstract output words. The analysis starts with the initial transducer state ι_0 of the abstract transducer to conduct runs for.

Transfer Relation $\rightsquigarrow_T$. The transfer relation $\rightsquigarrow_T \subseteq J \times G \times J \times \Pi$ defines abstract successor states of an abstract state $\iota = \{(q, \mathfrak{w}), \ldots\} \in J$ for a given control-flow transition $g \in G$ and abstraction precision $\pi \in \Pi$. We define this transition relation *without implicit stuttering*, that is, if there should be stuttering, the transducer must have corresponding transitions. The transfer relation is defined as follows:

$$\begin{aligned}\iota \overset{g}{\rightsquigarrow}_T \{\, \{(q, \mathfrak{w})\} \mid\ & (\mathfrak{v}, \mathfrak{v}_\ell) = \mathsf{look}(g, \ell) \\ & \wedge (q, \mathfrak{w}) \in \widehat{\mathsf{run}}_T(\iota, \mathfrak{v}, \mathfrak{v}_\ell) \\ & \wedge q \neq q_\perp \,\}.\end{aligned}$$

Please note that the function $\widehat{\mathsf{run}}$ is implicitly parameterized with an abstract closure operator abstclosure. The operator $\mathsf{look} : G \times \mathbb{N}_0 \to \mathfrak{I} \times \mathfrak{I}$ maps the given control-flow transition $g \in G$ to an abstract input word $\mathfrak{v}$ and provides a bounded lookahead of length ℓ in form of the abstract input word $\mathfrak{v}_\ell$ which is derived from the control-flow transitions that follow transition g on the control-transition relation of the underlying analysis task. 97

Alphabet Translation

The operator look does not only provide the lookahead but also translates between the alphabet of the graph that is traversed to the abstract input alphabet of the abstract transducer. That is, varying this operator provides different *views* on the given input, for example, a control-flow transition $g \in G$ can be translated to the function to that the transition belongs to, or to the successor control location that is reached by the control transition.

The operator merge can decide later if states should be tracked separately or not—see also Sect. 4.5 for a detailed discussion of the implications that result from merging state sets.

Operator $\downarrow_T$. The strengthening [31] operator $\downarrow_T : E_\times \times E_\times \times J \to J$ is called after all analyses that run in parallel have provided an abstract successor state as components for the composite state $e_\times = (e_1, \ldots, e_n) \in E_\times$. At this point, the strengthening operator can access the information that is present in any of the component states $e_i \in e_\times$ and use them to strengthen its own (component) state. We instantiate parameterized output words during strengthening. Information of an analysis that runs in parallel can be used to support various instantiation and synthesis mechanisms.

The strengthening $\iota' = \{(q', \mathfrak{w}')\} = \downarrow_T(e_\times, e'_\times, \iota)$ is conducted for a given transducer state $\iota = \{(q, \mathfrak{w})\}$, which is the result of conducting a transducer transition $\tau = (q, \mathfrak{v}, q', \mathfrak{w}) \in \delta$ for an input $(\bar{\sigma}, \widehat{\sigma}) \in \Sigma^* \times 2^{\Sigma^*}$. Beside the information that can be found in the composite states $e_\times$ and $e'_\times$, also the values that were bounded to the parameters of the abstract input word $\mathfrak{v}$ can be taken into account to instantiate the abstract output word $\mathfrak{w}'$. A consistent binding of parameters among different transitions, that is, for the whole program trace—as this is used by some aspects and corresponding weavers [5]—is not yet supported.

Operator merge_T. The merge operator $\mathsf{merge}_T : J \times J \times \Pi \to J$ controls if two transducer states should get combined, or if they should be explored separately and separate the state space. The behavior of the operator can be controlled based on a given precision $\pi \in \Pi$. The default is to always separate two different abstract states, that is, $\mathsf{merge}_T = \mathsf{merge}^{\mathsf{sep}}$ [31], which ensures the path sensitivity of the analysis. Please note that the abstract transducer analysis is typically one of several analyses that run as components of a composite analysis: Even if the analysis would conduct a merge, other component analyses might signal not to do so.

OPERATOR stop_T. The coverage check operator $\mathsf{stop}_T : J \times 2^J \to \mathbb{B}$ decides whether a given abstract state is already covered by a state reached or not. As default, we use the inclusion relation of the lattice, that is, $\mathsf{stop}_T = \mathsf{stop}^{\mathsf{sep}}$.

OPERATOR prec_T. The precision adjustment operator prec_T could conduct further abstraction of a given abstract state. We do not abstract here: A call $\mathsf{prec}_T(\iota, \pi, \cdot)$ returns the pair $(\iota, \pi) \in J \times \Pi$ without adjustments.

OPERATOR target_T. The *target operator* $\mathsf{target}_T : J \to 2^S$ determines the set of properties for which a given abstract state is a target state. Each *property is a task concern*, that is, the set of properties $S \subset H$ is a subset of the set H of task concerns. We assume that there is only one transition $\tau = (\cdot, \cdot, q', \cdot) \in \delta$ for each accepting control state $q' \in F$. We rely on a function $\zeta : \Delta \to 2^H$ that maps each transducer transition to a set of task concerns. Given an abstract transducer state $\iota = \{(q_1, \cdot), \dots (q_n, \cdot)\} \in J$, the operator returns:

98 *Concern Map*

$$\begin{aligned} \mathsf{target}_T(\iota) = \bigcup \{\ \zeta(t) \mid\ & (q, \cdot) \in \iota \\ & \wedge\ q \in F \\ & \wedge\, t = (\cdot, \cdot, q, \cdot) \in \delta\ \} \end{aligned}$$

3.5.2 Analysis Configurations

By relying on the CPA framework [31], the abstract transducer analysis is equipped with an inherent notion of configurability, and can be instantiated several times and in different ways within the framework, to conduct an analysis task in the most efficient and effective manner.

Transducer Composition. It might be necessary to execute several abstract transducers in parallel along with the state space exploration for an analysis task. Given a list $\langle T_1, \dots, T_n \rangle$ of abstract transducers to run, a list $\langle \mathbb{D}_1, \dots, \mathbb{D}_m \rangle$ of analyses, with $n \geqslant m > 1$, has to be instantiated. We assume that these transducers have the same abstract input domain and the same abstract output domain, and consider the composition of transducers with different abstract output domains to be future work. The first approach (*separation*) is to instantiate one analysis for each abstract transducer ($m = n$), which fosters a clear separation of concerns. Each of the m instantiated analyses adds one component to the composite (product) state that is formed by the composite analysis; the number of CPAs operators that are invoked transitively by the CPA algorithm increases. An alternative approach (*union*) is to construct the union $T_\cup = T_1 \cup \dots \cup T_n$ of the transducers to run and to run this single transducer $T_\cup$ with one abstract transducer analysis. Also *hybrid* approaches can be taken, that is, construct unions for subsets of the transducers, and run others separately.

State-Set Composition. One abstract state $\iota = \{(q_1, \cdot), (q_2, \cdot), \ldots\} \in J$ of the transducer analysis can contain several control states from the set Q of the abstract transducer to run for a given analysis task. The number of control states per abstract state can be controlled by the transducer analysis and its operators, for example, the operator merge, which decides whether or not to explore two abstract states separately. The decision to join two different control states into one state set of one abstract state of the transducer analysis can affect the path sensitivity of the analysis, that is, if it is possible to determine the branch of the state space that has led to a given control state.

3.6 RELATED WORK

Abstract transducers combine different concepts and techniques from formal methods, automata theory, domain theory, and abstract interpretation, to end up in a generic type of abstract machine. We discuss the related work based on the different concepts that can be found in abstract transducers and explain the relationship and differences to existing work.

Symbolic Alphabet. An abstract transducer can use arbitrarily composed abstract domains to define both its input and the output; for the input domain, we require that its lattice is dual to a Boolean algebra. We introduce a special class of abstract domain, the abstract word domain, to describe words of complex entities, such as program traces of concurrent systems. Symbolic finite automata and transducers [87] share the idea of using theories to describe sets of input and output symbols. Other types of automata describe their input symbols based on predicates [206] or as multi-valued input symbols [112, 169]. From the perspective of abstract transducers, trace partitioning domains [223], lattice automata [112], and regular expressions over lattice-based alphabets [195] are instances of abstract word domains.

Output Closure. With abstract transducers, we also introduce means to deal with ϵ-loops that are annotated with outputs, that is, to compute and use finite symbolic representations of outputs that potentially consist of exponentially many and infinitely long words. Compared to existing work [87, 207], we also consider ϵ-moves that lead to dead ends as relevant, handle them in our algorithms, and do not consider them as candidates for removal.

Lookahead. In each step of processing input, abstract transducers can conduct a lookahead on the remaining input to determine which transitions to take. Several other types of abstract machines provide the capability of lookaheads, for example, tree transducers, which had been extended to support regular lookaheads [100], or extended symbolic finite state machines [88]. A labeling with words instead of letters is also conducted in the case of generalized finite automata [237], but they consume full words in a transition step—instead of just one letter as is the case for abstract transducers.

Transducer Abstraction. By defining both the input alphabet and the output alphabet of abstract transducers based on an abstract domain, we can make use of the full range of abstraction mechanisms that were developed in the context of abstract interpretation for abstracting abstract transducers, that is, to widen their set of transductions. Approaches for abstracting classical automata have been presented in the past [58]. A more recent work [217] presented techniques for abstracting symbolic automata, which could also be adopted to abstract transducers. This work is the first that proposes to abstract a type of transducer for increasing its sharing, that is, widen the set of words for which particular outputs are produced.

Running Transducers. Running automata in parallel to a program analysis is an established concept in the fields of program analysis and verification [33, 38, 41]. Algorithmic aspects of how automata are executed, for example, how the current state of automata is represented in the state space of the analysis task, are in many cases [134] not discussed further, while the performance implications can be dramatical. Work in the context of configurable program analysis [29, 30, 40] is most transparent about this.

Transducers for Analysis and Verification. Transducers are widely used in the context of program analysis and verification. They are used, for example, for synthesis [216], to describe the input-output-relation of programs [49, 126, 216, 256], and for string manipulations [245, 256]. Automata that produce an output—which is then used in the analysis process—have been proposed in the form of assumption automata [30] for conditional model checking, error witness automata that output strengthening conditions [29] to narrow down the state space of the analysis process, and for correctness witnesses [40].

3.7 SUMMARY

This chapter has introduced abstract transducers, a type of abstract machines that map between an input language and an output language while taking a lookahead into account. In contrast to established finite-state transducers, abstract transducers have a strong focus on the intermediate language that they produce, which has several implications on the design of algorithms that operate on these machines. Both the input alphabet and the output alphabet of abstract transducers consist of *abstract words*, where one abstract word denotes a set of concrete words. Means for representing, constructing, and widening of abstract words, and for describing their relationship, are provided by the corresponding abstract word domain. Building on these abstract alphabets allows for *abstracting* these transducers.

99 *Abstract Machines*

100 *Abstract Words*

101 *Transducer Abstraction*

We use techniques from abstract interpretation and domain theory as the foundation for our abstraction mechanisms. The concept of abstract transducers enables several new applications: We discussed applications in the context of sharing task artifacts for reuse within program analysis tasks.

From the concept of abstract transducers, we instantiate the concept of *task artifact transducers*, which generalize a group of automata and transducers that are used in the context of program analysis and verification for reproducing and sharing information. These transducers provide information that contributes to an analysis task and its solution. The underlying graph structure of automata-based transducers allows us to capture the structure of information, share it, and enable its reuse. Task artifact transducers have sev-

102 *Task Artifact Transducers*

103 eral applications, and we present some of them along with this work: Yarn transducers, which provide sequences of program operations to weave into a transition system, and precision transducers, which are a means to define the level of abstraction for different parts of the state space. Other applications of task artifact transducers can be found in the context of providing and checking verification evidence, for example, transducers for error witnesses, which provide information that guides towards specification violations, or transducers for correctness witnesses, which provides certificates to check while traversing the control flow of programs.

Applications

“ *Imagination is a very high sort of seeing, which does not come by study, but by the intellect being where and what it sees, by sharing the path, or circuits of things through forms, and so making them translucid to others.* ”

Ralph Waldo Emerson

4 YARN TRANSDUCERS AND THE LOOM

Keywords: On-The-Fly Weaving, Specification Weaving, Automata Encoding, Symbolic Path Sensitivity, Hybrid Path Sensitivity

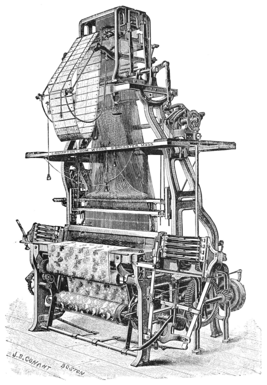


Figure 11: Carpet loom with Jacquard attachment. Source: Popular Science, Volume 39, 1891

This chapter introduces techniques for interposing (*weaving*) control-flow transitions, which are mapped to specific concerns, to the control transition relation of the analysis task, *on-the-fly, during the analysis process*. We call the control-flow transitions of a concern to introduce the Yarn. The analysis that interposes the Yarn into the transition relation is called the Loom. Any component of an analysis procedure can provide Yarn to weave, at different points in time, during the analysis process, for example, as a means to *delegate information* to encode about the state space to later and possibly different analysis steps. We introduce Yarn *transducers*, that is, automata that emit Yarn to weave as output; we use them to provide *formal specifications* and *models of the environment* to compose.

104 *History: Programs controlled Looms*

The idea of guiding a weaving process by programs dates back before the very beginning of universal computers and Turing complete programming languages and computer programs written in those. Figure 11 shows a carpet loom with a Jacquard attachment. The Jacquard attachment controls the loom based on a given "program": a sequence of punched cards that are joined together—forming an infinite loop—that defines the sequence of weaving operations to conduct. The Jacquard attachment and its punched cards also inspired Charles Babbage's work: the programmable Analytic Engine. First programs—sequences of operations—for this machine were written by Ada Byron (Countess Lady Lovelace)—she identified the importance of loops for computer programs.

105 *Composition of Analysis Tasks*

In static program analysis, an analysis task is composed of several parts (sometimes called modules, components, aspects, or processes) that represent different concerns of this task. For example, a verification task—for verifying the adherence to a particular specification—consists of the program, the specification, and the environment model. All these parts are themselves composed from smaller parts: For example, a specification is com-

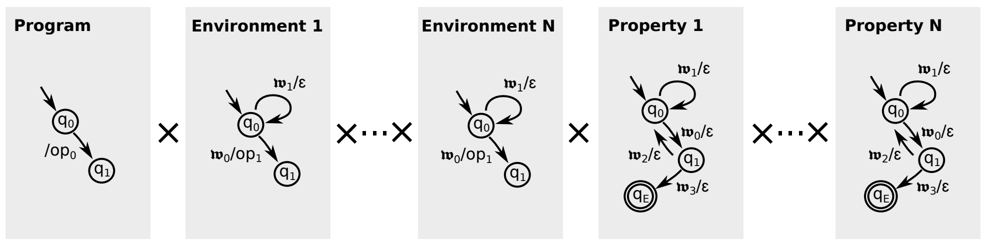

Program
q_0
/op$_0$
q_1
×
Environment 1
$\mathfrak{w}_1/\varepsilon$
q_0
$\mathfrak{w}_0$/op$_1$
q_1
×⋯×
Environment N
$\mathfrak{w}_1/\varepsilon$
q_0
$\mathfrak{w}_0$/op$_1$
q_1
×
Property 1
$\mathfrak{w}_1/\varepsilon$
q_0
$\mathfrak{w}_0/\varepsilon$
$\mathfrak{w}_2/\varepsilon$
q_1
q_E
$\mathfrak{w}_3/\varepsilon$
×⋯×
Property N
$\mathfrak{w}_1/\varepsilon$
q_0
$\mathfrak{w}_0/\varepsilon$
$\mathfrak{w}_2/\varepsilon$
q_1
q_E
$\mathfrak{w}_3/\varepsilon$

Figure 12: We can model the control-flow of a program, the environment model, and the specification—a set of properties—as YARN transducers. The output of these transducers (YARN) is composed (×) by weaving to end up in the final program analysis task.

posed of several properties—see Fig. 12 for an illustration of this scenario. An approach for program analysis can be *aware of this compositional structure*, and take advantage of the *additional knowledge* to make the analysis more efficient and effective, or the composition structure might not have been made explicit to the approach and it would have to rediscover it on its own, based on a costly dependency analysis [144, 158, 180].

106 *Offline Composition of Analysis Tasks*

Given a program $\overline{P}$ to verify, the established approach [18] to construct a verification task is to generate a new program $\overline{P}'$ that is the result of instrumenting the specification and the environment model to the program $\overline{P}$. The program $\overline{P}'$ is then given to the verifier, where checking the (safety) specification is reduced to checking the *reachability* of a specific control location: the error location l_{Err} [64, 136, 150]. That is, statements from the original program, the environment model, or the specification *cannot be distinguished* from each other by a verification tool without any additional hints regarding the composition structure: *traceability is lost*. Having a given encoding of the specification, or the environment model, at hand reduces the flexibility of choosing (or *transforming* to) a different encoding *on-demand*.

107 *Analysis Task Concerns as YARN Transducers*

Representing parts of a reasoning task as separate transducers (1) fosters a clear separation of concerns, it (2) brings the flexibility of choosing different encodings (for different parts) dynamically as needed (even different encodings for different parts of the state space), and (3) it provides the foundation for other techniques, for example, for the synthesis of abstraction precisions (invariants), or for slicing techniques.

108 *Hybrid Control Encoding*

YARN transducers provide sequences of control-flow transitions, which are labeled with program operations, to weave as output. Nevertheless, YARN transducers are still automata with a *current control state*, of which an analysis procedure has to *keep track*. Tracking the control state of a finite-state machine explicitly, that is, its concrete value—can cause an explosion of the abstract state space. To cope with this problem, the analysis that keeps track of the current control state of an automaton can emit YARN that allows for encoding the control-state of an automaton symbolically; services of the LOOM analysis are used to delegate the encoding of particular state-space facts to later analysis steps or analysis components. An *explicit*, *symbolic*, or *hybrid* (mixed) control encoding—and with a path sensitivity of this type—can be chosen, on demand, during the analysis process to keep track of the sequence of control states an automaton entered while proceeding along a program path.

Contributions. This chapter shares ideas and material (see page v) with our papers on "On-the-fly Decomposition of Specifications in Software Model Checking" [8] and presents the following contributions:

- We introduce YARN as the abstract entity to represent a set of sequences of program operations that contribute to a particular *concern* of an analysis task. We define *different operations* on YARNS, for example, their concatenation, and provide a detailed discussion of the role of assigning an appropriate *labeling* of control locations to achieve an efficient and effective analysis process.

- We present YARN *transducers*, a conceptually elegant approach, based on abstract transducers, for providing YARN to weave, for example, the full program itself, a safety specification, or an environment model. YARN transducers are a syntactic task artifact sharing model.

- We present the LOOM, a program analysis for composing YARN from different sources by weaving, for example, from YARN transducers. The LOOM provides the possibility to *delegate the encoding* of information to other analysis components or analysis steps (for more efficient or effective handling), aids in providing *traceability* of program fragments, and enables the idea of *on-the-fly weaving*.

- We introduce the concept of *dynamic control encoding*. An analysis that is responsible for running a transducer along a state-space exploration can delegate the encoding of the transducer's current control state, *on-demand*, to other components of the analysis tool by emitting *guards and assignments on state variables* as YARN to weave; this YARN can then be encoded—possibly more efficient—by other analyses, with different, possibly symbolic, abstract domains. This approach provides the flexibility to use the full range (hybrid) of encodings of control states dynamically: from pure explicit state representation to a fully weaved, and possibly symbolic, representation.

- We instantiate our concepts based on the CPA framework: The LOOM CPA implements the functionality of the LOOM and consumes YARN that is emitted by other analyses (CPAs). The YARN transducer CPA is responsible for executing a YARN transducer and for keeping track of its current control state; it uses *sets* of automaton control states and can emit conditions and assignments on *state variables* as YARN to allow for *different encodings* of the current control state *on demand*. We provide an open-source implementation of our approach.

- We conduct an *empirical study* on different schemes for composing and encoding YARN transducers that represent the formal specification and the environment model. We show the applicability of our techniques for multi-property verification—of which model-driven test generation is an instance.

Outline. We start by discussing the motivation for our work (Sect. 4.1): What are the origins of this work? Which problems did we face? Why are these problems important? What are the possible solutions? We will then (Sect. 4.2) introduce the notion of YARN: How is it defined? What were the requirements—especially from the perspective of software model checking—that led to our formalization? Which operations are possible on YARN? The section that follows (Sect. 4.3) introduces the LOOM, our mechanism for composing transition relations from YARN: How is it defined? At which points in time of an analysis process is it applicable? What does it guarantee to analyses that provide YARN to weave? How does its weaving procedure work? Section 4.4 introduces YARN transducers, an elegant technique to provide YARN to weave for composing analysis tasks: How are YARN transducers formally defined? What is their epsilon closure? How can YARN transducers be composed? How can YARN transducers be classified? How do YARN transducers become integrated into the analysis process? Section 4.5 introduces the notion of dynamic control encoding: How is it defined? What can we gain from it? What is the role of the LOOM analysis to implement this concept? Before we summarize the chapter (Sect. 4.8), we provide an empirical study (Sect. 4.6): Are they applicable in practice? Can these techniques make a verification process more efficient or effective?

4.1 MOTIVATION

109 *Multi-Property Verification*

This work has its roots in the context of multi-property verification [8]. Our goal is to provide the formal specification—and the environment model—as a set of (abstract) transducers, where each of them represents a different safety property—or part of the environment model. This separation of a specification into several properties—and their representation as separate transducers—aids in easily decomposing a verification task as needed.

110 *Focus on Task Composition and Encoding*

Already the process of composing an analysis task from several transducers poses many fundamental challenges, on which we focus in this chapter. Details of our decomposition framework can be found in the paper [8] and are not discussed further in this work.

111 *Challenges*

Despite the beauty of our approach, several challenges have to be addressed to make it both efficient and effective in practice. In the following, we discuss the requirements, challenges, and possible solutions.

4.1.1 Expressiveness Needed

An approach for specifying code fragments of program concerns should be as simple as possible—for the user, and for people that implement support for this approach in their analysis tools—while providing sufficient expressiveness for the task at hand.

112 *Expressiveness of Turing Machines*

Plain automata without output symbols lack expressiveness to encode all information needed to describe the different concerns that contribute to an analysis task, for example, the specification or the environment model. An elaborated form of automata is needed that can emit arbitrary sequences of program operations to weave.

We propose YARN transducers, an instance of *abstract transducers* [88]—see Sect. 3.2. Our YARN transducers can emit program fragments (YARN) that take advantage of the expressiveness of a Turing complete programming language. The LOOM analysis composes the YARN that is emitted by different analysis components, for example, by analyses that run YARN transducers.

4.1.2 Combinatorial Explosion

Depending on the program analysis task, several (to thousands) transducers have to be executed and composed along with the program analysis iterations. In the case of software verification, these transducers can represent, for example, the program to analyze, properties to check [8], and an environment model—see Fig. 12. For a variability-aware program analysis [9, 226], additional transducers that represent features to compose can be added. In the case of model-driven test generation [109, 141], each test-goal can be represented and monitored by a separate transducer.

In general, running n automata (also abstract transducers are automata) in parallel, with the set of control states $Q_1, \ldots, Q_n$, requires to track the product $Q_\times = Q_1 \times \ldots \times Q_n$, which can be exponential in the number of automata, that is, $|Q_\times| = \prod_{i \in [1,n]} |Q_i|$. The explosion is caused by tracking the different control states of the automata explicitly in the composite state; the more control states have to be described in one composite state, the less likely it is another composite state covers a given composite state. Despite this apparent drawback (combinatorial explosion) of tracking the control state of automata explicitly (with an abstract domain that maps explicit values to variables), both symbolic abstract domains and explicit abstract domains provide performance benefits:

113 *Combinatorial Explosion*

114 *Explicit vs. Symbolic*

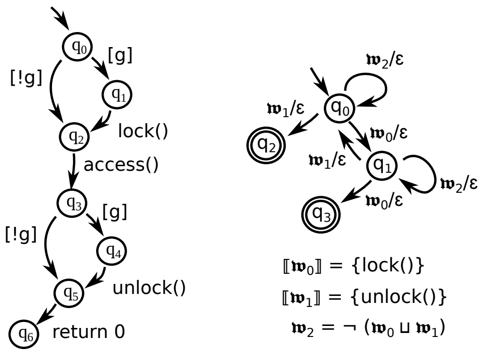
1. A *symbolic representation* of the state space can mitigate an exponential explosion of the number of abstract states. Research shows [28, 35] that representing large fragments of a verification task symbolically—for example, each loop-free fragment of a program as one formula in predicate logic [28]—can enhance the efficiency of a model-checking procedure considerably. This observation is generalized by the concept of late splitting and early joining [175, 184]: separate two states as late as possible, and join (merge) the abstract state space back as early as possible—and keep the number of abstract states as small as possible. Nevertheless, this approach relies on various assumptions, for example, that the size of the abstract state space indicates the overall efficiency.

115 *Advantages of a Symbolic Abstract Domain*

2. Tracking the current value of some variables—including variables that store the current control state of an automaton—in a concrete fashion as *explicit components* of the abstract state, can have performance benefits [68]. The current value of some variables might have to be tracked for the majority of the program paths. The program counter—also known as instruction pointer or control variable—is one of them; the value of the program counter represents the current position in the control flow of a program (control location). Always tracking the current control location in each abstract state explicitly reduces the number of abstraction refinement iterations and reduces the complexity of

116 *Advantages of an Explicit Abstract Domain*

```c
int main() {
  if (g)
    lock();
  access();
  if (g)
    unlock();
  return 0;
}
```


[g]
[!g]
lock()
access()
[g]
[!g]
unlock()
return 0
$\mathfrak{w}_2/\varepsilon$
$\mathfrak{w}_1/\varepsilon$
$\mathfrak{w}_0/\varepsilon$
$\mathfrak{w}_1/\varepsilon$
$\mathfrak{w}_2/\varepsilon$
$\mathfrak{w}_0/\varepsilon$
$[\![\mathfrak{w}_0]\!] = \{\text{lock()}\}$
$[\![\mathfrak{w}_1]\!] = \{\text{unlock()}\}$
$\mathfrak{w}_2 = \neg (\mathfrak{w}_0 \sqcup \mathfrak{w}_1)$

(a) Source code **(b)** Control flow **(c)** Specification

Figure 13: Example program and specification for discussing the problem of combinatorial explosion and the need for path sensitivity. Assume that the program variable `g` is of type `int`, initialized with a random value.

formulas that describe the set of concrete states that is represented by an abstract state [68]. In program analysis, we therefore always track the concrete value of the program counter *pc* of a verification task as an explicit component of the composite state [31, 161]. It determines the current position in the control flow and the control-flow transitions that can be taken next to derive abstract successor states.

117 *Counteractive Objectives*

Our objective is to (1) take advantage of the expressiveness and efficiency of symbolic representations—as provided, for example, by binary decision diagrams, or formulas in predicate logic—to represent sets of states of the analysis task, and (2) reduce the size and complexity of the reasoning problems that are handed over to the symbolic constraint solvers and the precision refinement procedures as much as possible, for example, by tracking the current position in the control-flow relation of the analysis task explicitly.

The problem that we are facing is that these two *objectives are counteractive*. In the following, we illustrate this based on the example verification task shown in Fig. 13. We first run a composite analysis that operates on tuples $e_i = (l_x, \varphi_z)$ of abstract states where l_x is the current control location in the control-flow relation—shown in Fig. 13b—and φ_z is a formula in predicate logic that represents a set $[\![\varphi_z]\!]$ of concrete program states:

$$\begin{tikzcd}
\boxed{(l_0, \varphi_0)} \arrow[r, "\text{[g]}"] \arrow[drr, "\text{[!g]}"'] & \boxed{(l_1, \varphi_1)} \arrow[r, "\text{lock()}"] & \boxed{e_1 = (l_2, \varphi_3)} \arrow[dr, dotted, dash] & & & \\
& & \boxed{e_2 = (l_2, \varphi_2)} \arrow[r, dotted, dash] & \text{merge} \arrow[r, dotted, dash] & \boxed{e_3 = (l_2, \varphi_2 \vee \varphi_3)} \arrow[r, "\text{access()}"] & \cdots
\end{tikzcd}$$

The analysis merges the states e_1 and e_2, resulting in state e_3 because the control-flow automaton is on the same location l_2 in both abstract states: This example satisfies both of our objectives, but we have not yet composed the specification automaton, so we do not verify anything.

We now compose the specification automaton that is shown in Fig. 13c to the task to verify that there is no call to `unlock()` without a preceding call to `lock()`. The composite analysis operates on tuples $e_i = (l_x, q_y, \varphi_z)$ of abstract states where l_x is the current control location of the control flow automaton, q_y is the current control state of the specification automaton, and φ_z represents a set of concrete program states:

$$\begin{tikzcd}
\boxed{(l_0, q_0, \varphi_0)} \arrow[r, "{[g]}"] \arrow[drr, "{[!g]}"'] & \boxed{(l_1, q_0, \varphi_1)} \arrow[r, "\texttt{lock()}"] & \boxed{e_1 = (\mathbf{l_2}, \mathbf{q_0}, \varphi_3)} \arrow[r, "\texttt{access()}"] & \cdots \\
 & & \boxed{e_2 = (\mathbf{l_2}, \mathbf{q_1}, \varphi_2)} \arrow[r, "\texttt{access()}"'] & \cdots
\end{tikzcd}$$

The analysis is not allowed to merge (join) the abstract states e_1 and e_2 because the specification automaton is in different control states q_0 and q_1 for them. The problem of this approach: Tracking the current control state of many automata in parallel leads to a combinatorial explosion of the abstract state space—the k-nary automaton product leads to an exponential number of states because the automata can be in pairwise different control states. Our first objective is violated: The amount of information encoded into the formula φ_i in predicate logic is limited, which *reduces the solver's chance to come up with Craig interpolants [190] that aid in the convergence of the analysis process*. An approach to cope with this problem is to track *sets of control states* to end up, for example, in a merged abstract state $e_m = (l_2, \{q_0, q_1\}, \ldots)$, but this leads to a new problem, which we discuss in the following section.

4.1.3 Path Sensitivity

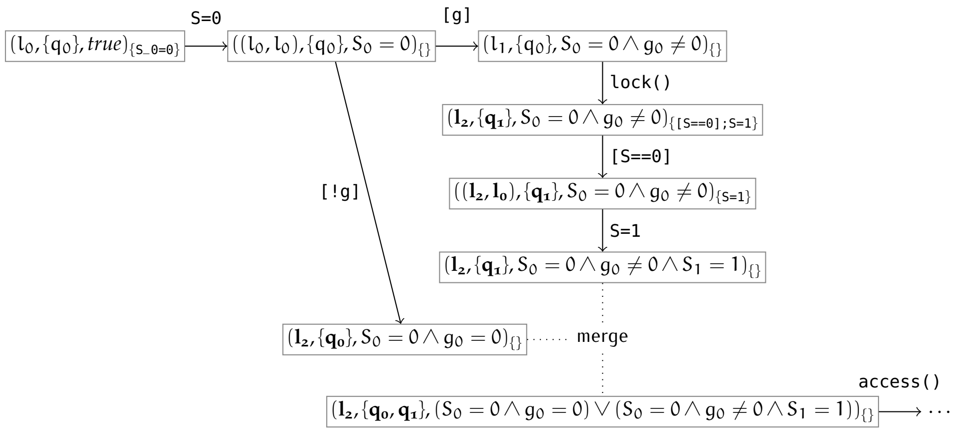
Why not merge abstract states independent of the current control state they are in and use *sets* of control states? We end up with the following result:

$$\begin{tikzcd}
\boxed{(l_0, \{q_0\}, \mathit{true})} \arrow[r, "{[g]}"] \arrow[drr, "{[!g]}"'] & \boxed{(l_1, \{q_0\}, g_0 \neq 0)} \arrow[r, "\texttt{lock()}"] & \boxed{(\mathbf{l_2}, \{\mathbf{q_1}\}, g_0 \neq 0)} \arrow[dr, dotted, no head] & & & \\
 & & \boxed{(\mathbf{l_2}, \{\mathbf{q_0}\}, g_0 = 0)} \arrow[r, dotted, no head] & \text{merge} \arrow[r, dotted, no head] & \boxed{e_m = (\mathbf{l_2}, \{\mathbf{q_0}, \mathbf{q_1}\}, g_0 = 0 \vee g_0 \neq 0)} \arrow[r, "\texttt{access()}"] & \cdots
\end{tikzcd}$$

The problem here is that we lose information, more precisely, path sensitivity is lost. Given the abstract state e_m, we know that the specification automaton is either in control state q_0 or q_1 when the control-flow location l_2 is reached; but we do not know which branch has been taken to arrive at the control state q_1 of the specification automaton. Also, the formula φ_z does not provide any information for ensuring path sensitivity and recovering the paths. The result is a *false alarm* if `unlock()` is called later for branch `[g]` while still being in control state q_0 of the specification automaton.

We need a technique for ensuring path sensitivity while keeping the explosion of the abstract state space manageable. Making an analysis path-sensitive requires to track information about all relevant branchings and values of the referenced data locations; we would have to enrich the abstract domain of our transducer analysis such that it would be able to encode all this information. In the end, this abstract domain would have to provide a similar expressiveness as, for example, the predicate domain [28, 122].

We choose a novel route and take advantage of the services that are provided by the Loom analysis to introduce additional control flow transitions by weaving. That is, the analysis that is responsible for executing an automaton (or transducer) delegates the encoding of the facts that are relevant for path sensitivity by emitting the respective Yarn to weave: The emitted Yarn introduces and initializes a program variable `s` that represents the current control state of the specification automaton, and it prepends code with guards and assignments to the state variable on all state transitions:


$(l_0, \{q_0\}, \mathit{true})_{\{S\_0=0\}}$
S=0
$((l_0, l_0), \{q_0\}, S_0 = 0)_{\{\}}$
[g]
$(l_1, \{q_0\}, S_0 = 0 \wedge g_0 \neq 0)_{\{\}}$
lock()
$(\mathbf{l_2}, \{\mathbf{q_1}\}, S_0 = 0 \wedge g_0 \neq 0)_{\{[S==0];S=1\}}$
[S==0]
[!g]
$((\mathbf{l_2}, \mathbf{l_0}), \{\mathbf{q_1}\}, S_0 = 0 \wedge g_0 \neq 0)_{\{S=1\}}$
S=1
$(\mathbf{l_2}, \{\mathbf{q_1}\}, S_0 = 0 \wedge g_0 \neq 0 \wedge S_1 = 1)_{\{\}}$
$(\mathbf{l_2}, \{\mathbf{q_0}\}, S_0 = 0 \wedge g_0 = 0)_{\{\}}$
merge
access()
$(\mathbf{l_2}, \{\mathbf{q_0}, \mathbf{q_1}\}, (S_0 = 0 \wedge g_0 = 0) \vee (S_0 = 0 \wedge g_0 \neq 0 \wedge S_1 = 1))_{\{\}}$
$\cdots$

We can see that the third component—a formula in predicate logic, in single-static assignment form—of an abstract state is sufficient to provide path sensitivity; the automaton analysis does not prevent any merges of abstract states. The path sensitivity is achieved by a *hybrid* configuration of an explicit-value analysis and a symbolic analysis: The analysis that keeps track of the current position in the control-flow automaton still uses an explicit value.

4.1.4 Empirical Evidence

Literature provides no substantial empirical evidence on the efficiency of different encodings of the control state of automata. We aim to get a better understanding of how symbolic or concrete encoding of current control states impacts the performance of a software model checker. We study whether the state of the formal specification (or the environment) should be modeled with *explicit values* or *symbolically*, or if a *hybrid* approach would yield the best performance for the verification task at hand. We hypothesize that increasing the chance of merging abstract states can increase the performance of a verifier, but state merging might not always be positive [128, 175].

Several components of a verification engine (variables for an empirical study) interact with the choice of the encoding scheme: An abstraction precision refinement procedure might have to conduct more or fewer refinement iterations; whereas larger abstraction block sizes [28, 35] might reduce the number of refinement iterations needed. Delegating information to encode between analysis components, based on emitting Yarn, can shift the analysis performance in unexpected ways: If the transducer analysis, which always operates with the full precision (it always encodes the current control state of the automaton) delegates the task of encoding he current control state to another analysis component by emitting Yarn, an analysis that is based on abstraction precision refinement could come into play: The positive effect of using state variables for encoding the control state boils down to delegating to an analysis that uses abstraction refinement, and can thus keep the abstract state space compact.

4.2 YARN

Before we introduce the LOOM—our mechanism for orchestrating and conducting the task of composing sequences of control transitions by weaving—in the next section, we introduce the notion of YARN, that is, the information that any program analysis component can emit for weaving. We start by formally defining YARN and will then (1) discuss different considerations that led to our final design choices, and will (2) present a set of operations for dealing with YARN in the context of a program analysis.

Definition 51: YARN

YARN describes a set of sequences of program operations for implementing parts of a specific program concern. Formally speaking, YARN is a tuple $\mathfrak{y} = (h, (G_{\mathfrak{y}}, L_{en}, L_{ex})) \in \mathfrak{Y}$, with the set of yarns $\mathfrak{Y} = H \times 2^{L \times Op \times L} \times 2^{L} \times 2^{L}$. A YARN $\mathfrak{y}$ consists of a control transition relation $G_{\mathfrak{y}} \subseteq L \times Op \times L$ to weave, which is mapped to a program concern $h \in H$ from the set of all program concerns H. The set $L_{en} \subseteq L$ denotes the *control entry locations*, and the set $L_{ex} \subseteq L$ denotes the *control exit locations*. Both set L_{en} and set L_{ex} must not be empty.

Similar to a control-flow automaton, a YARN describes the control transition relation of a program. Each control-flow automaton can be described as one YARN $\mathfrak{y} \in \mathfrak{Y}$: The set of exit locations of a $CFA = (L, l_0, G)$ *can* be defined by all control locations in L from that no further transitions leave. That is, $L_{ex}(\mathfrak{y}) = \{l \mid (\cdot,\cdot,l) \in G \wedge (l,\cdot,\cdot) \notin G\}$. But any location in the transition relation can potentially be an entry location, an exit location, or even both. Imagine, for example, a YARN $\mathfrak{y}$ with the transition relation $G_{\mathfrak{y}} = \{(q_1,\cdot,q_2),(q_2,\cdot,q_3),(q_3,\cdot,q_1)\}$, which describes a loop: What is an entry location?

YARN that an analysis emits can have one of two *delta types* $\triangle = \{\triangleright, \triangleleft\}$. The *additive* YARN $\triangleright$ is inserted at a specific position in the control flow; the *substituting* YARN $\triangleleft$ replaces a specific control transition that has not yet been woven, that is, it relies on a lookahead to the YARN that is intended to get woven next—see Sect. 4.3.3. The ability to substitute control transitions can be used, for example, for (1) mutation-based test generation, (2) to replace calls to external methods by an actual implementation, or (3) for program slicing, which skips control transitions that are irrelevant for the reasoning task at hand. For this work, we use and further discuss *additive* YARN *only*, and keep substituting YARN for future work—this is why we have kept the YARN type out from the formal definition for now. 118 *Delta Types* 119 ⚠

A YARN is closely related to a model program [247], a notion that is in the context of model-based test generation [246]. The set of accepting control points of a model program corresponds to the set of exit locations of a YARN. Model programs are not mapped to specific concerns explicitly. The notion of advice is the concept from aspect-oriented programming that has the closest relationship to YARN. Nevertheless, an advice is defined on the level of source code and not at the level of control-flow transitions. 120 *Related Concepts*

4.2.1 Dependencies between Yarn

As several analysis components can provide Yarn to weave in one analysis step, the weaving mechanism must decide which Yarn to weave next. To make functional dependencies between Yarn for different concerns clear, an explicit means to define dependencies is needed—we cannot determine this automatically since the concatenation of two sequences of control transitions is not commutative without rewriting them [10]. Different orderings of composing a set of sequences of control transitions can lead to different semantics of the result—semantics of the control-flow relation that represents the composed analysis task.

Concern Dependency Graph. In case the dependency information that can be determined automatically is not sufficient for a deterministic choice and a total order, we must take an additional ordering relation into account to make the analysis process deterministic. We organize all concerns from that the transition system of the system to analyze can be composed of in a concern dependency graph—see Fig. 14 for an illustration of such a graph:

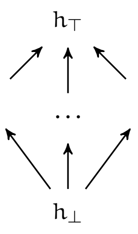


Figure 14: A concern dependency graph yields a partial order.

Definition 52: Concern Dependency Graph

A *concern dependency graph* is a graph that determines the dependencies between concerns from a set H of concerns. We define it as a tuple $\mathbb{H} = (H, \eta, h_\top, h_\perp)$, with the set H of concerns, a transitive *concern dependency relation* $\eta \subseteq H \times H$, the *top concern* $h_\top$, and the *bottom concern* $h_\perp$. The top concern has the *lowest* priority in the weaving process: All other concerns are (transitively) dependent on it and it depends on no other concern. The bottom concern has the *highest* priority in the weaving process: It depends on all other concerns an no concern depends on the bottom concern.

Weaving Yarn with the *highest priority is finished first* before any other Yarn with lower priority, from a queue with Yarn to weave, becomes woven. Since we do *not allow cycles* in a concern dependency graph we can compute a partial order of program concerns.

Proposition 9. Weaving a Yarn for a concern h_a is always finished before another Yarn for the concern h_a becomes woven.

Proof. A topological ordering defines the priorities of concerns to weave: The dependency graph does, by definition, not have cycles. The weaving of the Yarn from a concern h_a can be intercepted by Yarn of a concern h_b if that has higher priority, weaving Yarn of concern h_b could then be intercepted by Yarn from concern h_a, meaning that concern h_a has a higher priority than concern h_b: Contradiction (and forbidden cycle in the dependency graph). The weaving of Yarn for a concern h_a will always be finished before another Yarn for this concern is woven. □

Related Concepts

121 The problem of dependencies between code to weave has already been identified in context of traditional aspect weaving: A weaving schedule is derived [163] from a weaving-interaction graph if this graph has a topological

ordering. Similar orderings are also needed by term rewriting systems to ensure their termination [92, 107].

Mapping of* Yarn *to Concerns. To be able to take the dependencies of a YARN for correct weaving into account, each YARN must be mapped to a concern $h \in H$. We additionally map each control location to a program concern—and all control locations of a given YARN map to the concern of the YARN. The function $H(l) \in H$ provides the concern to which a control location $l \in L$ belongs to. We add an attribute to each control location $l \in L$ to easily identify this mapping: $l.h = H(l) \in H$.

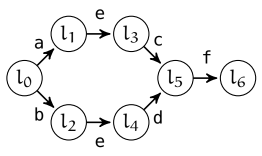
4.2.2 Role of Labeling

We now motivate and discuss the role of labeling the positions in the control flow of an analysis task to end up in an effective program analysis procedure. Two factors are essential: (1) It should be possible to compose programs that take advantage of the full Turing completeness of the underlying programming language—which arises from conditional (backward) *jumps* to other positions. Moreover, (2) it must be possible to identify common positions in the control-flow to allow for *joining* the state space back after branching—if this does not affect the soundness of the analysis unintentionally.

Path Explosion Problem. The language $\mathcal{L}(\overline{P})$ of a program $\overline{P}$ is the set of sequences of program operations that can be constructed syntactically by traversing its control-flow automaton—see Sect. 2.1.4 for more details. Such a language can consist of infinitely long and exponentially (in the number of branchings) many words: This reflects the *path explosion problem* [48, 175], which is one of the problems that make model checking hard. To not sacrifice completeness (by not considering all possible execution traces), the state space that has to be explored should be merged [175] from time to time.

To identifying *common points in the control flow* of the system under analysis, we require that the YARN that is emitted by an analysis for weaving is enriched with control locations: Given a sequence $\overline{o} = \langle op_1, \dots, op_n \rangle \in Op^*$ of program operations, we can label each operation op_i within this sequence with a control location $l_i \in L$ that is reached after executing the operation. The analysis *can* then merge two abstract states if they belong to the same control location (same labeling), which is the case at positions where the control flow merges. Joining abstract states, for that the control flow of the analysis task does not join, would *sacrifice soundness*: Information from the control transition relation is lost.

Proposition 10. Merging two abstract states *affects* soundness, that is, it can lead to wrong statements about the program under analysis if they map to different control locations.



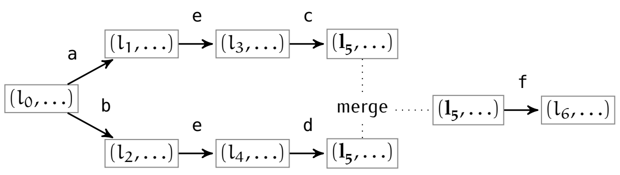
$$\begin{tikzcd}
 & l_1 \arrow[r, "e"] & l_3 \arrow[dr, "c"] & & \\
l_0 \arrow[ur, "a"] \arrow[dr, "b"'] & & & l_5 \arrow[r, "f"] & l_6 \\
 & l_2 \arrow[r, "e"'] & l_4 \arrow[ur, "d"'] & &
\end{tikzcd}$$

Figure 15: Split and join

Proof. The correctness of the proposition follows from the well-known need of having a flow-sensitive program analysis (and also from the need of having a path sensitive analysis). Take, for example, the control-flow relation $G_b \subset L \times Op \times L$ that is illustrated in Fig. 15: The analysis must not join the abstract states on control location l_2 and location l_3 since it would otherwise allow for a sequence of program operation $\overline{o} = \langle b, c, f \rangle$ which is not in the language original control-flow relation, that is, $\overline{o} \notin \mathcal{L}(G_e)$. $\square$

The current control location, that is, the position in the control-flow relation of the analysis task, is stored as part of each abstract state that the software model checker constructs. It is precisely the control location $l' \in L$ that is reached *after* the program operation $op \in Op$ on a control-flow transition $g = (l, op, l') \in G$ was conducted. Given an abstract composite state $e = (l, \ldots)$ and the control transition g based on that an abstract successor state $e' = (l', \ldots) \in E$ is computed, that is, the transfer $e \overset{g}{\rightsquigarrow} e'$ is conducted. Abstract state e', which is the result of applying operation op to abstract state e, maps to the exit point l' of the control transition g as its current control location. (Please note, that this also depends on the *direction of the program analysis*: For a backward reachability analysis, the entry location should get stored.) The model checker takes, apart from other information that is tracked in abstract states, the stored control locations into account to decide whether or not to merge the state space.

Example 6. Given the control-flow automaton in Fig. 15, a program analysis constructs the following abstract reachability graph. The current control location is stored along with each abstract state and is used to decide whether or not to merge the abstract state space:



Fresh Control Locations. Despite the importance of having and knowing common control locations, it is sometimes important—to not make wrong statements about the system to analyze—to assign control locations that are guaranteed to not having been used before in the control flow of the analysis task. We use the notion of fresh control locations: 122

$l^\# = \mathsf{fresh}()$

Definition 53: Fresh Control Location

A control locations is a *fresh control location* $l^\#$—with respect to a given analysis task, which is represented by a program $\overline{P}$—if it has not been used before for the given task, that is, $l^\# \notin L_{\overline{P}}$. Fresh control locations are generated by the 0-ary and nondeterministic operator $\mathsf{fresh} \to L$ that returns a new control location $l^\# \in L$ and marks it used for the given program, that is, $L'_{\overline{P}} = L_{\overline{P}} \cup \{l^\#\}$. A consecutive sequence of k calls to the operator fresh generates a set $\{l^\#{}_1, \ldots, l^\#{}_k\} \subseteq L$ of fresh control locations.

Based on the concept of fresh control locations, we introduce an operator for lifting a sequence of program operations to an equivalent sequence of control-flow transitions:

Definition 54: Lifting to Control Transitions

The operator $\mathsf{lift}_G : Op^* \to 2^{L \times Op \times L}$ lifts a given finite sequence of program operations to a sequence of control transitions. Given the sequence $\overline{o} = \langle op_1, \dots, op_n \rangle$, the operator returns the *ordered* set of control transitions $\{(l_0, op_1, l_1), \dots, (l_{n-1}, op_n, l_n)\}$, where each control location $l_i = \mathsf{fresh}()$ is a fresh control location. The operator extends naturally to sets of sequences of program operations.

We define a separate operator that creates YARN from a given set of sequences of program operations for a given concern:

Definition 55: Lifting to YARN

The operator $\mathsf{lift}_{\mathfrak{Y}} : H \times 2^{Op^*} \to \mathfrak{Y}$ lifts a given set of finite sequences of program operations $\widehat{\overline{o}} = \{\overline{o}_1, \dots, \overline{o}_n\}$ for a given program concern h to a YARN. The resulting YARN $\mathfrak{y} = (h, G_{\mathfrak{y}}, L_{en}, L_{ex})$ is created from the control flow transitions $G_{\mathfrak{y}} = \bigcup_{\overline{o} \in \widehat{\overline{o}}} \mathsf{lift}_G(\overline{o})$ that are lifted from the set of sequences of program operations, and the set of entry locations L_{en} and exit locations L_{ex} that correspond to the first, respective last, control locations in the created sequences of control transitions.

Proposition 11. Transforming a set of sequences of program operations $\widehat{\overline{o}} \subseteq Op^*$ into a control transition relation $G_{\widehat{\overline{o}}}$ by assign fresh control locations along each sequence $\overline{o} \in \widehat{\overline{o}}$ produces an equivalent result, that is, $\mathcal{L}(G_{\widehat{\overline{o}}}) = \widehat{\overline{o}}$, with $G_{\widehat{\overline{o}}} = \bigcup_{\overline{o} \in \widehat{\overline{o}}} \mathsf{lift}_G(\overline{o})$, where a different sequence of operations $\overline{o}$ results in a different sequence of control location.

Proof. For each sequence of program operations $\overline{o}$ a new fresh initial control location $l_0 = \mathsf{fresh}()$ is generated. We denote this set of initial control locations by L_0. A control transition relation with a set of initial control locations can be transformed into one with a single initial control location by generating a fresh control location $l_{00} = \mathsf{fresh}()$, and by adding neutral control transitions (with the neutral program operation nop) from there to each of the control locations in the set L_0. The language $\mathcal{L}(G_x)$ of a control transition relation $G_x \subseteq L \times Op \times L$ is the set of sequences of control transitions that are well-founded, that is, each sequence starts the initial control location. Each sequence of control transitions can be projected to a sequence of program operations. The transformation from sets of sequences of program operations to the control transition relation ensures that the control locations that are generated for one sequence $\overline{o} \in \widehat{\overline{o}}$ are disjoint to all locations that are generated for one of the other sequences in the set. That is, a transition (l, op, l') can only reach a location l' that was generated for the sequence of operations that it represents. The construction process ensures, that for each sequence of program operations $\langle op_1, \dots, op_n \rangle$ also a corresponding sequence of control transitions $\{(l_0, op_1, l_1), (l_1, op_2, l_2), \dots, (l_{n-1}, op_n, l_n)\}$ is created. $\square$

From previous considerations, we derive the following definition:

Definition 56: Sound Labeling

Transforming a given set of sequences of program operations—denoted by the language $\mathcal{L}_1 \subseteq Op^\infty$—into a control-transition relation $G_2 \subseteq L \times Op \times L$ must maintain the language, that is, it must hold that $\mathcal{L}(G_2) = \mathcal{L}_1$. A labeling of control locations that satisfies this requirement is called *sound*.

4.2.3 Yarn Composition

So far, we have discussed that an analysis can emit Yarn to compose into the control transition relation of the analysis task. Providing Yarn with labeled program operations helps to identify common positions in the corresponding control flows, and thus allows to merge the abstract state space at these points—to mitigate the path explosion problem.

Based on the sequences of program operations that have been observed so far (and possibly based on a lookahead), *several* analyses can provide Yarn for weaving: The resulting transition relation, of the analysis task, is composed of Yarn for *different concerns*, with dependencies to each other. 123 Each analysis that emits Yarn can ensure only the consistency of the labeling it provides. *Consistency of labeling* among different concerns is not ensured—typically, one analysis provides Yarn for one concern.

Consistency of Labeling

Yarn *Composition by Weaving.* Before we continue to discuss which labeling of control locations is chosen for composing Yarn for different concerns, we describe how composition by weaving is conducted.

Definition 57: Inserting Yarn

An analysis emits Yarn $\mathfrak{y} = (h, G_\mathfrak{y}, L_{en}, L_{ex})$ for weaving after observing a sequence $\bar{\sigma} = \langle (l_0, op_1, l_1), \dots (l_{n-1}, op_n, l_n) \rangle$ of control-flow transitions and corresponding program operations and in the presence of a matching lookahead $\hat{\sigma}$. Weaving of Yarn $\mathfrak{y}$ is conducted by redirecting the control flow: Instead of conducting the transfer to control location l_n, the flow is redirected to the set of entry locations L_{en}, and from the exit locations L_{ex} back to control location l_n.

Whether or not weaving Yarn affects the control flow of the transition system of the analysis task in an unexpected way is determined by the labeling of the control locations:

Definition 58: Sound Weaving

Weaving Yarn with an unspecified labeling of control locations is called *sound* if it results in a control-transition relation G_1 with the same language than weaving the Yarn with a fresh labeling, resulting in a control-flow relation G_2, that is, $\mathcal{L}(G_1) = \mathcal{L}(G_2)$.

Compound Control Locations. Which labeling is needed to identify common points in the *composition* result while maintaining a sound weaving result? Consider the following example:

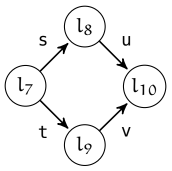
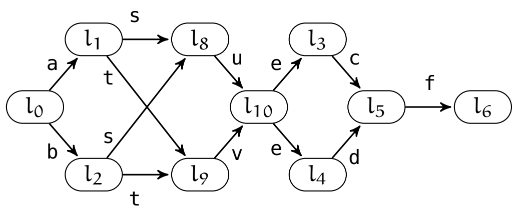


Figure 16: YARN to weave

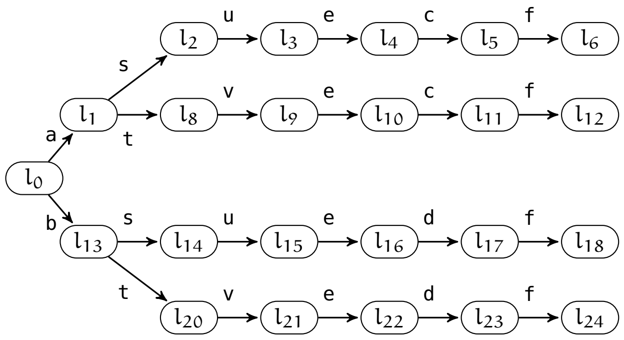
Example 7. Take the control-flow automaton CFA_h that is illustrated in Fig. 16 and its transition relation G_h to compose by weaving on the transition relation G_b of the control-flow automaton CFA_b that is illustrated in Fig. 15 as the base. The transitions of G_h should be added just before each control transition $\overset{e}{\to}$ of the base transition relation G_b. The labels for transitions of G_b are from the interval $[l_0, \dots, l_6]$, and the labels for G_h are from $[l_7, \dots, l_{10}]$, that is, the intervals are disjoint and each label is unique among the two transition relations. Nevertheless, producing YARN based on disjoint sets of labels does not guarantee a sound composition of two transition relations; we (would) get the following result:



The composition is unsound because, for example, the word $\langle \mathtt{a}, \mathtt{t}, \mathtt{v}, \mathtt{e}, \mathtt{d}, \mathtt{f} \rangle$ is in the language of the composed transition relation. The transition relation G_h is woven two times, at two control locations $\{l_1, l_2\}$ to the control-flow of the analysis task. Weaving is conducted in the context on two different sets of program traces $\{\langle \mathtt{a} \rangle\}$ and $\{\langle \mathtt{b} \rangle\}$ that have been observed until reaching the respective weaving location.

Another example illustrates the problem of assigning fresh labels only:

Example 8. In this example, we compose the control flows as outlined in Example 7, except that we assign fresh labels to the YARN to compose. Composing the two transition relations by weaving with a fresh labeling of control locations results in an analysis task with the control-flow relation $G_\times$ and the language $\mathcal{L}(G_\times) = \{\langle \mathtt{a}, \mathtt{s}, \mathtt{u}, \mathtt{e}, \mathtt{c}, \mathtt{f} \rangle, \langle \mathtt{a}, \mathtt{t}, \mathtt{v}, \mathtt{e}, \mathtt{c}, \mathtt{f} \rangle, \langle \mathtt{b}, \mathtt{s}, \mathtt{u}, \mathtt{e}, \mathtt{d}, \mathtt{f} \rangle, \langle \mathtt{b}, \mathtt{t}, \mathtt{v}, \mathtt{e}, \mathtt{d}, \mathtt{f} \rangle\}$:



We can see that the analysis cannot merge the state space because each control location in the resulting transition system has assigned a fresh label. Information on common points in the control flow of the original transition relation is lost.

Inserting a sequence of program transitions with fresh control locations does not affect soundness, but can lead to a path explosion. To *mitigate the path explosion problem* and *ensure the soundness* of the analysis of the composed transition relation, we have to provide a *labeling for identifying* common positions in the *composed* transition relation.

Our proposed solution is to create compound control locations to end up in a result that is in the sweet spot of sound weaving and an efficient state-space exploration that does not jeopardize completeness of the analysis procedure. Before weaving YARN of a given concern, the YARN is re-labeled with compound control locations:

Definition 59: Compound YARN Operator $\rtimes_\#$

The *compound* YARN operator $\rtimes_\# : \mathfrak{Y} \times L \rightarrow \mathfrak{Y}$ creates a new YARN with all control locations compound with a given control location. Given an YARN $\mathfrak{y} = (h, G_\mathfrak{y}, L_{en}, L_{ex})$ and a control location $l_c \in L$ to compose with, the operator returns the YARN $\mathfrak{y}' = (h, \{((l_c, l), op, (l_c, l')) \mid (l, op, l') \in G\}, \{l_c\} \times L_{en}, \{l_c\} \times L_{ex}) \in \mathfrak{Y}$. The application of the operator is written either infix $\mathfrak{y} \rtimes l_c$ or in prefix notation $\rtimes(\mathfrak{y}, l_c)$.

Proposition 12. Creating compound YARN with the operator $\rtimes_\#$ maintains the language of the original YARN, that is

$$\mathcal{L}(\mathfrak{y}') = \bigcup_{l_c \in L} \mathcal{L}(\mathfrak{y} \rtimes l_c).$$

Proof. Let $\mathfrak{y}_1 = (h, G_1, L_{en1}, L_{ex1})$ and $\mathfrak{y}_2 = (h, G_2, L_{en2}, L_{ex2})$ be two YARNS with $\mathfrak{y}_2 = \mathfrak{y}_1 \rtimes_\# l_c$. We have to show that each sequence of program operations $\bar{o} \in \mathcal{L}(\mathfrak{y}_1)$ is in $\mathcal{L}(\mathfrak{y}_2)$, and there is no sequence in $\mathcal{L}(\mathfrak{y}_2)$ that is not in $\mathcal{L}(\mathfrak{y}_1)$. The construction process guarantees that for each entry location of YARN $\mathfrak{y}_1$, there is a corresponding entry location (l_c, l) in the set of entry locations of YARN $\mathfrak{y}_2$. Furthermore, it guarantees that for each entry location (l_c, l) there is a corresponding location l in the set L_{en1}. The same applies to the set of exit locations. This also applies for the set of control-flow transitions: There is no control transition $((l_c, l), op, (l_c, l')) \in G_2$ for that exists no transition $(l, op, l') \in G_1$, and the other way around.

For each sequence of program operations $\bar{o} = \langle op_1, \ldots \rangle \in \mathcal{L}(\mathfrak{y})$ exists, by construction, a sequence $\langle (l, op_1, l'), \ldots \rangle$ of control transition that is well-founded in the transition relation $G_\mathfrak{y}$ of a YARN $\mathfrak{y}$. Since each transition $(l, op, l') \in G_1$ was transformed to $((l_c, l), op, (l_c, l')) \in G_2$ also each sequence of control transitions $\langle (l_0, op_1, l_1), (l_1, op_2, l_2), \ldots \rangle$ that is well-founded in the transition relation G_1 has a corresponding well-founded sequence $\langle ((l_c, l_0), op_1, (l_c, l_1)), ((l_c, l_1), op_2, (l_c, l_2)), \ldots \rangle$ in the transition relation G_2. The sequences of program operations are maintained. □

Following examples illustrate the creation and use of compound YARN:

Example 9. Given the YARN $\mathfrak{y}_1 = (\cdot, G_\mathfrak{y}, \{l_1\}, \{l_3\})$, with $G_\mathfrak{y} = \{(l_1, op_a, l_2), (l_2, op_b, l_3)\}\})$, that represents the following transition relation:

$$\begin{tikzcd} l_1 \arrow[r, "op_a"] & l_2 \arrow[r, "op_b"] & l_3 \end{tikzcd}$$

A composition of $\mathfrak{y}_1$ with control location l_5, that is, $\mathfrak{y}_1 \rtimes_\# l_5$, results in a new YARN that represents following transition relation:

$$\begin{tikzcd} (l_5, l_1) \arrow[r, "op_a"] & (l_5, l_2) \arrow[r, "op_b"] & (l_5, l_3) \end{tikzcd}$$

Example 10. We now compose the control flows similar as in Example 7, except that we use compound control locations. To weave the transition $l_7 \overset{s}{\to} l_8$ on location $l_1 \in L$, we create the control transition $(l_7) \overset{s}{\to} (l_1, l_8)$. The result is the following transition system:

$$\begin{tikzcd}
 & & (l_1, l_8) \arrow[dr, "u"] & & & & \\
 & l_1 \arrow[ur, "s"] \arrow[r, "t"'] & (l_1, l_9) \arrow[r, "v"'] & (l_1, l_{10}) \arrow[r, "e"] & l_3 \arrow[dr, "c"] & & \\
l_0 \arrow[ur, "a"] \arrow[dr, "b"'] & & & & & l_5 \arrow[r, "f"] & l_6 \\
 & l_2 \arrow[r, "s"] \arrow[dr, "t"'] & (l_2, l_8) \arrow[r, "u"] & (l_2, l_{10}) \arrow[r, "e"'] & l_4 \arrow[ur, "d"'] & & \\
 & & (l_2, l_9) \arrow[ur, "v"'] & & & &
\end{tikzcd}$$

Context. An analysis emits YARN for weaving after observing a sequence of program operations $\bar{\tau} = \langle op_1, \dots, op_n \rangle$ from a corresponding sequence $\langle (l_0, op_1, l_1), \dots, (l_{n-1}, op_n, l_n) \rangle$ of control transitions, which we call the *context*. The analysis expects that the YARN is woven by redirecting the control flow from location l_n to the entry locations of the given YARN. The weaving and composition process must ensure that the transitions from the emitted YARN are only reachable if the context is satisfied, that is, exactly after traversing along the sequence of program operations described by $\bar{\tau}$.

The labeling of the control locations of the YARN to weave determines whether it can be reached for the given context only. One approach to end up in such a labeling of YARN is to create a compound YARN with a fresh control location. We define a separate operator for this purpose:

Definition 60: Relabeling Operator $\pm_\#$

Given a YARN $\mathfrak{y}$, the unary YARN relabeling operator $\pm_\# : \mathfrak{Y} \to \mathfrak{Y}$ produces a new YARN for that all control locations are replaced by control locations that have not yet been used in the transition relation of an analysis task while maintaining the structure of the transition relation. A call $\pm_\#(\mathfrak{y})$ returns a new YARN $\mathfrak{y}' = \mathfrak{y} \rtimes l^\#$ for that all control locations are compound with a fresh control location $l^\# = \mathsf{fresh}()$. The operator ensures that the language is preserved, that is, $\mathcal{L}(\mathfrak{y}) = \mathcal{L}(\mathfrak{y}')$.

Proposition 13. Creating a compound YARN $\mathfrak{y}_x = \mathfrak{y} \rtimes_\# l_c$ from a YARN $\mathfrak{y}$ for weaving it into the transition relation in a context $\bar{\tau}$ is sound if the control location l_c to compose with uniquely identifies the context $\bar{\tau}$. Note that different (fresh) control locations (labels) can identify the same context.

Proof. To show soundness of composition by weaving, we will show—according to Def. 124—that composing YARN η with freshly labeled control locations yields the same language of the analysis task than weaving the compound YARN $\eta \rtimes_{\#} l_c$, where the control location l_c uniquely identifies the context $\bar{\tau}$ (and nothing else). The effect of weaving such a compound YARN is clear: (1) The entry locations of the YARN are only reachable in the presence of the context. Moreover, (2) none of the transitions of the compound YARN can enter a control location that is not composed with the context identifier (it would otherwise not be part of the YARN), that is, no control locations outside the compound YARN are reachable. □

4.2.4 Identification of Cycles

A program that is written in a Turing-complete programming language can contain loops, and its language can consist of infinitely long program traces. A model checking procedure has to be able to identify loop heads to compute abstractions that overapproximate their behavior to end up with a finite model of the program.

124 *Loop Head*

A *loop head* is a control location $l_{\circlearrowleft} \in L$ from that a sequence of control transitions can appear (syntactically) infinitely often on a program control path. We assume that any recursion is eliminated and replaced by a semantically equivalent loop. We require that each loop head $l_{\circlearrowleft} \in L$ that is emitted as part of YARN to weave has an attribute $l.\mathit{loophead} \in \mathbb{B}$ set that marks it as such, that is, $l_{\circlearrowleft}.\mathit{loophead} = \mathit{true}$. The set of all loop heads is denoted by $L_{\circlearrowleft} \subseteq L$. Different established algorithms for identifying loop heads of programs [28, 157, 220] can be applied.

4.2.5 Traversal Order

A program analysis algorithm is guided by a traversal strategy that defines which abstract state to explore next, that is, how to traverse the state space. Depending on the objective of an analysis, a different traversal strategy might be preferred: Is the goal to identify bugs early, or to arrive at a coverage of the state space fast?

For software model checking and data-flow analysis we typically use a *wait-at-meet* traversal strategy [78]: Given a control location l_m, all abstract states that belong to its predecessor locations should be explored first before abstract states for l_m are explored. See Fig. 17 for an illustration of this scenario. An analysis can merge two abstract states if there is either (1) no successor state for both of them in the set of reached states, or (2) if it removes already computed successor states from the set of reached states. Exploring the state space based on the wait-at-meet order allows merging abstract states while avoiding to discard already computed abstract states, and with it not to waste spent computing resources.

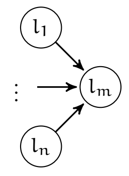


Figure 17: Wait at meet

The traversal strategy of the CPA algorithm is configured by providing an implementation of the operator choose—see Sect. 2.5.1 for more details. Typically, we implement the wait-at-meet strategy by assigning a *wait-at-meet order number* $l.\mathit{wmo} \in \mathbb{Z}$ to each control location $l \in L$ in the transition relation of the system to analyze. Ordering all control locations in reverse post-order [78] is one approach for ending up in a wait-at-meet order.

125 *Lexicographical Ordering of Control Locations*

Composing additional control transitions to a transition system invalidates an existing wait-at-meet ordering. We solve this issue trivially by using a *compound wait-at-meet ordering*: Given a compound location $l_\times$ that is composed of a sequence $\langle l_1, \ldots, l_n \rangle$ of control locations, we assign $l_\times.\mathit{wmo} = (l_1.\mathit{wmo}, \ldots, l_n.\mathit{wmo}) \in \mathbb{Z}_1 \times \ldots \times \mathbb{Z}_n$. Please note, that also a non-compound control location can be considered to be a compound location, composed of only one control location. The wait-at-meet ordering of compound control locations is then *lexicographical*.

4.2.6 Pure Yarn

Some control locations that can be found along a transition relation of an analysis task have *special semantics*. For example, *target locations* that signal the violation of a set of properties when being reached. Another example are *bottom locations* on that the state-space traversal should stop—caused, for example, by specific system calls (program operations) that cause the process to terminate immediately. This type of location is of particular importance for analyses that have to signal that a special analysis state is reached *when arriving at particular control locations* in the emitted Yarn.

126 *Target Locations and Bottom Locations*

To signal target locations, we map a set of properties to each control location of an analysis task: We assign the attribute $l.\mathit{violated} \subseteq S$ to each control location $l \in L$, which corresponds to the map $\mathsf{violated} : L \to 2^S$ of control locations to violated properties. To signal that a given control location l is a bottom location, we assign the attribute $l.\mathit{bottom} \in \mathbb{B}$, which corresponds to the function $\mathsf{bottom} : L \to \mathbb{B}$. A bottom location is also denoted by $l_\bot$.

The information from control locations can be transformed into information on the labels of transitions between control locations. For example, the transition relation $l_1 \overset{op_1}{\to} l_2$, with $l_2.\mathit{bottom} = \mathit{true}$, can be transformed into $l_1 \overset{op_1}{\to} l_3 \overset{op_2}{\to} l_4$, where program operation op_2 signals that the bottom location has been reached. We call this transformation purification:

Definition 61: Yarn Purification

Purifying a Yarn denotes the process of transforming a given Yarn by moving all special semantics from control locations into the labels of transitions between control locations. Purification is implemented by the operator $\mathsf{pure} : \mathfrak{Y} \to \mathfrak{Y}$ which takes a Yarn $\mathfrak{y}$ as input and produces a *pure* Yarn $\mathfrak{y}'$ as output.

4.2.7 Yarn Language

We now take the language-theoretic perspective on Yarn, define what the language of a Yarn is, and provide different operators to deal with different Yarn and their languages. We assume that each Yarn that is provided as one of the arguments of the different operators is *pure*. Similar to the language of

127 *On Pure Yarn*

control-flow automaton and its transition relation, the language of a Yarn is a set of sequences of program operations:

Definition 62: Yarn Language

The *language* of a given Yarn $\mathfrak{y} = (\mathfrak{h}, G_\mathfrak{y}, L_{en}, L_{ex})$ is the set $\mathcal{L}(\mathfrak{y}) \subseteq Op^\infty$ of sequences of program operations that start in one of the control locations in $L_{en} \subseteq L$ and that are well-founded in the control transition relation $G_\mathfrak{y} \subseteq L \times Op \times L$. The language $\mathcal{L}(\mathfrak{y})$ consists of the set of *finite* sequences and *countable infinite* sequences of program operations that terminate in one of the exit locations L_{ex}, and the set of *infinite* sequences of program operations that, by definition, never terminate in one of the exit locations.

The operators $\{\rtimes_\#, \circ_\#, \Cup_\#, \cup_\#\}$ that we introduce in the following have the subscript # to signals that they provide functionality for manipulating, constructing, or weaving Yarn. We omit the subscript if the meaning is clear from the context.

Concatenation. A central operation to deal with words is their concatenation. We also provide this operation for pairs of Yarn, which is their sequential composition, and with it, the concatenation of the words of their languages.

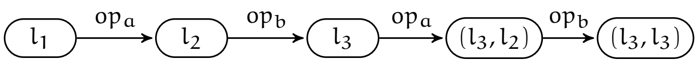
Definition 63: Yarn Concatenation $\circ_\#$

The concatenation of two Yarn $\mathfrak{y}_1$ and $\mathfrak{y}_2$ is a binary associative operator $\circ_\# : \mathfrak{Y} \times \mathfrak{Y} \to \mathfrak{Y}$. The result of concatenating the two Yarn is a new Yarn $\mathfrak{y}_\circ = \mathfrak{y}_1 \circ \mathfrak{y}_2$ with the language

$$\mathcal{L}(\mathfrak{y}_\circ) = \mathcal{L}(\mathfrak{y}_1 \circ \mathfrak{y}_2) = \bigcup \forall_{\bar{o}_1 \in \mathcal{L}(\mathfrak{y}_1)} \forall_{\bar{o}_2 \in \mathcal{L}(\mathfrak{y}_2)} \{\, \bar{o}_1 \circ \bar{o}_2 \,\}.$$

Please note that the set of exit locations L_{ex} of a Yarn must never be empty. Nevertheless, the set might contain bottom locations to signal that the program—or at least the state space exploration—terminates at that point. Control transitions that follow are ignored by the analysis.

Example 11. Let $\mathfrak{y}_1$ be the Yarn that was defined for Example 9 on page 78. The concatenation of $\mathfrak{y}_1 \circ_\# \mathfrak{y}_1$ results in a new Yarn $\mathfrak{y}_\times$ with the following transition relation:



$$\begin{tikzcd} l_1 \arrow[r, "op_a"] & l_2 \arrow[r, "op_b"] & l_3 \arrow[r, "op_a"] & (l_3, l_2) \arrow[r, "op_b"] & (l_3, l_3) \end{tikzcd}$$

***Epsilon* Yarn.** Concatenation is the binary and associative operator of the free monoid of the set of words over a particular alphabet. The third component of a free monoid is its identity element, also known as the neutral element. The *epsilon* YARN takes this role, and is the neutral element regarding concatenation:

Definition 64: Epsilon YARN $\mathfrak{y}_\emptyset$

The *epsilon* YARN $\mathfrak{y}_\emptyset = (h_\perp, \{(l_\epsilon, \mathrm{nop}, l'_\epsilon)\}, \{l_\epsilon\}, \{l'_\epsilon\}) \in \mathfrak{Y}$ consists of one control transition only: a transition that is labeled with nop, which means that the transition does not affect the semantics of the control flow—it is dual to an ϵ-move in an ϵ-NFA. All control locations that are connected by such nop-transitions are considered to be in the same equivalence class, that is, active at the same time: *Temporal properties are not affected* by the introduction of nop-transitions. The YARN $\mathfrak{y}_\emptyset$ is assigned to the $h_\perp$ program concern, that is, it has the *highest priority* in the weaving process.

Language Union. A YARN that has the union of the languages of two given YARN as its language becomes constructed by the YARN union operator:

Definition 65: YARN Union $\cup_\#$

The *union* of two YARN $\mathfrak{y}_1$ and $\mathfrak{y}_2$ is denoted by the function $\cup_\# : \mathfrak{Y} \times \mathfrak{Y} \to \mathfrak{Y}$. We define the operator as

$$\cup_\#(\mathfrak{y}_1, \mathfrak{y}_2) = \pitchfork_\#(\mathfrak{y}_1) \Cup_\# \pitchfork_\#(\mathfrak{y}_2).$$

The join $\mathfrak{y}_\cup = \mathfrak{y}_1 \cup \mathfrak{y}_2$ of two YARN $\mathfrak{y}_1$ and $\mathfrak{y}_2$ results in the union of their respective languages, that is,

$$\mathcal{L}(\mathfrak{y}_\cup) = \mathcal{L}(\mathfrak{y}_1 \cup_\# \mathfrak{y}_2) = \mathcal{L}(\mathfrak{y}_1) \cup \mathcal{L}(\mathfrak{y}_2).$$

The operator extends to sets of YARN naturally.

4.2.8 YARN Domain

A YARN is an abstract word that represents a set of sequences of program operations—its language. We now introduce a YARN domain to describe the relationship between different YARNS; it is an abstract word domain with YARNS as its abstract elements to provide a mapping between these abstract elements and sets of concrete words. See Sect. 3.1 for the general discussion on abstract words and abstract word domains.

One YARN $\mathfrak{y}$, which is an abstract word, denotes $[\![\mathfrak{y}]\!] \subseteq Op^\infty$ a prefix-closed set of sequences of program operations. The YARN domain describes the mapping between YARNS and the denoted sets of sequences of program operations, and describes the relationship between different YARNS:

Definition 66: YARN Domain

The YARN *domain* is an abstract word domain that describes the relationship between different YARNS and provides means to map between YARNS and sets of sequences of program operations. This abstract domain is defined by the tuple $D_{\mathfrak{Y}} = (\mathsf{pr}(Op^\infty), \dddot{\mathfrak{Y}}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$. The concrete words to map to are described by the prefix lattice $\mathsf{pr}(Op^\infty)$, which defines the relationship between different sequences of program operations and is used to form prefix-closed sets of those based on a powerset lattice. The semantic denotation function $[\![\cdot]\!] : \mathfrak{Y} \to 2^{Op^\infty}$ maps a given YARN $\mathfrak{y}$ to the denoted set of sequences of program operations. The abstraction function $\langle\!\langle\cdot\rangle\!\rangle : 2^{Op^\infty} \to \mathfrak{Y}$ creates YARN from a given set of sequences of program operations. The YARN *lattice* $\dddot{\mathfrak{Y}} = (\mathfrak{Y}, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$ describes the relationship between different YARNS. A pair of YARNS is in the inclusion relation $(\mathfrak{y}_1, \mathfrak{y}_2) \in \sqsubseteq$ if and only if $[\![\mathfrak{y}_1]\!] \subseteq [\![\mathfrak{y}_2]\!]$. See Sect. 3.1.2 for the general discussion of abstract word domains. We assume that information about the program concern is encoded along with each sequence of program operations.

4.3 LOOM

This section introduces the LOOM, an analysis to compose the control transition relation of the system to analyze. Any analysis component that is executed along with the LOOM can provide YARN, that is, control transitions to compose by weaving. The LOOM can be used to either compose the control transition relation of the analysis task *before* the actual program analysis algorithm is executed, or to compose or alter the control flow *on-the-fly*, that is, during the analysis or verification process.

Definition 67: LOOM

The LOOM is a *program analysis* for *composing* the control transition relation of an analysis task by weaving, from different sources that provide YARN to weave for different concerns.

128 Labeling of Transitions

The LOOM analysis provides the *labels* for transitions between states, that is, it is the analysis component for constructing a *labeled* transition system [215], or *labeled* Kripke structure [60]. Given an abstract state, the LOOM determines the list of control transitions to encode next to end up in a set of successor states. In the case of model checking, where the goal is to give (proof the satisfaction of) certain guarantees, the weaving process has to ensure to not unintentionally alter the semantics of the system under analysis. An analysis of the YARN to weave can be used [158] to guarantee that desired temporal properties are preserved, for example, if the YARN is *spectative* [158], which is typically the case for temporal specifications to weave.

The LOOM analysis composes YARN to weave on top of an *initial control flow relation*—for example, the control-flow automaton that represents the program that was initially provided to the verification tool as an argument. This weaving process results in the *control flow relation of the analysis task*. Transitions of the initial control flow relation have priority only if no analysis component would have provided YARN to weave. Transitions of the initial control flow have the lowest priority in the weaving process, that is, the top concern $h_\top \in H$ is mapped to them—see Sect. 4.2.1 for a discussion of the priority of concerns in the weaving process.

129

Initial Transition Relation

4.3.1 Composition Periods

The LOOM analysis can compose a syntactic task model—of an analysis task, which is a program—in different periods of an analysis workflow. We distinguish between three distinct periods:

Definition 68: Offline Composition

The traditional approach for automatically composing a program from different, more or less structured, program fragments is *offline composition*: Additional functionality is instrumented into a base program in a *separate process*—the result is a fully composed program that can then be handed over to another process, for example, an analysis tool or an execution environment, as an argument.

Definition 69: Upstream Composition

We say that a composition approach operates in the *upstream* if (1) the composition is finished *before* the actual interpretation of the composition result is conducted, and (2) it takes place in the *same instance* of an analysis tool, interpreter, or virtual machine.

Definition 70: On-The-Fly Composition

A composition approach operates *on-the-fly* if the composition is interleaved with the interpretation of the composition result: The interpretation of the composition result can influence the ongoing composition process and vice versa.

If conducted on-the-fly, YARN to weave is introduced into the control flow of the system to analyze during the analysis step, that is, while computing the abstract successor states in the transfer relation of the analysis. This approach (1) provides more flexibility in choosing from an encoding of the YARN to weave that fits the needs of the current situation best. Furthermore, (2) it can help to reduce the number of transitions to weave by taking the current context of the analysis into account, it (3) helps to avoid costs for static weaving during the initialization step of the analysis tool, and (4) it enables new analysis approaches, for example, in the context of multi-property verification, model-driven test generation, program synthesis [186], and precision synthesis. In our work that presented [8] the initial version of the Loom analysis, we utilize the concept of dynamic precision adjustment and lazy abstraction to enable or disable weaving particular sequences of program operations for specific parts of the abstract state space dynamically. If applied for scenarios with a lot to weave—for example, in model-driven test generation where we have to weave potentially thousands of test goals, or in multi-property verification [8]—weaving everything into the program code introduces a lot of noise in the program, especially in cases where only a subset of the woven transitions is relevant for the reasoning task at hand. This adds another burden to the precision refinement procedure, makes calls to the SMT solver more expensive, and makes the program-comprehension task for users the tool harder [8].

Different program analysis techniques are applicable independent of the chosen point of time of composition by weaving: Irrelevant transitions can always be ignored later by slicing [150] or abstraction techniques [15, 35, 67]. Such techniques can, for example, be triggered in the precision refinement phase of a model checker. A domain-type [7] analysis can be conducted on woven transitions to make the subsequent analysis step more efficient.

4.3.2 Delayed Weaving of Yarn

Several analysis components are executed along with the Loom, and each of them can provide Yarn to weave. The Loom *guarantees* to consume and weave every Yarn that is provided by any of these analysis components. Each analysis that provides Yarn to weave can rely on this guarantee and, for example, pass on information based on the Yarn to other analysis steps or components to keep the overall analysis sound. The Loom might not necessarily weave a given Yarn already when determining the next abstract successor state because other analysis components might have provided Yarn to weave as well. The Loom analysis maintains a priority queue of Yarn to compose by weaving in each of its abstract states, for each control location; priorities are defined based on the concern dependency graph. The following example illustrates how dependencies influence the weaving process:

Example 12. In this scenario, dependencies between concerns have to be taken into account to decide which control transition to weave next to the transition relation of the analysis task. The component analyses provide Yarn to weave as part of an abstract state e_1:

$$\begin{tikzcd}\cdots \arrow[r] & \boxed{e_1 = (l_i, \dots, \mathfrak{y}_1, \dots, \mathfrak{y}_2, \dots)}\end{tikzcd}$$

The Yarn $\mathfrak{y}_1 = (h_1, \{(l_s, a, l_t)\}, \dots)$ is emitted for concern h_1 and Yarn $\mathfrak{y}_2 = (h_2, \{(l_k, b, l_m)\}, \dots)$ for the concern h_2. The Loom has to decided which Yarn to weave first on location $l_1 \in L$; it does so by taking a concern dependency graph $\mathbb{H}$ with the concern dependency relation $\eta = \{(h_1, h_\top), (h_2, h_\top), (h_2, h_1), (h_\bot, h_1), (h_\bot, h_2)\}$ into account—the left-hand side in the binary relation has higher priority in the weaving process. That is, Yarn $\mathfrak{y}_2$ becomes woven prior to the Yarn $\mathfrak{y}_1$ since $(h_2, h_1) \in \eta$:

$$\begin{tikzcd}\cdots \arrow[r] & \boxed{e_1 = (l_i, \dots, \mathfrak{y}_1, \dots, \mathfrak{y}_2, \dots)} \arrow[r, "b"] & \boxed{((l_i, l_k), \dots)} \arrow[r, "a"] & \boxed{((l_i, l_s), \dots)} \arrow[r] & \cdots\end{tikzcd}$$

4.3.3 Lookahead

Certain program analyses—for example, the YARN transducer analysis that we present in Sect. 4.4.4—perform a lookahead in the control transition relation to conduct their analysis. The *depth* $\ell \in \mathbb{N}_0$ of a lookahead determines how many transitions (labeled with program operations) ahead of the current position in the transition relation an analysis component can take into account to decide which action to perform. For example, to emit YARN that checks if the arguments that are passed to a function satisfy certain requirements, or to replace certain program operations—which is not addressed in this work further. The possibility of conducting a lookahead allows us to introduce YARN *before* or *between* a particular sequence of program operations.

A lookahead on the YARN that should be woven next is only possible if it is available for a lookahead before it becomes actually introduced into the transition relation of the analysis task. This is possible by taking advantage of the LOOM's possibility to *delay* the weaving process to build up a *queue* of control transitions to weave and perform the lookahead on them. In the case that already the initial control flow relation represents the fully composed analysis task, lookaheads of arbitrary length are possible without relying on further techniques. In the case transitions from additive YARN can be composed into the transition relation, we need mechanisms to also include those transitions for lookaheads. We limit our discussions of the LOOM to a lookahead of depth $\ell = 1$ and additive YARN only.

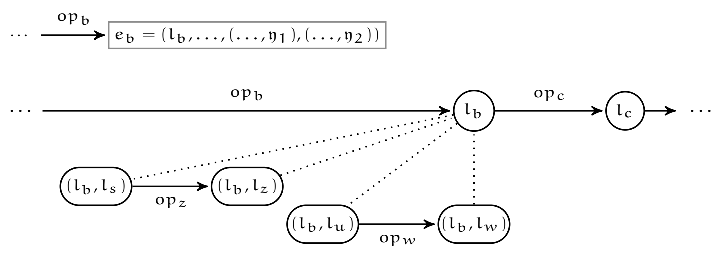
To delay the weaving of YARN, we can introduce a neutral ϵ-transition before and immediately after each abstract state (of the LOOM analysis) for that YARN was emitted to weave. That is, given a control location $l \in L$, proceeding along the transitions $\{g \mid (l, \cdot, l') \in G\}$ that leave location l is delayed until no more additional YARN becomes emitted for composition on control location l. This approach is sufficient for our needs, that is, for a lookahead of depth $\ell = 1$.

4.3.4 LOOM Analysis

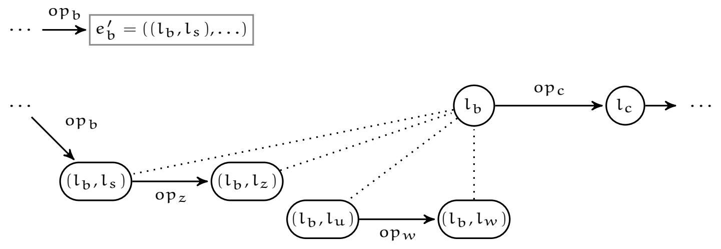
After we have discussed the general functionality of the LOOM in the preceding sections, we now describe our LOOM analysis in full detail. We formalize the LOOM analysis as a Configurable Program Analysis [31, 32] that takes the YARNS to weave from other analyses that run in parallel in terms of component analyses—see Sect. 2.5.3 for details on composite analyses.

130
Different Realizations

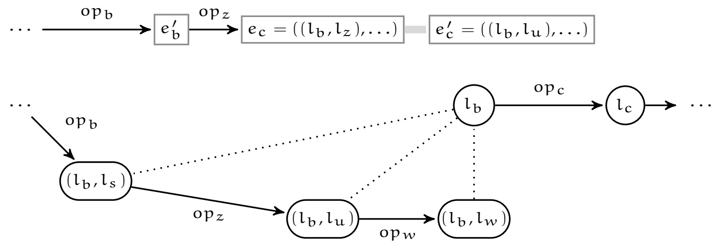
In this work—compared to the paper [8] in that we presented a first prototype of a LOOM analysis—the analysis *combines* tracking the current control location in the transition relation of the analysis task with the weaving functionality. Keeping track of the current control location is traditionally the functionality of a separate Location CPA [31]. This design choice was made because of the *strong coupling* of these two concerns within a tool for program analysis. The LOOM analysis conducts its weaving by redirecting the analysis flow to other, or newly introduced, control locations—see Def. 124. An example that illustrates the weaving process, as conducted by the LOOM analysis, is provided in Fig. 18.



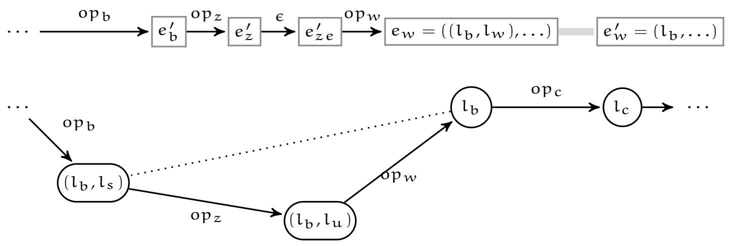
(a) Abstract state e_b is a successor state for the control transition $\xrightarrow{op_b} l_b$. Yarn was emitted for composition by weaving on location l_b by two analysis components: Yarn $\eta_1 = (h_1, \{(l_s, op_z, l_z)\}, \ldots)$ and $\eta_2 = (h_2, \{(l_u, op_w, l_w)\}, \ldots)$. The Loom's transfer relation introduces composed transitions into the transition relation G of the analysis task: $G' = G \cup \{((l_b, l_s), op_z, (l_b, l_z)), ((l_b, l_u), op_w, (l_b, l_w))\}$.



(b) Since there were two Yarns emitted for weaving on location l_b, the Loom has to take the concern dependency graph into account to choose which one to weave first, and decides to continue on the compound control location (l_b, l_s). The Loom analysis redirects the control flow to location (l_b, l_s) by strengthening state e_b, ending up in state e'_b.



(c) When reaching control location (l_b, l_z), no successor locations are specified in the transition relation. The Loom analysis comes into play: There are still transitions to add before location l_b. Abstract state e_c gets strengthened, which results in state e'_c, such that the analysis continues on the compound control location (l_b, l_u).



(d) No control transition immediately follows (l_b, l_w) and no further transitions where provided to compose on l_b. The analysis thus decides to strengthen e_w, ending up in state e'_w, with a redirect to l_b.

Figure 18: An example to illustrate composition by weaving with the Loom.

Loom *CPA* $\mathbb{L}^\#$. We implement the Loom as a Configurable Program Analysis (CPA) [32] that wraps other CPAs—similar to a composite CPA [31]. The Loom CPA $\mathbb{L}^\# = (\mathbb{W}, D_{\mathbb{L}^\#}, \rightsquigarrow_{\mathbb{L}^\#}, \mathsf{merge}_{\mathbb{L}^\#}, \mathsf{stop}_{\mathbb{L}^\#}, \mathsf{prec}_{\mathbb{L}^\#}, \mathsf{target}_{\mathbb{L}^\#})$ is defined by the following components and operators:

Algorithm 4 CollectCompound(M_{in}, G_{in})

```
Input: A set of LOOM states $M_{in} \in M$,
       and a transition relation $G_{in} \subseteq G$.
Output: A pair $(M_{out}, G_{out})$
 1: $M_{out} = \emptyset$
 2: $G_{out} = G_{in}$
 3: for $m = (l, l_o, \xi, w) \in M_{in}$ do
 4:   // Extract all YARN from $w$ and create a compound version of it.
 5:   $Y_\times = \{ \mathfrak{y} \rtimes l \mid \mathfrak{y} \in \text{collect}(w) \}$
 6:   // In case a lookahead is needed, add additional YARN that
 7:   //    inserts an $\epsilon$-transition before new transitions.
 8:   if $Y_\times \neq \emptyset$ then
 9:     $Y_\times = Y_\times \cup \{\mathfrak{y}_\emptyset \rtimes l\}$
10:   // Enqueue the compound YARN for weaving on location $l$.
11:   $\xi' = \xi \cup \{ (l, l_{en}) \mid l_{en} \in \mathfrak{y}.L_{en} \wedge \mathfrak{y} \in Y_\times \}$
12:   // Add the compound YARN to the resulting transition relation.
13:   $G_{out} = G_{out} \cup \bigcup_{(h, G_\mathfrak{y}, \ldots) \in Y_\times} G_\mathfrak{y}$
14:   // Add a modified version of the loom state to result.
15:   $M_{out} = M_{out} \cup \{ (l, l_o, \xi', w) \}$
16: return $(M_{out}, G_{out})$
```

Wrapped CPA $\mathbb{W}$. The first component of our LOOM CPA is the Configurable Program Analysis $\mathbb{W}$ that is *wrapped* by the LOOM CPA. This wrapped analysis can be composed from yet other analyses. The LOOM CPA delegates calls to this CPA if appropriate.

Abstract Domain $D_{\mathbb{L}^\#}$. The abstract domain $D_{\mathbb{L}^\#} = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ defines how information that is relevant for traversing and composing a control transition relation is encoded. The semi-lattice $\mathcal{E}$ defines the relationship between the abstract LOOM states $M = (L \times L \times 2^{L \times L}) \times W$. One *abstract* LOOM *state* $m = (l_c, l_o, \xi, w) \in M$ consists of the *current control location* l_c, a *redirect origin location* l_o from that the LOOM has redirected the analysis (l_c is equal to l_o if there was no redirect), a *remaining redirects relation* ξ, and a *wrapped abstract state* w that is element of the lattice $\mathcal{W}$ of the wrapped CPA $\mathbb{W}$—typically a composite CPA [31]. Abstract LOOM states always refer to control locations and control transitions of the transition relation that reflects the composed analysis task. The semi-lattice $\mathcal{E} = (M, \sqsubseteq, \sqcup, \bot)$ is a tuple that consists of the set of abstract LOOM states M, an inclusion relation $\sqsubseteq$, a join operator $\sqcup$, and a bottom element $\bot$. The inclusion relation $\sqsubseteq \subseteq M \times M$ defines which abstract LOOM state overapproximates (includes) which other LOOM state. Two LOOM states $m_1 = (l_1, l_{1o}, \xi_1, w_1)$ and $m_2 = (l_2, l_{2o}, \xi_2, w_2)$ are in the inclusion relation $(m_1, m_2) \in \sqsubseteq$ if and only if $l_1 = l_2$ and $\xi_1 = \xi_2$ and $(w_1, w_2) \in \sqsubseteq_\mathcal{W}$. The join $\sqcup : M \times M \to M$ defines the least upper bound of two LOOM states; its definition follows directly from the inclusion relation. The denotation function $[\![\cdot]\!] : M \times 2^C$ provides the set of concrete states that are represented by an abstract LOOM state. The abstraction function $\langle\!\langle\cdot\rangle\!\rangle : 2^C \to M$ transforms a set of concrete states to an abstract LOOM state.

OPERATOR $\rightsquigarrow_{\mathbb{L}^\#}$. The transfer relation $\rightsquigarrow_{\mathbb{L}^\#}$ of the LOOM analysis is the core component for the process of composing the transition relation of the analysis task by weaving. It invokes the transfer relation of the wrapped CPA—possibly a Composite CPA. The transfer operation is divided in three phases:

1. *Transfer.* Given a LOOM state $m = (l, l_o, \xi, w)$, the first phase determines all transitions that leave control location l and computes a list of abstract successor states $\widehat{m}' \subseteq M$ for them. The set of control transitions that leave location l is defined by $\mathsf{outFrom}(l) = \{g \mid g = (l, op, l') \in G_t\}$. The transition control relation G_t describes the control flow of the analysis task that was composed up to this point in time. 131 *Function* outFrom

   We call the transfer relation of the wrapped analysis $\mathbb{W}$ to compute successors for the wrapped abstract state w. We end up in the list of successor LOOM states $\widehat{m}' = \langle\, (l', l', \xi, w') \mid g = (l, \cdot, l') \in \mathsf{outFrom}(l) \wedge w \overset{g}{\rightsquigarrow}_{\mathbb{W}} \widehat{w}' \wedge w' \in \widehat{w}' \,\rangle$. The transfer relation returns the empty list $\emptyset$ of successor states if the current control location l is a bottom location, that is, $l.\mathit{bottom} = \mathit{true}$. 132 *Bottom Location*

2. *Collect*. This phase creates *compound* YARN from the YARN that has been emitted along with the successor states $\widehat{m}'$ and adds it to the transition relation G_t of the analysis task. It extracts the YARN to weave from all states that are wrapped within any state in $\widehat{m}'$. From the set of abstract LOOM states $\widehat{m}'$ we derive a set $\widehat{m}''$ of abstract LOOM states and update the control transition relation G_t of the analysis task. We use the function $\mathsf{collect} : E \to 2^{\mathfrak{Y}}$ that, given an abstract (composite) state, returns all YARN that can be found within it, that is, that was emitted by one of the component analyses. Algorithm 4 illustrates the process of deriving a new control transition relation and changed set of LOOM states $\widehat{m}''$, that is, $(\widehat{m}'', G_t') = \mathsf{CollectCompound}(\widehat{m}', G_t)$. 133 *Function* collect

   In case a lookahead—see Sec. 4.3.3—is needed, we introduce ϵ-moves whenever additional yarn, for composition by weaving, was provided in this analysis step.

3. *Redirect.* The next step is to determine the control-flow location from that the next control transitions should be taken, which possibly leads to a *redirect* to another control location, and with it, control transitions that have possibly been composed to the transition relation in a previous phase of the operator, or a previous step of the state space exploration algorithm.

   The process of choosing the set of control locations to redirect to is implemented in the algorithm $\mathsf{ChooseRedirectTargets}$ (Alg. 5). We use the function $\mathsf{withPriority} : 2^L \to 2^L$, which is implicitly parameterized with a concern dependency graph $\mathbb{H}$, to determine the set of control locations to proceed the state space traversal on. Given a set of control locations $L_c \subset L$ the function returns a subset of control locations $L_r \subseteq L_c$ with the highest priority in the weaving process regarding the dependency graph $\mathbb{H}$. Given a control location l, the function $\mathsf{compoundOn} : l \to l$ returns the major control location l_p in case location l is a composite location $l = (l_p, l_c)$, and returns the control location l itself otherwise. 134 *Function* withPriority 135 *Function compoundOn*

Algorithm 5 ChooseRedirectTargets(l, ξ)

Input: A control location $l \in L$ to redirect from,
a remaining redirects relation $\xi \subseteq L \times L$

Output: A set of control locations $L_t \in L$ to redirect to.

```
1: // Choose from locations to compose with l based on the
2: //    priority that results from the concern dependency graph ℍ.
3: $L_r = \mathsf{withPriority}_{\mathbb{H}}(\xi(l))$
4: if $L_r \neq \emptyset$ then
5:     // There is no YARN left to compose with location l.
6:     return $L_r$
7: if $\mathsf{compoundOn}(l) = \bot$ then
8:     // The location is not composed with any other locations.
9:     return $\{l\}$
10: if $\mathsf{outFrom}(l) = \emptyset$ then
11:     // The location is composed with another location:
12:     //    Check for a redirect from there.
13:     return $\mathsf{ChooseRedirectTargets}(\mathsf{compoundOn}(l), \xi)$
14: return $\emptyset$
```

Given the set of LOOM states $\widehat{m}''$ from the previous phase, we compute a new set $\widehat{m}'''$, for that the current control location of some of the states might have been changed. Along with a change of the current control locations l_c, also the redirect origin location l_o, and the remaining redirects relation ξ might have been changed:

$$\begin{aligned}\widehat{m}''' = \{\, (l', l_c, \xi', w) \mid (l_c, l_o, \xi, w) \in \widehat{m}'' \\ \wedge L_R = \mathsf{ChooseRedirectTargets}(l_c, \xi) \\ \wedge l' \in L_R \\ \wedge \xi' = \xi \setminus \{l_c, l'\} \,\}\end{aligned}$$

Please note, that the last two phases correspond the strengthening operator $\downarrow_{\mathbb{L}^\#}$ that would be used if the LOOM is implemented as a component CPA, which is wrapped by a composite CPA—as this was the case in our paper [8] that presented a first version of the LOOM analysis.

OPERATOR $\mathsf{target}_{\mathbb{L}^\#}$. Given a LOOM state $m = (l, l_o, \xi, w) \in M$, with a wrapped state w and the current control location l. The operator $\mathsf{target}_{\mathbb{L}^\#}$ returns *true* if and only if the current control location has assigned a non-empty set of possibly violated properties, or if its wrapped state is a target state, that is:

$$\mathsf{target}_{\mathbb{L}^\#}((l, w)) = l.\mathit{violated} \neq \emptyset \vee \mathsf{target}_{\mathbb{W}}(w) = \mathit{true}$$

OPERATOR $\mathsf{merge}_{\mathbb{L}^\#}$. The merge operator $\mathsf{merge}_{\mathbb{L}^\#}$ determines to which extent information of two LOOM states should get combined. Given a call $\mathsf{merge}(e, r)$, the operator returns r if $e.l = r.l$, otherwise it delegates the merge of the wrapped states e_w and r_w to the merge operator $\mathsf{merge}_{\mathbb{W}}$ of the wrapped CPA $\mathbb{W}$, resulting in state e'_w; the operator $\mathsf{merge}_{\mathbb{L}^\#}$ returns state r if $e'_w = r_w$, otherwise it returns $(e.l, e.l_o, e.\xi, e'_w)$.

Operator $\mathsf{stop}_{\mathbb{L}^\#}$. The coverage check $\mathsf{stop}_{\mathbb{L}^\#} : M \times 2^M \to \mathbb{B}$ determines if a given Loom state is covered fully by a state that has already been reached. That is, a call $\mathsf{stop}_{\mathbb{L}^\#}(m, R)$ returns *true* if and only if there exists a state $r \in R$ with $r.l = m.l$ and $\mathsf{stop}_{\mathbb{W}}(w, \{r.w \mid r \in R\}) = \textit{true}$.

Operator $\mathsf{prec}_{\mathbb{L}^\#}$. The precision adjustment operator $\mathsf{prec}_{\mathbb{L}^\#}$ just delegates the precision adjustment of the wrapped state w to the precision adjustment operator $\mathsf{prec}_{\mathbb{W}}$ of the wrapped CPA $\mathbb{W}$.

4.4 YARN TRANSDUCER

The Loom analysis *composes* the control-flow relation of the analysis task *by weaving* Yarn that is provided by other analyses that run in parallel to—or as component analyses of—the Loom analysis. In this section, we present Yarn transducers and the corresponding Yarn transducer analysis as one possible *mechanism for providing* Yarn *to weave*; the latter executes Yarn transducers during an analysis run.

Definition 71: Yarn Transducer

A Yarn *transducer* (or $\mathfrak{y}$*-transducer*) is a transducer that emits Yarn to weave based on the sequence of program operations that it has observed so far—and possibly a lookahead of finite length on program operations that follow. We instantiate an abstract transducer to work as a Yarn transducer, which results in the tuple

$$Y = (Q, D_{\mathrm{in}}, D_{\mathrm{out}}, \iota_0, F, \delta).$$

The set of control states Q, the set of initial transducer states ι_0, the set of final control states F, and the transition relation δ have the meaning that was defined for generic abstract transducers—see Sect. 3.2. The abstract input domain and the abstract output domain define the characteristics that are specific for Yarn transducers:

- Abstract Input Domain D_{in}. The *abstract input domain* is an abstract word domain $D_{\mathrm{in}} = (\mathsf{fl}(Op^*), \ddot{\mathfrak{I}}, [\![\cdot]\!]_{\mathrm{in}}, \langle\!\langle\cdot\rangle\!\rangle_{\mathrm{in}})$, with the lattice $\ddot{\mathfrak{I}}$ of abstract input words. One *abstract input word* denotes a set of finite sequences of program operations, which is reflected by the semantic denotation function $[\![\cdot]\!]_{\mathrm{in}} : \mathfrak{I} \to 2^{Op^*}$. The abstraction function $\langle\!\langle\cdot\rangle\!\rangle_{\mathrm{in}} : 2^{Op^*} \to \mathfrak{I}$ provides a mapping from sets of program operation sequences to abstract input words.

- Abstract Output Domain $D_{\mathfrak{Y}}$. The *abstract output domain* of a Yarn transducer is the Yarn domain $D_{\mathrm{out}} = (\mathsf{pr}(Op^\infty), \ddot{\mathfrak{Y}}, [\![\cdot]\!]_{\mathrm{out}}, \langle\!\langle\cdot\rangle\!\rangle_{\mathrm{out}})$. Each transition of a Yarn transducer is annotated with a Yarn $\mathfrak{y} \in \mathfrak{Y}$ as its output symbol. See Sect. 4.2.8 for the full definition of this abstract word domain.

The infinite *set of all* Yarn *transducers* is denoted by $\mathbb{Y}$.

The Loom analysis can *compose a list of* Yarn *transducers* by weaving their output; the result can again be expressed as a Yarn transducer—that is, a Yarn transducer can be produced by composing it from Yarn transducers. Please note that we only discuss additive Yarn in this work. In model checking, two categories of Yarn are of high relevance: (1) Yarn that represents the formal specification, and (2) Yarn that represents the environment model.

4.4.1 Lifting to Yarn Transducers

Abstract transducers are a rather abstract concept. Specifying a Yarn transducer as intended by the formalism can be cumbersome: A practitioner has to define a Yarn as the abstract output symbol for each transition—that is, provide a program concern and a set of control transitions.

To reduce the effort for specifying Yarn transducers, we provide means to create them by lifting them from less expressive representations, for example, from finite-state transducers with both a concrete input alphabet and a concrete output alphabet:

Definition 72: Transducer Lift

We *lift* a finite-state transducer $A = (Q, \Sigma^*, \Theta^*, q_0, F, \delta)$ that describes a fragment of a program concern h, with a set of control states Q, a concrete input alphabet Σ^*, a concrete output alphabet Θ^*, an initial control state q_0, and set of final control states F to a Yarn transducer. The resulting Yarn transducer $Y = (Q, D_{\text{in}}, D_{\text{out}}, \iota_0, F, \delta')$ has the abstract input domain $D_{\text{in}} = (\mathsf{fl}(Op^*), \ddot{\mathfrak{I}}, [\![\cdot]\!]_{\text{in}}, \langle\!\langle\cdot\rangle\!\rangle_{\text{in}})$, the abstract output domain $D_{\text{out}} = (\mathsf{pr}(Op^\infty), \ddot{\mathfrak{Y}}, [\![\cdot]\!]_{\text{out}}, \langle\!\langle\cdot\rangle\!\rangle_{\text{out}})$, and the initial set of transducer states $\iota_0 = \{(q_0, \mathfrak{v}_\epsilon)\}$. At the heart of this lifting result is the new transition relation

$$\begin{aligned}\delta' = \{\ (q, \mathfrak{v}, q', \mathfrak{w}) \mid \tau = (q, \overline{o}_1, q', \overline{o}_2) \in \delta \\ \wedge\ \mathfrak{v} = \langle\!\langle \{\overline{o}_1\} \rangle\!\rangle_{\text{in}} \\ \wedge\ \mathfrak{w} = \mathsf{derive}_{\mathfrak{Y}}(h, \tau)\ \}.\end{aligned}$$

The operator $\mathsf{derive}_{\mathfrak{Y}} : H \times (L \times Op^* \times L) \to \mathfrak{Y}$ creates a Yarn for a given concern h from a transition $\tau \in \delta$. Its result is defined as follows:

$$\begin{aligned}\mathsf{derive}_{\mathfrak{Y}}(h, (q, \overline{o}_1, q', \overline{o}_2)) = (h, \{(l, op, l')\}, \{l\}, \{l'\}), \\ \textit{if } \overline{o}_1 = \epsilon \textit{ then } l = L(q) \textit{ and } l' = L(q') \\ \textit{if } |\overline{o}_1| = 1 \textit{ then } l = l^{\#} \textit{ and } l' = L(q') \\ \textit{if } \overline{o}_2 = \epsilon \textit{ then } op = nop \\ \textit{if } \overline{o}_2 = \langle op_1 \rangle \textit{ then } op = op_1\end{aligned}$$

The fresh labeling for the first control location of a non-ϵ-move ensures that its second location is only reached in the transition matched.

We assume that there is also a lifting operation that lifts from a finite-state transducer that emits one concrete output symbol on each of its transitions to a finite-state transducer that emits a sequence of output symbols (word) on each transition. Lifting to Yarn transducers is also possible from control-flow graphs or control flow automata—all transitions of such Yarn transducers are then ϵ–moves. If not stated otherwise, we assume that each abstract transducer emits Yarn for one concern only. This helps to reduce the complexity of the presentation.

4.4.2 Parameterized Yarn Transducers

A Yarn transducer can have parameterized Yarns on its transitions as abstract output words. A parameterized Yarn must be instantiated before the Loom analysis can weave it to the transition relation of the analysis task. We *instantiate* parameterized Yarn and corresponding parameterized program operations by binding expressions—in the programming language of the program—to the parameters. Section 3.1.4 provides a general discussion on parameterized abstract words and their instantiation. Figure 6a on page 30 illustrates an example of a Yarn transducer that emits parameterized words.

4.4.3 Classification of Yarn

Depending on the application of the Loom analysis, or more precisely, the type of (temporal) property to analyze, the weaving process must satisfy specific requirements such that neither completeness nor preserve temporal properties. These ideas are related to the correctness by construction paradigm [181], which aims at composing components that do not affect, or have a known impact on, the correctness of the resulting system.

Environment Yarn. A (sufficient) precise environment model for a given software system is essential to make software model checking practically applicable. It is crucial for reducing the number of false alarms [199] that would otherwise—if the environment would not be in place—arise from program states that are not feasible in the real system in its real (application) environment. The environment model is regulative [158]: It aims at reducing the number of false alarms by restricting the observable behavior of the system by strengthening assumptions. A Yarn is *regulative* if "the projection of the augmented state graph on the variables of the underlying system is identical to the state graph of the underlying system, except that some states are repeated (with new edges from aspect operations) and some edges are removed. States are ignored that become disconnected (unreachable) from the augmented state graph with entrance points (external method calls)." [158].

136 *Regulative*

Specification Yarn. Yarn that represents the formal specification must be spectative. A Yarn is *spectative* if "the projection of the augmented state graph onto the state variables of the underlying system is identical to the underlying state graph, except that the projection contains additional repetitions of states connected by edges that correspond to aspect operations." [158] That is, specification Yarn must observe the behavior but not modify or restrict it. The operations that are introduced by the weaving process must never modify or restrict the state space of the program under analysis: (1) assignment operations are only allowed to assign values to variables that have been introduced by the Yarn itself, and (2) for each control location, the disjunction of all predicates from assume operations on the outgoing transitions must evaluate to *true*—an empty list of assumes evaluates to *true*. An analysis shall not stop exploring a program path after a property violation; other properties could be violated later along the path as well (completeness). The Yarn has to introduce a split (branching) of the state space for this purpose.

137 *Spectative*

4.4.4 Yarn Transducer Analysis

A Yarn transducer analysis—or Yarn *analysis*, for short—executes a given Yarn transducer as one of several analysis components of the verification engine. The Yarn analysis emits Yarn to weave and is used in combination with a Loom analysis, which consumes the emitted Yarn and takes care of weaving it into the transition relation of the analysis task—see Sect. 4.3 on details of the weaving process.

We instantiate this analysis based on the abstract transducer CPA—see Sect. 3.2—and redefine some of its operators. One instance of a Yarn analysis represents one Yarn transducer $Y = (Q, D_{in}, D_{out}, \iota_0, F, \delta)$. The analysis (1) keeps track of the current state of the abstract transducer (including the emitted output) and determines its successors based the transition relation δ and the transitions of the analysis task, (2) it provides Yarn to weave into the transition relation of the analysis task, and (3) it can signal if a target state has been reached. That is, we define the Yarn analysis by the tuple

$$\mathbb{D}_Y = (D_Y, \rightsquigarrow_Y, \downarrow_Y, \mathsf{merge}_Y, \mathsf{stop}_Y, \mathsf{prec}_Y, \mathsf{target}_Y).$$

The components and operators of the analysis are defined as follows:

Abstract Domain D_Y. The abstract domain $D_Y = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ is defined based on a map lattice $\mathcal{E} = (J, \top, \bot, \sqsubseteq, \sqcup, \sqcap)$, with $J = 2^{Q \to \mathfrak{Y}}$, where each element $\iota \in J$ of the lattice is a transducer state. One transducer state $\iota = \{(q, \mathfrak{y}), \dots\} \in J$ is a mapping $\iota : Q \to \mathfrak{Y}$ from control states to Yarns. The Yarn analysis starts with the initial transducer state ι_0 of the Yarn transducer to conduct runs for.

Operator $\rightsquigarrow_Y$. The transfer relation $\overset{g}{\rightsquigarrow}_Y \subseteq J \times G \times J \times \Pi$ defines the set of abstract successor states of an abstract transducer state $\iota = \{(q, \mathfrak{y}), \dots\} \in J$, for a given control-flow transition $g \in G$ (the label on the control transition) and abstraction precision $\pi \in \Pi$. To deal with ϵ-moves, we use the output closure operator that maintains the set of concrete output words, similar to a regular closure operator $\mathsf{abstclosure}_\infty$ as defined in Sect. 3.2.3. The transfer relation $\rightsquigarrow_Y$ is in other respects equivalent to the transfer relation $\rightsquigarrow_T$ of the generic abstract transducer analysis—see Sect. 3.5.1.

Operator prec_Y. We use the precision adjustment operator prec_T of the abstract transducer analysis: $\mathsf{prec}_Y = \mathsf{prec}_T$, that is, we do not abstract here: A call $\mathsf{prec}_Y(\iota, \pi, \cdot)$ returns the pair $(\iota, \pi) \in J \times \Pi$ without adjustments.

Operator merge_Y. The operator $\mathsf{merge}_Y : J \times J \times \Pi \to J$ keeps two abstract states always separate, that is, $\mathsf{merge}_Y = \mathsf{merge}^{\mathsf{sep}}$. An alternative implementation of the operator is presented later in Sect. 4.5.2.

Operator stop_Y. The operator $\mathsf{stop}_Y : J \times 2^J \to \mathbb{B}$ checks whether there is already an abstract state that subsumes a given state. This means that $\mathsf{stop}_Y(\iota, \mathcal{R})$ returns *true* if and only if $\exists \iota' \in \mathcal{R} : \iota \sqsubseteq \iota'$.

Operator target_Y. The operator $\mathsf{target}_Y : J \times \mathbb{B}$ returns *true* if and only if the given abstract state $\iota = \{(q_1, \mathfrak{y}_1), \dots (q_n, \mathfrak{y}_n)\} \in J$ contains a pair $(q_i, \cdot)$, with $q_i \in F$, that is, q_i is a target state of the Yarn transducer.

4.5 DYNAMIC CONTROL ENCODING

A program analysis decides whether or not to merge two abstract states, and this could mean that it loses track of a path if the property-related behavior is different for the branches. An analysis, and its abstract domain, can encode sufficient information to recover the path taken in a later step cheaply, or it might lose this information and report false alarms and no or imprecise counterexamples.

Tracking the current control state of a YARN transducer can lead—as we have discussed in Sec. 4.1.3—to a combinatorial explosion of the abstract state space: A pair of abstract states is kept separated if the transducer is not in the same control state for them. In this section, we describe an *extension of our* YARN *analysis* that (1) solves this problem, and (2) provides a starting point for studying the effectiveness and efficiency of different encodings of the current control state of automata.

138 Yarn for Path Sensitivity

We take advantage of the fact that the abstract domain of the YARN analysis is defined based on a map lattice: One abstract state of the analysis can represent a set of control states (with corresponding outputs)—in the default configuration, we use singleton maps. Merging these abstract states whenever the control flow of the analysis merges leads to a loss of path sensitivity: Different control states can be reached on different branches of the control flow. We cope with this problem by concatenating *additional* YARN and then rely on other analyses that run in parallel to ensure path sensitivity (potentially more) efficiently. This allows us to encode control states symbolically without losing path sensitivity. We say that analyses that have these attributes have *symbolic path sensitivity*, the technique to end up it such is *symbolic control encoding*.

139 Applicability

Our approach is well-suited for scenarios in which the specification to check describes the *protocol* to adhere to for using a set of (API) methods [23, 39, 228], and for expressing *test goals* [29, 33, 141] of a test-generation procedure. A pure symbolic encoding of automata states, by instrumenting the code *offline*, that is, before the verifier starts, has already been studied [22,
140 Novelty
142, 241]. We do the encoding *on-the-fly*, which enables us to change the type of encoding *dynamically* during the state-space exploration—see Sect. 4.3.1 for a more detailed discussion of points in time to encode. We use separate YARN transducers to emit YARN for different program concerns (and do
141 ⚠
not use their union), with a separate YARN transducer analysis. Please note that the applicability of the concept that we present here is not restricted to YARN transducers: In general, it is applicable to encode the control state of an arbitrary automaton if a LOOM analysis is present.

4.5.1 Control Encoding Strategies

In the following section, we present our extended YARN transducer analysis that supports three modes of control encoding. A *control encoding strategy* defines how the current control location or the current control state in a control transition relation is encoded in the state space of an analysis task. We distinguish between *explicit*, *symbolic*, and *hybrid* control encoding—please note that the terms control location and control state are used interchangeably in the following definitions:

Definition 73: Explicit Control Encoding

A control encoding is *explicit* if each abstract state can be mapped to exactly one control state of a given control transition relation.

Definition 74: Symbolic Control Encoding

A control encoding is *symbolic* if an abstract state can be mapped to a set of different control states of a given control transition relation.

Definition 75: Hybrid Control Encoding

We call a control encoding *hybrid* if a particular control state must never be mapped to an abstract state among others control states for some fraction of the state space, while this can be the case for another fraction thereof.

In the following section, we present an extended YARN transducer analysis that can automatically conduct an alternative encoding (explicit, symbolic, or hybrid). Figure 19 illustrates how the YARN transducers look like that are implicitly constructed by these strategies. In the end, the strategies describe a *choice between control-dependent and data-dependent verification problems*, that are solved best either by explicit or symbolic verification techniques.

4.5.2 Extended YARN Transducer Analysis

This section provides *extensions* of the YARN transducer analysis to allow for different control encoding strategies. We describe how an explicit control encoding can be transformed to a symbolic, or hybrid, control encoding by emitting YARN to keep track of the automaton state in a path sensitive manner. The LOOM with its on-the-fly weaving of emitted YARN is crucial for this functionality.

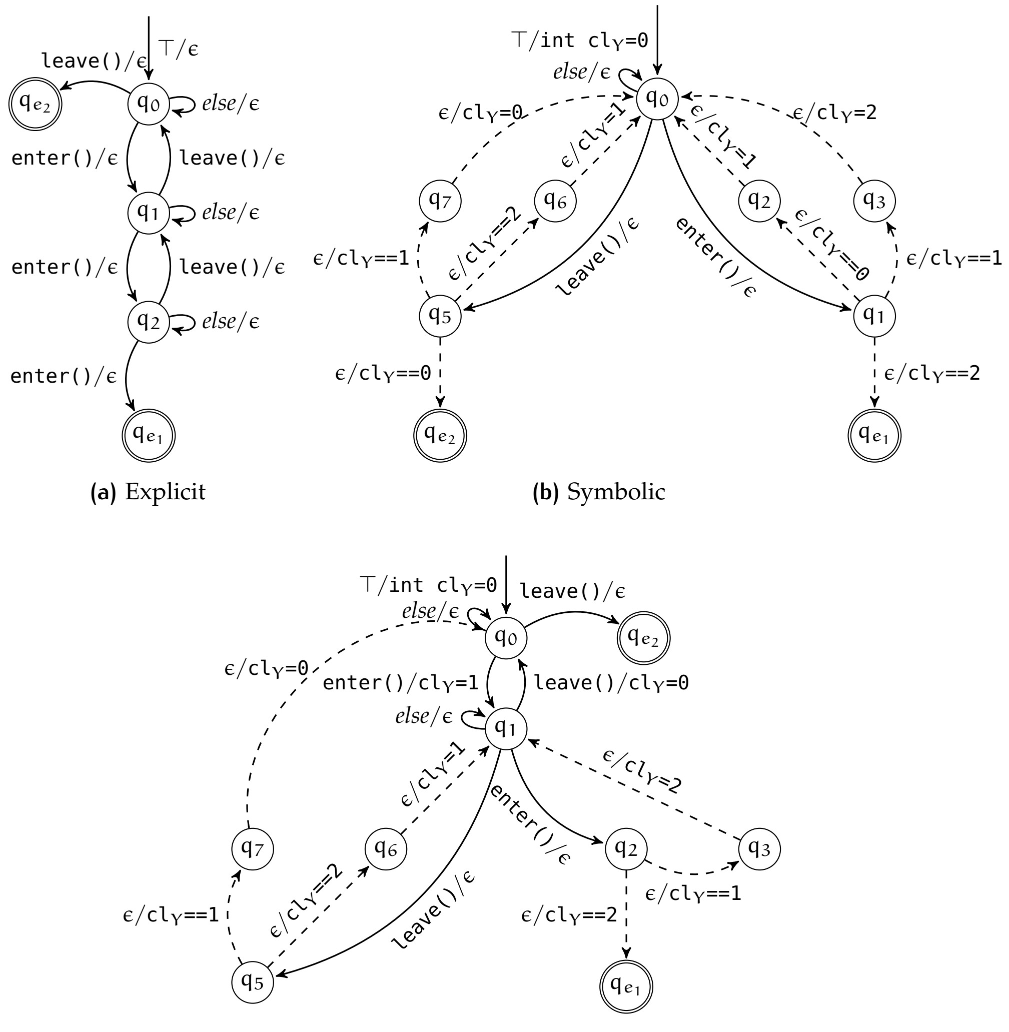
The control encoding strategy of a YARN transducer analysis is configured by a pair $\chi = (\rightsquigarrow, \mathsf{merge})$ of a particular transfer relation and a particular merge operator, which results in the *extended* YARN *transducer analysis* $\mathbb{D}_{Y_\chi} = (D_Y, (\,\cdot\,,\,\cdot\,), \downarrow_Y, \mathsf{stop}_Y, \mathsf{prec}_Y, \mathsf{target}_Y)$—to have effect, we assume that the transducer to run is provided with explicit control encoding.

Explicit. The explicit control encoding is the default configuration of the YARN transducer analysis that we have presented in this work—see Sect. 4.4.4 for details of this analysis. That is, an explicit control encoding is enabled by the pair $\chi_{explicit} = (\rightsquigarrow_Y, \mathsf{merge}^{\mathsf{sep}})$ of operators.

Symbolic. A symbolic control encoding is enabled by the pair $\chi_{symbolic} =$ 142 $(\rightsquigarrow_{Y_S}, \mathsf{merge}_{Y_S})$ of operators. Symbolic control encoding relies on a *state variable* $\mathtt{cl}_Y \in X$ that stores the current control state of the YARN transducer Y. Before the state variable $\mathtt{cl}_Y$ can be used, it has to be declared and initialized. 143 We create an *initialization* YARN η_d, which declares and initializes the state

State Variable $\mathtt{cl}_Y$

State Numbering



(c) Hybrid

Figure 19: Different control encodings of a YARN transducer that represents a specification. Our YARN transducer analysis can derive them automatically from transducers with explicit control encoding. The variable $\mathtt{cl}_Y \in X$ is used to store the *current control state* of the automaton Y (transducer). Dashed - - -➤ edges indicate transitions within the *epsilon closure*. The trigger condition *else* is syntactic sugar and indicates that this transitions matches if none of the other leaving transitions matches. Please note that we have annotated the illustrated transitions with concrete words instead of abstract words for ease of presentation.

variable $\mathtt{cl}_Y$ as `int` $\mathtt{cl}_Y$ `= 0`, where the number $0 \in \mathbb{N}_0$ corresponds to control state q_0—that is, there is a mapping $\mathrm{qn} : Q \to \mathbb{N}_0$, where $\mathrm{qn}(q)$ denotes the number of the control state q. The YARN transducer analysis starts in the new initial transducer state $\iota_0{}' = \{(q, \mathfrak{y}_d \circ \mathfrak{y}) \mid (q, \mathfrak{y}) \in \iota_0\}$, that is, the initialization YARN is prepended to the initial output of the YARN transducer. This is the typical approach to encode automata in programs [22, 241]. In the end, we convert control dependencies into data dependencies. The responsibility for ensuring the path sensitivity is delegated to another analysis that runs in parallel.

Algorithm 6 $\rightsquigarrow_{Y_S}(\iota, g)$

```
Input: A transducer state $\iota \in J$,
       and a control-flow edge $g \in G$.
Output: A set of abstract transducer states $\subseteq J$
1: $S = \{ \{(q, \mathfrak{y}, q', \mathfrak{y}')\} \mid (q, \mathfrak{y}) \in \iota \wedge (q', \mathfrak{y}') \in \overset{g}{\rightsquigarrow}_Y(\{(q, \mathfrak{y})\}) \}$
2: if $\forall (q, \cdot, q', \cdot) \in S : q = q'$ then
3:     return $\{ (q', \mathfrak{y}') \mid (\cdot, \cdot, q', \mathfrak{y}') \in S \}$
4: return $\{ (q', \mathsf{transguard}(q, q') \circ \mathfrak{y}') \mid (q, \cdot, q', \mathfrak{y}') \in S \}$
```

Transfer Relation $\rightsquigarrow_{Y_S}$. The transfer relation $\rightsquigarrow_{Y_S}$ builds on the relation $\rightsquigarrow_Y$ and strengthens its result by prepending Yarn that restricts the reachability of the successor states and the emitted Yarn by adding additional *guards and assignments* [94] *on the state variable* $\mathtt{cl}_Y$ that encodes the current control state of the transducer Y.

Algorithm 6 outlines the functionality of the transfer relation $\rightsquigarrow_{Y_S}$ with symbolic control encoding. We add state guards and new assignments to the state variable $\mathtt{cl}_Y$ if there is a control state transition, that is, if there exists at least one tuple $(q, \cdot, q', \cdot) \in S$ with $q \neq q'$, otherwise we do not adjust the transitions and their output: We create the Yarn with a guard and assignment on the state variable using the operator $\mathsf{transguard} : Q \times Q \to \mathfrak{Y}$. Given a call $\mathsf{transguard}(q, q')$, with the predecessor control state q and the successor control state q', it returns a Yarn $\mathfrak{y} = (h_t, G_g, \{l_a\}, \{l_x\})$ with $G_g = \{(l_a, \mathtt{cl}_Y\texttt{==}\mathtt{qn}(q), l_b), (l_b, \mathtt{cl}_Y\texttt{=}\mathtt{qn}(q'), l_x)\}$. An optimization of is to not add guarding assumptions, but the assignment, if the preceding abstract transducer state ι only contains one control state.

144 *transguard*

Operator merge_{Y_S}. The operator $\mathsf{merge}_{Y_S} : J \times J \times \Pi_Y \to J$ always joins two abstract states if none of them is a target state. That is, target states get kept separate to avoid the *problem of hidden targets* in case an analysis would not continue the state space exploration after a violating (target) state—which would affect completeness. Given a call $\mathsf{merge}_{Y_S}(\iota, \iota', \pi)$, the operator returns ι' if either $\mathsf{target}(\iota)$ or $\mathsf{target}(\iota')$ evaluate to *true*, it returns $\iota \sqcup \iota'$ otherwise. We assume that the transfer relation always produces singleton state sets—or empty sets, which stops the state space exploration at that point, and the operator merge is not called.

Hybrid. We configure the analysis to have hybrid control encoding by using the pair $\chi_{hybrid} = (\rightsquigarrow_{Y_S}, \mathsf{merge}_{Y_H})$ of operators.

Operator merge_{Y_H}. The operator $\mathsf{merge}_{Y_H} : J \times J \times \Pi_Y \to J$ joins two abstract states if (1) none of them is a target state, and (2) if none of their control states is listed for explicitly tracking—or dually: if the are listed for symbolic tracking. That is, the operator merge_{Y_H} extends the operator merge_{Y_S} and can keep abstract states with particular control states separate. Given a call $\mathsf{merge}_{Y_S}(\iota', \iota'', \pi)$, with $\iota'' = \{(q''_1, \cdot), \ldots, (q''_n, \cdot)\}$, $Q'' = \bigcup \{q \mid (q, \cdot) \in \iota''\}$, and $\iota' = \{(q', \cdot)\}$, the operator returns ι' if (1) $\mathsf{target}(\iota')$ or $\mathsf{target}(\iota')$ evaluate to *true*, or (2) if $\mathsf{sepctrl}(q', Q'', \pi)$ evaluates to *true*; it returns $\iota \sqcup \iota'$ otherwise. The operator sepctrl is defined as follows:

Operator sepctrl. The operator $\mathsf{sepctrl} : Q \times 2^Q \times \Pi \to \mathbb{B}$ determines if a given control state should be kept separated from other control states in the presence of the given abstraction precision. That is, a control state can be kept separated in certain parts of the state space, whereas it becomes joined into a transducer state with other control states in other parts—which enables the idea of Lazy Abstraction [136]. Different strategies to keep the state-space separated can be implemented and evaluated based on this operator. One strategy can be, for example, to keep control states that belong to different properties to check separated—in case all properties of the specification to check are represented by one YARN transducer.

4.6 EMPIRICAL STUDY

This chapter introduces several concepts and techniques, whereas their usefulness and applicability have been discussed on a theoretical level or small examples. We now present an empirical study that aims to (1) provide empirical evidence on the *practical applicability* of the concepts and techniques, and (2) show how different configurations of these *techniques influence the performance* of a model checker. We focus our study on model checking that aims at proving the absence of violations of safety properties.

4.6.1 Research Questions

Our study is guided by a set of research questions. We operationalize these questions later in the form of experiments. Before we continue to the questions, we define some terms: We call an analysis configuration *preferable* if it increases either the (1) efficiency or (2) the effectiveness of the verification procedure; also a (3) decrease of the size of the abstract model, or (4) the reduction of the number of refinement iterations can be preferable for some applications. The *performance* of a verification procedure is characterized by its efficiency and effectiveness. We measure the *efficiency* of a verifier in terms of CPU time needed to provide a solution (verdict) for a given verification task. We measure the *effectiveness* of a verifier in terms of solved verification tasks, that is, the number of tasks for that the verifier provided a solution.

145 Performance: Efficiency and Effectiveness

Earlier, in Sect. 4.1.2, we have motivated the need of having a symbolic control encoding with the fact that an explicit encoding of the control state of automata can counteract the symbolic encoding of the state space of the program to check. Our first question aims at studying the actual impact of the choice of control encoding on the performance of a model checker that models the state space of a program symbolically:

RQ 1 (Interaction). To which extent does an *explicit control encoding counteract a symbolic encoding* of the state space of the program to analyze, and how does this affect the performance of the verification procedure?

We have discussed, in Sect.3.5.2, several configurations of a verifier for composing transducers and their states. The following questions aim at studying the influence of these configuration choices on the performance of a verification procedure—to come up with a recommended configuration for a given set of verification tasks.

RQ 2 (Partitioning). Is there a set of verification tasks for which *separating (partitioning) the encoding* of control states by concern provides a better performance in terms of efficiency or effectiveness—compared to not separating the control states of different concerns.

RQ 3 (Explicit). Is there a set of verification tasks for which *explicit control encoding* is preferable over symbolic control encoding in the sense that it provides a better efficiency or effectiveness?

RQ 4 (Symbolic). Is there a set of verification tasks for which *symbolic control encoding* is preferable over explicit control encoding in the sense that it provides a better efficiency or effectiveness?

RQ 5 (Hybrid). Are there verification tasks for which a *hybrid control encoding* is preferable over both explicit and symbolic control encoding in the sense that it provides a better efficiency or effectiveness?

4.6.2 Experiment Setup

This subsection describes the chosen case studies and the configurations of the verifier. We provide a replication package along with this work, which also describes details on the benchmarking environment—see the appendix of this work for details.

Case Study Linux. The case study Linux consists of 250 Linux kernel modules that we consider *hard* in terms of the size of the abstract state space and the time needed to solve them. These modules were filtered out of 4 332 modules [8] and cause at least one refinement, result in an abstract reachability graph with at least 5 000 abstract states, yield the verdict *true*, and take at least 30 s to verify with our standard predicate analysis configuration. One verification task is composed of one Linux kernel module and a formal specification that consists of 14 properties to check—see Tab. 13 on page 171 for details of the checked properties. The number of different kernel modules was reduced to allow for sensitivity analyses on our given hardware resources. We use a timeout of 900 s and a memory limit of 30 GB, with 24 GB for the JVM, for each verification task.

Case Study Scenarios. We use a number of handcrafted scenarios to evaluate our techniques. A *scenario* is a *family* of programs that focuses on a specific problem dimension of a verification task. Based on these scenarios, we generate programs with different size and complexity to gain a fundamental understanding of different configurations of the presented techniques. Please note that we use a timeout of 600 s for each verification task from the case study Scenarios.

146 *Scenarios*

SCENARIO **Sequential**. Programs from the family Sequential are parameterized, and scaled, by the number of lock–unlock-properties $N \in \mathbb{N}$. In this scenario, a sequence of N conditional calls to `lockI()` is conducted, followed

```
1  int a1 = nondet();
2  int a2 = nondet();
3  ...
4  int aN = nondet();
5
6  if (a1) lock1();
7  if (a2) lock2();
8  ...
9  if (aN) lockN();
10
11 // while(nondet())
12   access();
13
14 if (a1) unlock1();
15 if (a2) unlock2();
16 ...
17 if (aN) unlockN();
```

(a) Sequential

```
1  int a1 = nondet();
2  int a2 = nondet();
3  ...
4  int aN = nondet();
5
6  if (a1) {
7    lock1();
8    if (a2) {
9      lock2();
10     if (a...) {
11       lock...();
12       if ...
13     } } }
14
15 // while(nondet())
16   access();
17
18 if (a1) {
19   unlock1();
20   if (a2) {
21     unlock2();
22     if (a...) {
23       unlock...();
24       if ...
25     } } }
```

(b) Nested

```
1  int s = 0;
2  while(1) {
3    if (s==0) {
4      lock1();
5      s = 1;
6    } else if (s==1) {
7      lock2();
8      s = 2;
9    } else if (s==2) {
10     lock..();
11     s = ...;
12   }
13   ...
14   } else if (s==N-...){
15     unlock...();
16     s = N-1;
17   } else if (s==N-1) {
18     unlock...();
19     s = N;
20   } else if (s==N) {
21     unlock...();
22     s = 0;
23   }
24 }
```

(c) Reactive

Figure 20: Our control-encoding evaluation scenarios

by the same number of conditional calls to `unlockI()`. Figure 20a illustrates the scenario.

SCENARIO **SequentialWhile.** The scenario SequentialWhile is similar to Sequential except that the loop in line 11 is activated—by uncommenting the line. The presence of these loops forces the analysis procedure to compute an abstraction on the loop head.

SCENARIO **Nested.** Programs from the family Nested—which is shown in Fig. 20b—are parameterized, and scaled, by the number of lock–unlock-properties $N \in \mathbb{N}$. The parameter N determines the number of nested and conditional lock–unlock-pairs.

SCENARIO **NestedWhile.** The scenario NestedWhile is similar to Nested except that the loop in line 15 is activated—by uncommenting the line. This modification causes the analysis procedure to compute an abstraction on the introduced loop head.

SCENARIO **NestedWhileSequential.** Programs from the scenario NestedWhile-Sequential are a sequential composition of functionality (and the control flow) of the scenarios NestedWhile and Sequential; the code of scenario Sequential is called after line 25 of scenario NestedWhile. The scenario is scaled by the parameter $N \in \mathbb{N}$, which defines both the nesting depth of scenario NestedWhile and the number of subsequent calls of scenario Sequential.

SCENARIO **Reactive.** Programs from the scenario family Reactive—which is shown in Fig. 20c—are scaled by the number of loop states $N \in \mathbb{N}$, which are possible values of program variable s. It imitates the control-flow structure that can be found in programs that are present in reactive systems (event-condition-action systems).

SCENARIO **ReactiveNested.** Programs from the scenario family ReactiveNested compose the functionality (and the control flow) of the scenarios Nested and Reactive; the code of scenario Nested is called after line 21 of scenario Reactive. The scenario is scaled by parameter $N \in \mathbb{N}$, which defines both the number of loop states of the code from Reactive and the nesting depth of the code from scenario Nested.

SCENARIO **ReactiveSequential.** Programs from the scenario family ReactiveSequential compose the functionality of the scenarios Sequential and Reactive; the code of scenario Sequential is called after line 21 of scenario Reactive. The scenario is scaled by the parameter $N \in \mathbb{N}$, which defines both the number of loop states of the code from Reactive and the number of sequential locks and unlocks as conducted for scenario Sequential.

Verifier Configuration. Our implementation is based on CPAchecker [32, 34]. We configure a program analysis based on predicate abstraction [122] with adjustable-block encoding [35]. The refinement is driven by spurious counterexamples (CEGAR) [67] from which we derive Craig interpolants [43, 84]; the set of predicates to compute predicate abstractions is derived from the Craig interpolants. The reachability graph is constructed from scratch—that is, beginning from the initial abstract state e_0—after each precision refinement. The abstraction precision—in the form of a set of predicates—is shared globally, that is, the same set of predicates is used for all abstraction computations per iteration of the CEGAR loop.

4.6.3 Experiments

To answer our research questions, we conduct a series of experiments. We study multi-property verification [8] configurations, that is, *all* properties of a specification are verified *at once, simultaneously*. We do not study configurations that partition the set of properties or verify only a subset of the properties—we did this in previous work [8]. We consider only those verification tasks for that at least two properties are relevant, that is, for that two properties are not vacuously [172] satisfied.

We focus on a set of (independent and controlled) variables: The control encoding strategies $ENC = \{\mathsf{Symbolic}, \mathsf{Explicit}, \mathsf{Hybrid}_{\mathsf{Hints}}, \mathsf{Hybrid}_{\mathsf{Modulo}}\}$, the block operators $BLK = \{\mathsf{blk}_{\mathsf{L}}, \mathsf{blk}_{\mathsf{SBE}}\}$ that are used by the predicate analysis, the automaton compositions $ACO = \{\mathsf{Separate}, \mathsf{Union}\}$, the SMT solvers $SLV = \{\mathsf{MathSAT5}, \mathsf{SMTInterpol}\}$, and the case studies $CST = \{\mathsf{Linux}, \mathsf{Scenarios}\}$. More variables, including possibly confounding ones, are discussed in the threats to validity.

The hybrid control encoding strategy $\mathsf{Hybrid}_{\mathsf{Modulo}}$ is configured with the operator $\mathsf{sepctrl}_3$ that keeps every third control state $q \in Q_Y$ of a given YARN transducer Y separate if $\mathsf{qn}(q) \mod 3 = 0$, where the function qn maps each control state to a number. The control encoding strategy $\mathsf{Hybrid}_{\mathsf{Hints}}$ uses hints that were provided by the user—in forms of annotations of the

147
Hybrid Encoding Strategies

Table 3: Results for the case study Scenarios with the block operator blk_L, the solver MathSAT5, and Atoms as granularity of predicates. The best results are emphasized.

		blk_L, Separate				blk_L, Union			
		$\mathsf{Hybrid}_{\mathsf{Hints}}$	Symbolic	$\mathsf{Hybrid}_{\mathsf{Modulo}}$	Explicit	$\mathsf{Hybrid}_{\mathsf{Hints}}$	Symbolic	$\mathsf{Hybrid}_{\mathsf{Modulo}}$	Explicit
Task Set	Tasks	Solved Tasks				Solved Tasks			
Sequential	22	**22**	**22**	9	6	**22**	**22**	**22**	**22**
SequentialWhile	22	**10**	**10**	6	5	8	8	8	**9**
Nested	22	**22**	**22**	**22**	**22**	**22**	**22**	**22**	**22**
NestedWhile	22	10	10	**22**	**22**	7	7	8	**10**
Reactive	27	**27**	**27**	20	25	14	14	**18**	17
ReactiveNested	27	**14**	6	5	13	**7**	5	6	**7**
ReactiveSequential	27	8	**10**	5	4	3	8	**10**	**10**
NestedWhileSequential	27	**21**	10	6	5	4	6	6	**9**
Total	196	**134**	117	95	102	87	92	100	**106**

control states to encode: Whenever the verification task is composed with the scenario Nested, the specification that belongs to the nested control structure is encoded explicitly, that is, its states are kept separated. More elaborated hybrid encoding strategies could be implemented based on our framework. We stay with these simple strategies for a proof of concept.

Experiment 1. We first study different control encoding strategies based on the scenarios. That is, we study the configurations $K_1 = \{\mathsf{Scenarios}\} \times \mathit{ENC} \times \mathit{BLK} \times \mathit{ACO} \times \mathit{SLV}$. Later, we extend the set of studied configurations to $K_1' = K_1 \times \mathit{PGR}$ by different granularities of predicates $\mathit{PGR} = \{\mathsf{Atoms}, \mathsf{Interpolants}\}$. The parameter *PGR* decides whether or not the Craig interpolants that are discovered in the precision refinement procedure—see Sect. 2.4.2 for details on CEGAR—should be split into their atoms (Atoms) or of they should be kept as they were computed (Interpolants)—see Sect. 5.6.3 for a discussion of the granularity of predicates.

Experiment 2. The second experiment aims at studying the effects of different control encoding strategies based on the case study Linux: We study the configurations $K_2 = \{\mathsf{Linux}\} \times \mathit{ENC} \times \mathit{BLK} \times \mathit{ACO} \times \mathit{SLV} \times \mathit{PGR}$.

4.6.4 Results

We now discuss our findings separated by research questions. The overall picture is discussed in Sect. 4.6.5.

RQ 1: Interaction of Encoding Strategies. The state space of the program is encoded using a predicate analysis with predicate abstraction and adjustable block encoding [35]—which allows to chose from different block operators.

Table 4: Results for the case study Scenarios with the block operator $\mathsf{blk}_{\mathsf{SBE}}$, the solver MathSAT5, and Atoms as granularity of predicates.

		$\mathsf{blk}_{\mathsf{SBE}}$, Separate				$\mathsf{blk}_{\mathsf{SBE}}$, Union			
		$\text{Hybrid}_{\text{Hints}}$	Symbolic	$\text{Hybrid}_{\text{Modulo}}$	Explicit	$\text{Hybrid}_{\text{Hints}}$	Symbolic	$\text{Hybrid}_{\text{Modulo}}$	Explicit
Task Set	Tasks	Solved Tasks				Solved Tasks			
Sequential	22	6	6	6	6	6	6	5	5
SequentialWhile	22	6	6	6	6	6	6	5	5
Nested	22	18	18	22	22	7	7	10	19
NestedWhile	22	17	16	22	22	7	7	9	18
Reactive	27	27	27	27	27	9	9	9	20
ReactiveNested	27	17	12	14	18	12	5	8	11
ReactiveSequential	27	5	5	5	5	4	5	4	4
NestedWhileSequential	27	4	4	4	4	3	3	3	3
Total	196	100	94	106	110	54	48	53	85

The choice of the block operators determines the fraction of the state space to encode into one SMT formula. The general idea of these block encoding strategies—which are defined by the block operator—is that aggregating more information into *bigger formulas benefits performance*, and too big formulas could overwhelm a solver [28, 35]. Disjunctions of the block formulas of two abstract states are constructed whenever a merge of the corresponding abstract states is allowed.

blk_L *better*

148 Generally, we can observe the expected behavior: With single-block encoding, the different control-encoding strategies provide, compared to configurations with an abstraction strategy that summarizes only on loop heads, the worst performance. While there are some scenarios for which the block operator $\mathsf{blk}_{\mathsf{SBE}}$ aids in providing the best performance, the operator $\mathsf{blk}_{\mathsf{L}}$ is the best choice to get good overall performance for the scenarios. The best configuration with the operator $\mathsf{blk}_{\mathsf{L}}$ can solve 134 verification tasks from the case study Scenarios, whereas the best configuration with the operator $\mathsf{blk}_{\mathsf{SBE}}$ can solve only 110 tasks—see Table 3 and Table 4 for a detailed overview over the number of solved tasks. Figure 21 presents a sensitivity plot that illustrates that enabling large-block encoding is beneficial for most studied configurations; only three of the 15 configurations cannot benefit. The scatter plots in Fig. 22 provide another perspective on the results, which illustrates that the chosen control-encoding strategy has a larger impact if the block operator $\mathsf{blk}_{\mathsf{L}}$ is enabled. These results support the idea that control-encoding strategies should aid a large-block encoding, that is, provide the control state for symbolic encoding into block formulas.

The hypothesis that we formulate based on RQ 1 is that an explicit *control encoding* strategy counteracts the symbolic *encoding of the state space* of the program under analysis, and influences the performance of the verification procedure negatively.

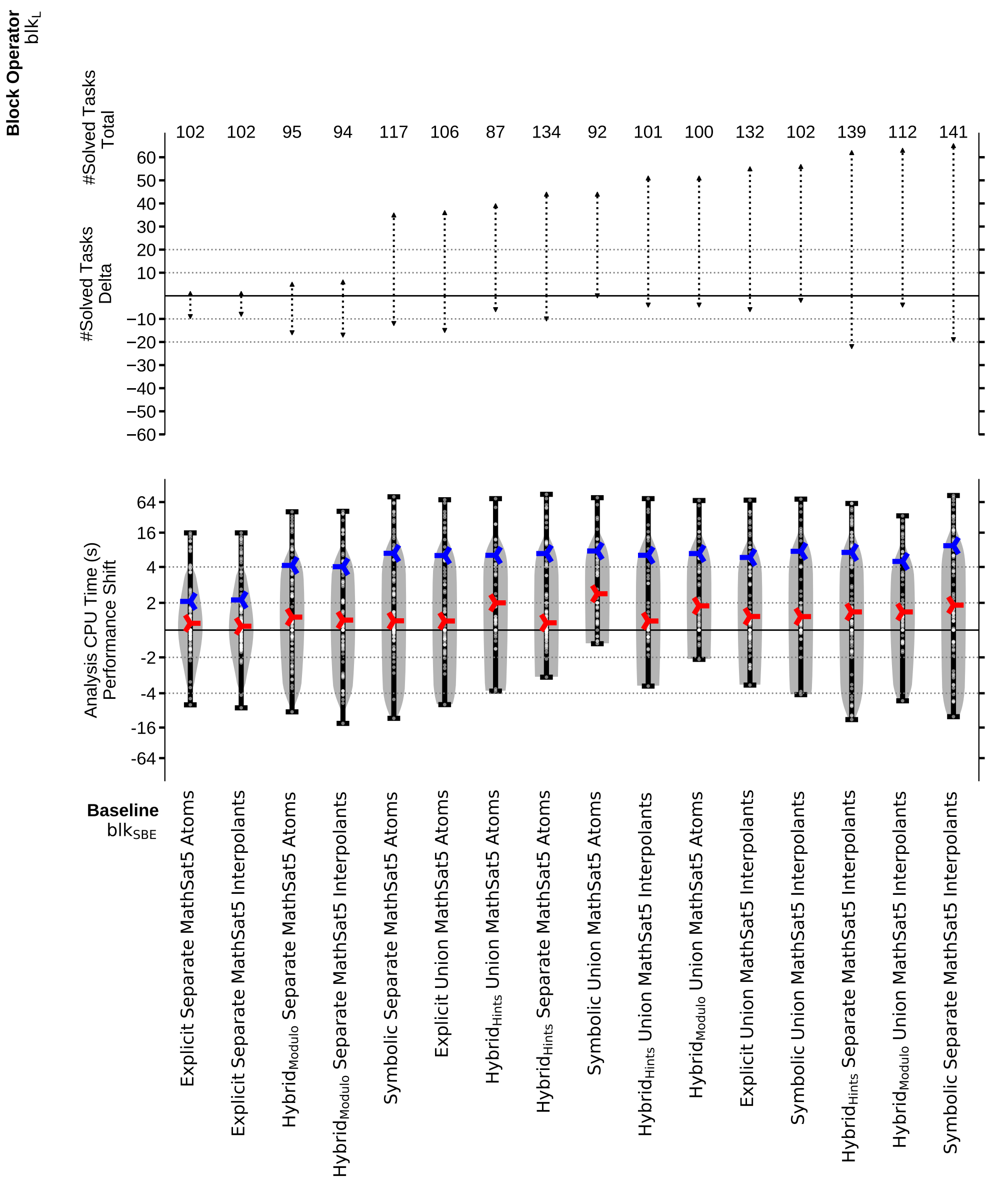


Figure 21: Sensitivity plot—see Sect. A.1.2—for the case study Scenarios that illustrates the extent to which the performance (analysis CPU time in seconds) of different control encoding configurations is *sensitive to the choice of the block operator*. The baseline configurations use the block operator blk$_{SBE}$, and we compare with configurations that are equal except that the block operator blk$_L$ is used. The mean performance shift is marked with the symbol $\prec$, the median is marked with the symbol $\succ$.

149 *Interaction with* blk$_L$

We first discuss verifier configurations for which the block operator blk$_L$ is used and all control states are kept separated (Separate), that is, each YARN transducer is executed in a separate analysis: For these configurations, we can see that an explicit control encoding only provides a better performance for the scenarios Nested and ReactiveNested; all other scenarios are solved with a better performance by either a purely symbolic or a hybrid control encoding configurations. That is, already this result allows us to *accept our hypothesis* that an explicit control encoding interacts with a symbolic control encoding of the state space of programs negatively.

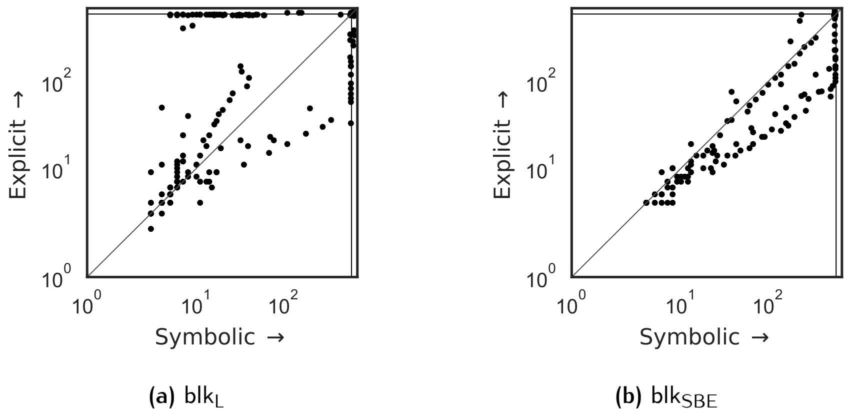


Figure 22: Scatter plots with the *CPU time for analysis* for the case study Scenarios. We use a verifier configuration with the solver MathSAT5; the set of predicates corresponds to the Craig interpolants (Interpolants), that is, they are not split into atoms.

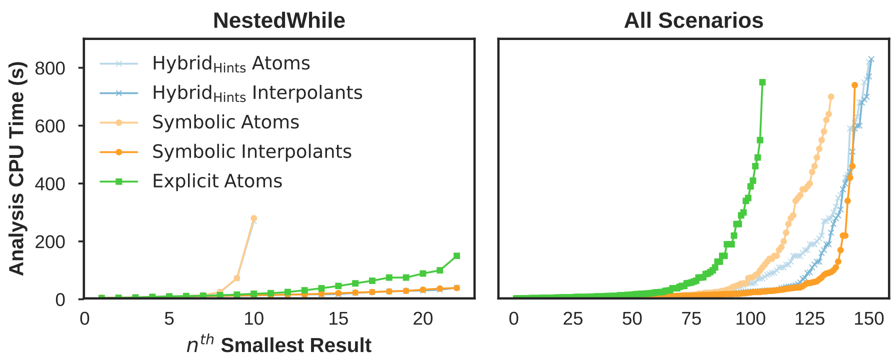


Figure 23: Quantile plots that illustrate the effect of *splitting Craig interpolants* into atoms (Atoms) vs. not doing so (Interpolants); the analysis configurations are based on the solver MathSAT5 and the block operator blk$_L$.

150 *Interaction with* blk$_{SBE}$

For verifier configurations with single-block encoding—which is enabled by the block operator blk$_{SBE}$—we cannot see a considerable difference in terms of solved tasks when choosing different control encoding—for both case studies. There is a difference in the efficiency (in terms of CPU time), which is due to the *number of precision refinements* (for predicates on the variables that encode the state of the specification transducers).

151 *Positive Effects of* Explicit *on* blk$_L$

There is a subset of the scenarios for which the control encoding strategy Explicit provides the best results, namely all scenarios that are composed with the scenario Nested or its variant NestedWhile. Symbolic control encoding configurations get into trouble because of a large *number of predicates*—which are derived by splitting the Craig interpolants of infeasible counterexamples into their atoms—*and the structure of the formula* to encode, which results from the structure of the control flow. The large number of predicates (along with the structure of the formula) results in a large number of satisfying assignments, which have to be enumerated to compute the Boolean predicate abstraction. Enabling explicit control encoding (1) reduces the number of predicates to track and (2) reduces the complexity of the formulas to compute abstractions for—by not allowing to merge abstract states and the block formulas that are stored along with them. We hypothesise that splitting Craig interpolants into atoms is not always good and should get done only if the control-flow structure of the underlying problem does not have a nested structure. That is, not splitting Craig interpolants into atoms can enhance the performance of a symbolic state space encoding in the presence of certain control flow structures. The first evidence for this hypothesis is illustrated in Fig. 23, which illustrates the positive effects of not splitting Craig interpolants.

We summarize the results for this research question as follows:

Summary (*RQ 1*) The performance of our verification procedure has a *considerable sensitivity* to the choice of the *block abstraction strategy*. The influence of the chosen control encoding strategy increases if large fractions of the state space of the program to analyze are encoded symbolically in large block formulas, as done, for example, in large-block encoding.

RQ 2: Partitioned Control Encoding. The hypothesis that we formulate based on RQ 2 is that a *separating the encoding of the control states* of different automata (partitioned encoding of the control state, enabled by Separate) provides a better performance, that is, it can reduce the CPU time for analysis and increase the number of tasks that can be solved. Figure 24 illustrates an abstract reachability graph for a configuration for which the control encoding is not partitioned, which partitions the state space.

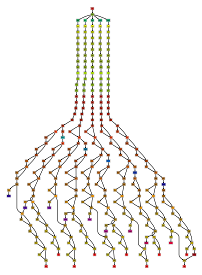


Figure 24: The effect of combining all YARN transducers into one using an union (Union) and separating its states (Explicit): The state space splits already after the initial abstract state $e_0 \in E$.

Already from Table 3 and Table 4 we can see that configurations with Separate can solve more tasks than configurations with Union enabled. The sensitivity plot in Fig. 25 provides details on the sensitivity of more analysis configurations. We can see that all configurations with Symbolic enabled can benefit considerably by the partitioning. Also, all configurations with $\mathsf{blk}_{\mathsf{SBE}}$ enabled work best if Separate is enabled.

Configurations with Union enabled cannot provide any results for the case study Linux. The reason is that the abstract state space explodes considerably for such configurations due to the early state space splitting—see Fig. 24.

Summary (*RQ 2*) A *partitioned control encoding* is in general *preferable*, that is, can provide a better or equal performance for the studied verification tasks. Especially verification tasks for which already the modeling of the program itself would result in a large state space, such as modules of an operating system, can benefit from a partitioned control encoding.

RQ 3: Explicit Control Encoding. Based on RQ 3, we formulate the hypothesis that there is a set of verification tasks for which an explicit control encoding provides the best performance in terms of CPU time for the analysis. The symbolic control encoding has to be superior for all verification tasks to falsify this hypothesis.

Already a first glance at the result indicates clear evidence for this hypothesis: The explicit control encoding provides the best result for the scenario NestedWhile among all studied configurations—see Table 3, Table 5, and Table 4. A closer look at the analysis runs and results for the scenario NestedWhile reveals that the positive effects of explicit control encoding are due to the granularity of the predicates for predicate abstraction—see our discussion for RQ 1. Changing the granularity of predicates to full

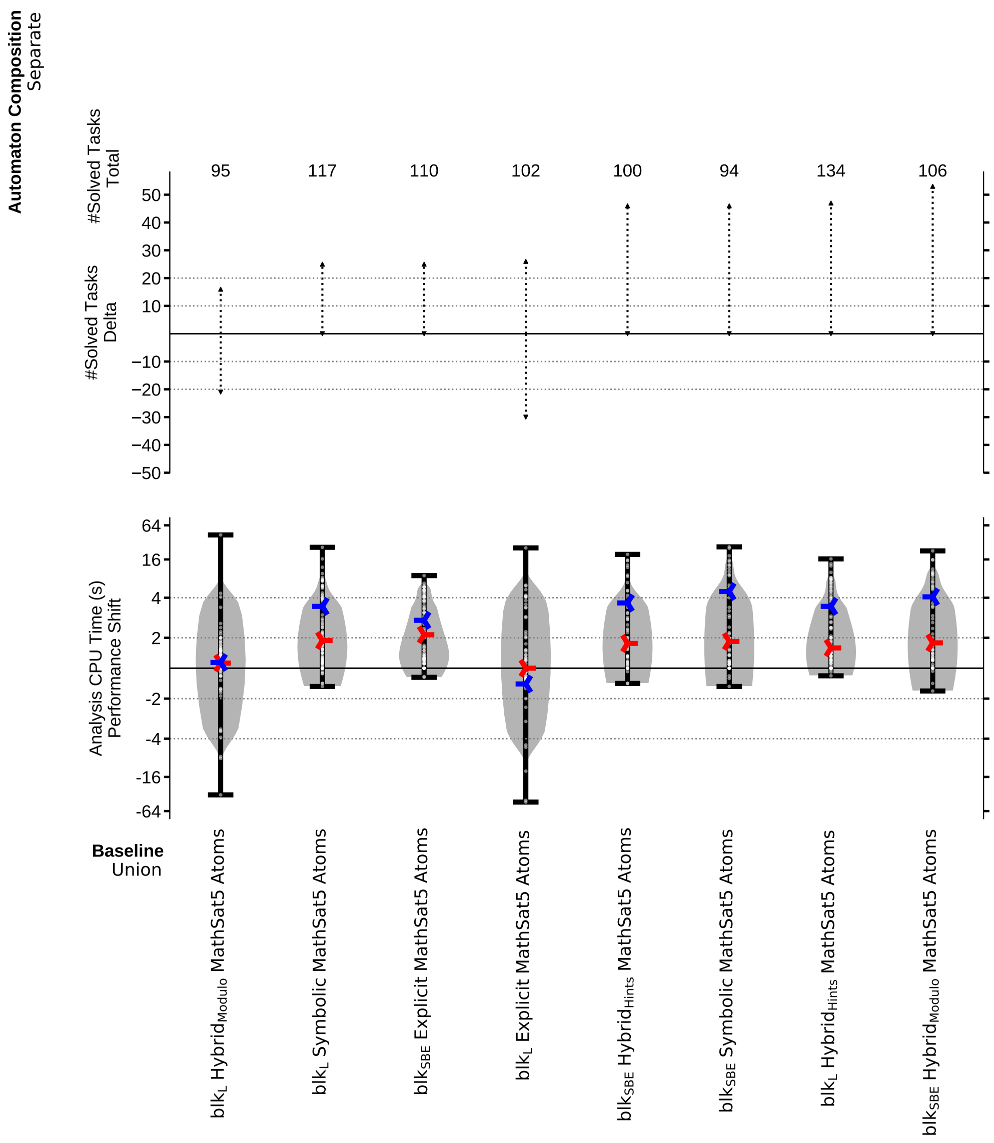


Figure 25: Sensitivity plot—see Sect. A.1.2—that illustrates the sensitivity of different control encoding configurations to the choice of the automaton composition parameter for the case study Scenarios: How does the performance change if Separate is used instead of Union?

Craig interpolats, yields (for this scenario) a symbolic control encoding performance similar to those of explicit control encoding strategy. Nevertheless, a change to this granularity of predicates affects the analysis performance for the scenarios Reactive and ReactiveSequential negatively.

Enabling the explicit control encoding (Explicit) has a considerable positive effect on the efficiency of configurations with single-block encoding (blk$_{SBE}$), which is illustrated by the scatter plot in Fig. 22b. The positive effect is due to the reduction of the abstraction refinement iterations. Reusing abstraction precisions (the set of predicates that is discovered by the abstraction refinement procedure) from other runs can cancel out this effect [36].

The results are different for the case study Linux: Configurations with an explicit control encoding enabled are superior for some verification tasks—

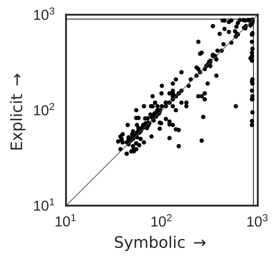


Figure 27: Scatter plot with the *CPU time for analysis* for the Linux case study.

Table 5: Results for the case study Scenarios with the block operator blk_L, the solver MathSAT5, and Interpolants as granularity of predicates.

		blk_L, Separate				blk_L, Union			
		$\mathsf{Hybrid}_{\mathsf{Hints}}$	Symbolic	$\mathsf{Hybrid}_{\mathsf{Modulo}}$	Explicit	$\mathsf{Hybrid}_{\mathsf{Hints}}$	Symbolic	$\mathsf{Hybrid}_{\mathsf{Modulo}}$	Explicit
Task Set	Tasks	Solved Tasks				Solved Tasks			
Sequential	22	22	22	9	6	22	22	22	22
SequentialWhile	22	22	22	6	5	18	18	17	9
Nested	22	22	22	22	22	22	22	22	22
NestedWhile	22	22	22	22	22	14	14	16	22
Reactive	27	14	14	19	25	8	8	9	15
ReactiveNested	27	8	6	6	13	10	5	6	9
ReactiveSequential	27	8	6	4	4	3	4	6	9
NestedWhileSequential	27	21	27	6	5	4	9	14	24
Total	196	139	141	94	102	101	102	112	132

Table 6: Results for the case study Scenarios with the block operator blk_L, the solver SMTInterpol, and Interpolants as granularity of predicates.

		blk_L, Separate				blk_L, Union			
		$\mathsf{Hybrid}_{\mathsf{Hints}}$	Symbolic	$\mathsf{Hybrid}_{\mathsf{Modulo}}$	Explicit	$\mathsf{Hybrid}_{\mathsf{Hints}}$	Symbolic	$\mathsf{Hybrid}_{\mathsf{Modulo}}$	Explicit
Task Set	Tasks	Solved Tasks				Solved Tasks			
Sequential	22	22	22	9	6	22	22	22	22
SequentialWhile	22	22	22	6	5	14	14	15	9
Nested	22	22	22	22	22	22	22	22	22
NestedWhile	22	22	22	22	22	12	11	16	22
Reactive	27	14	14	16	25	9	9	11	14
ReactiveNested	27	10	6	5	13	10	3	6	8
ReactiveSequential	27	8	8	4	4	3	3	5	8
NestedWhileSequential	27	21	27	6	5	4	11	14	26
Total	196	141	143	90	102	96	95	111	131

see the scatter plots in Fig. 27. The sensitivity plot in Fig. 28 provides a different perspective on these results. Table 7 provides the results on the level of verification tasks.

Summary (*RQ* 3) An *explicit control encoding* of the control state of a specification automaton can have positive effects on the performance of a verification procedure. However, some of the effects are more a performance interaction than something that can be attributed to a more efficient control state encoding.

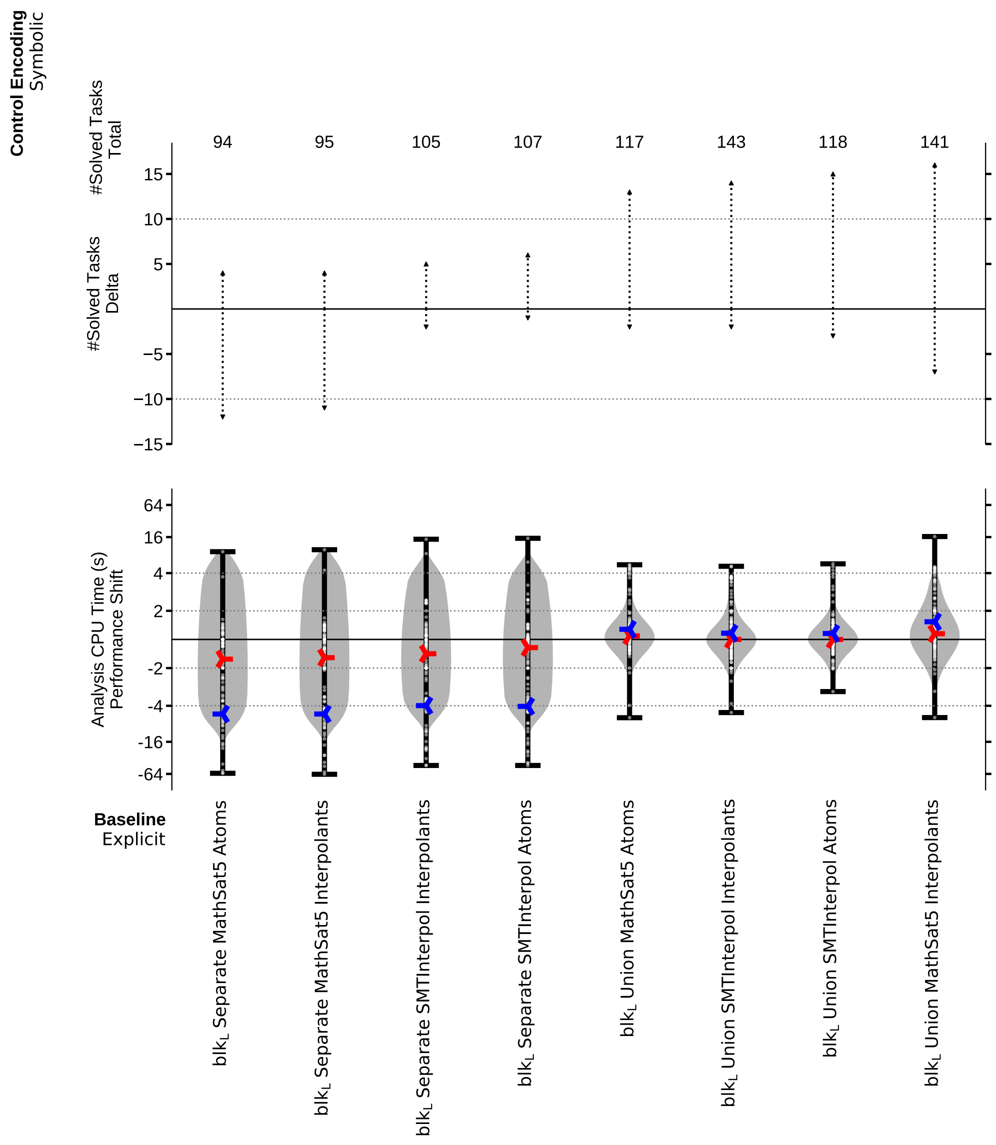


Figure 26: Sensitivity plot—see Sect. A.1.2—that illustrates the sensitivity of different analysis configurations to the choice of the control encoding strategy for the case study Scenarios: How does the performance change if Symbolic is used instead of Explicit?

RQ 4: Symbolic Control Encoding. From RQ 4 we derive the hypothesis that there is a set of verification tasks for which symbolic control encoding is superior to explicit control encoding in terms of CPU time spent for the analysis. To falsify this hypothesis, there must be no symbolic encoding that provides a better performance than an explicit control encoding. This is not the case: A symbolic control encoding is preferable for the majority of the verification tasks of the case study Scenarios, especially in terms of effectiveness. Nevertheless, this highly depends, as we already discussed previously, on the techniques that are implemented on the analysis that conducts the symbolic encoding, for example, the granularity of predicates that are used for predicate abstraction.

Table 7: Performance shifts on the level of verification tasks for the case study Linux, with the solver SMTInterpol, and with Craig interpolants split into Atoms. We show only tasks with a shift of at least 1 in either direction.

	Symbolic	Explicit		Symbolic	Explicit		Symbolic	Explicit	Symbolic	Explicit
Program	Analysis Time (s)		Shift	|**reached**|		Shift	#Refinements		Max. Models	
crypto-ccp-ccp-crypto	130	770	●┼	32 080	69 751	●┼	16	16	2	2
realtek-atp	190	490	●┼	19 254	19 618	●	16	14	3	3
dvb-frontends-stv0900	170	380	●┼	144 991	145 106	●	9	9	1	1
dvb-frontends-stv0367	140	290	●┼	134 052	134 167	●	9	9	1	1
libertas_tf-libertas_tf	140	280	●┼	27 615	28 086	●	10	10	2	2
marvell-sky2	680	320	┼●	88 321	46 557	┼●	14	13	3	2
amd-nmclan_cs	160	71	┼●	23 406	8 980	┼●	15	14	4	2
atheros-atl1e-atl1e	380	160	┼●	30 323	30 177	●	7	6	3	3
atheros-atlx-atl2	210	88	┼●	34 432	13 232	┼●	13	12	2	2
dec-tulip-de2104x	330	120	┼●	46 195	15 778	┼●	16	15	8	9
atheros-atlx-atl1	430	150	┼●	44 496	24 557	┼●	13	12	4	3
micrel-ks8851	270	82	┼●	32 442	8 765	┼●	26	24	14	12

Summary (*RQ* 4) A *symbolic control encoding* of the control-state of specification automata *can be preferable*, that is, can provide a better performance, for many verification tasks. We have provided solid evidence that symbolic control encoding can have a considerable positive effect on the performance of a verification procedure, both in terms of solved tasks and in terms of spent CPU time for the analysis.

RQ* 5*: Hybrid Control Encoding. We answer RQ 5 based on the hypothesis that there is a set of verification tasks for that hybrid control encoding is superior to both an explicit control encoding and a symbolic control encoding either in terms of CPU time for the analysis or the number of verification tasks. We can see from Fig. 29 that the performance of configurations with Hybrid enabled is in-between the performance of Symbolic (with Interpolants enabled) and Explicit configurations for the case study Scenarios. The reason for the performance advantage of hybrid control encoding configurations for some verification tasks—see also Table 3—is due to an interaction with the predicate abstraction mechanism, as already discussed for RQ 1.

Summary (*RQ* 5) A *hybrid control encoding* of the control-state of specification automata *can have an effect* on the performance of a verification procedure for certain verification tasks. We have *no evidence* that verification engineers should aim to use a hybrid control encoding to get the best performance from a well-engineered verification tool that uses predicate abstraction—with the best possible granularity of predicates—to model the state space.

4.6.5 Discussion

Our study shows the *practical applicability* of techniques that we have introduced along with this chapter, first of all, the LOOM analysis with its service to compose YARN into the transition system of the analysis task, on-the-fly, during the analysis.

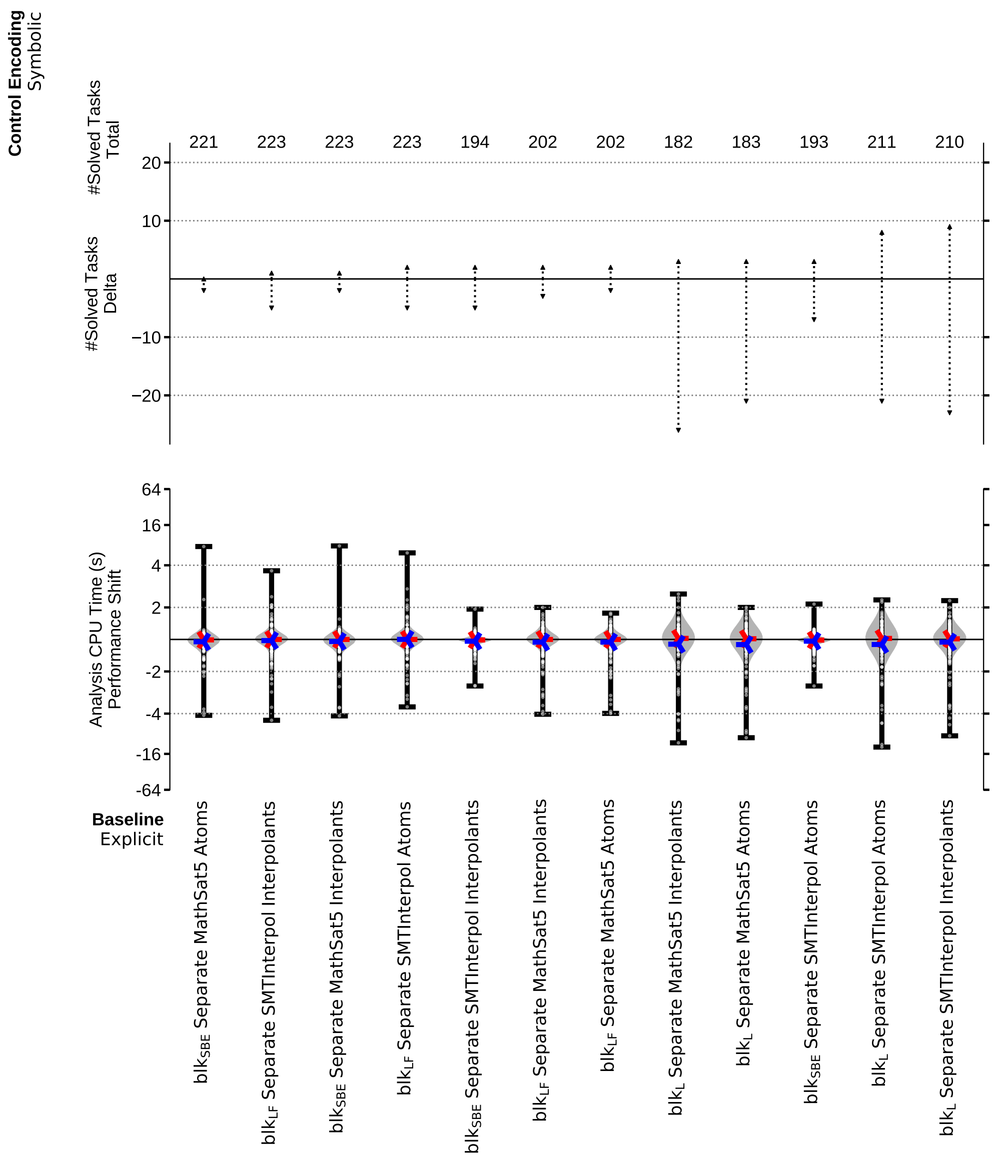


Figure 28: Sensitivity plot that illustrates the sensitivity of different analysis configurations to the choice of the control encoding strategy (Explicit vs. Symbolic) for the case study Linux.

Delegating the encoding of control states to another analysis by introducing YARN to weave provides more choices for of encoding the state space—including the control state of finite-state machines—of the verification task, and we can benefit from the optimizations and contributions that were made for the more powerful general-purpose analysis techniques—such as analyses based on predicate domains [28, 122] with predicate abstractions.

Our study shows that the performance characteristics of a modern verification tool are dependent on many variables that *influence each other*—that is, there is a performance interaction between them. We have learned that an explicit encoding of certain information such as the current control state of a finite-state machine, or the value of a program variable in general, influences the size and complexity of block formulas as used for predicate abstraction—by not allowing to merge some abstract states and with it to construct dis-

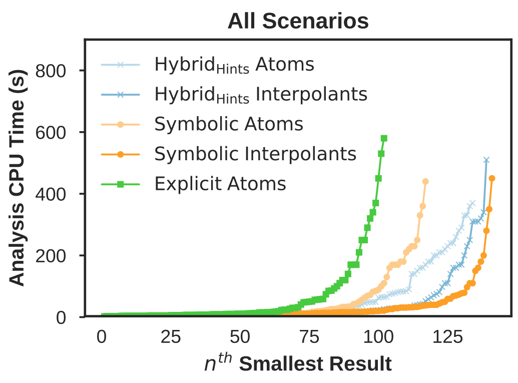


Figure 29: Quantile plot for the case study Scenarios and different control encoding strategies. With MathSAT5, Interpolants, and blk$_\text{L}$ enabled.

junctions of block formulas. The size and complexity of block formulas influence (1) the quality of Craig interpolants a refinement procedure can come up with, (2) the costs for checking the formulas for (un-)satisfiability, (3) the number of precision refinements that are needed, and (4) the size of the abstract state space, and with it the chance to converge to a fixed point in the state space exploration process.

If we would only consider the results for the case study Scenarios we could conclude that a symbolic encoding of the current control state of finite-state machines is the best choice in the presence of a verification tool that implements state-of-the-art techniques—or even some techniques that are beyond state of the art, such as a structure-aware dynamic choice of the *granularity of predicates*. However, we also have to take the results for the case study Linux into account, for which an explicit control encoding seems to be the best choice. It turns out that saved abstraction *refinement iterations*, which result from the explicit encoding of control states, can provide a performance benefit for this type of task. We take advantage of this observation in the next chapter, where means are presented to automatically synthesize predicates from the Yarn that is emitted for control encoding.

From a more general perspective, this study demonstrates the practical applicability of Yarn transducers and the Loom for program verification: Different properties to verify are encoded in a set of Yarn transducers, the Loom analysis weaves the emitted Yarn, and the possibility to delegate the encoding of information to other analyses or analysis steps is demonstrated by our experiments on dynamic control encoding.

4.6.6 Threats to Validity

While we presented a carefully designed study, several threats can lower the validity of our results.

Studying the performance of a set of analysis configurations based on a set of scenarios aims at increasing the *internal validity* of our results. Nevertheless, there is an infinite number of possible program variants that are expressible based on a Turing-complete programming language. The focus on a small sample of verification might be prone to overfitting of the an-

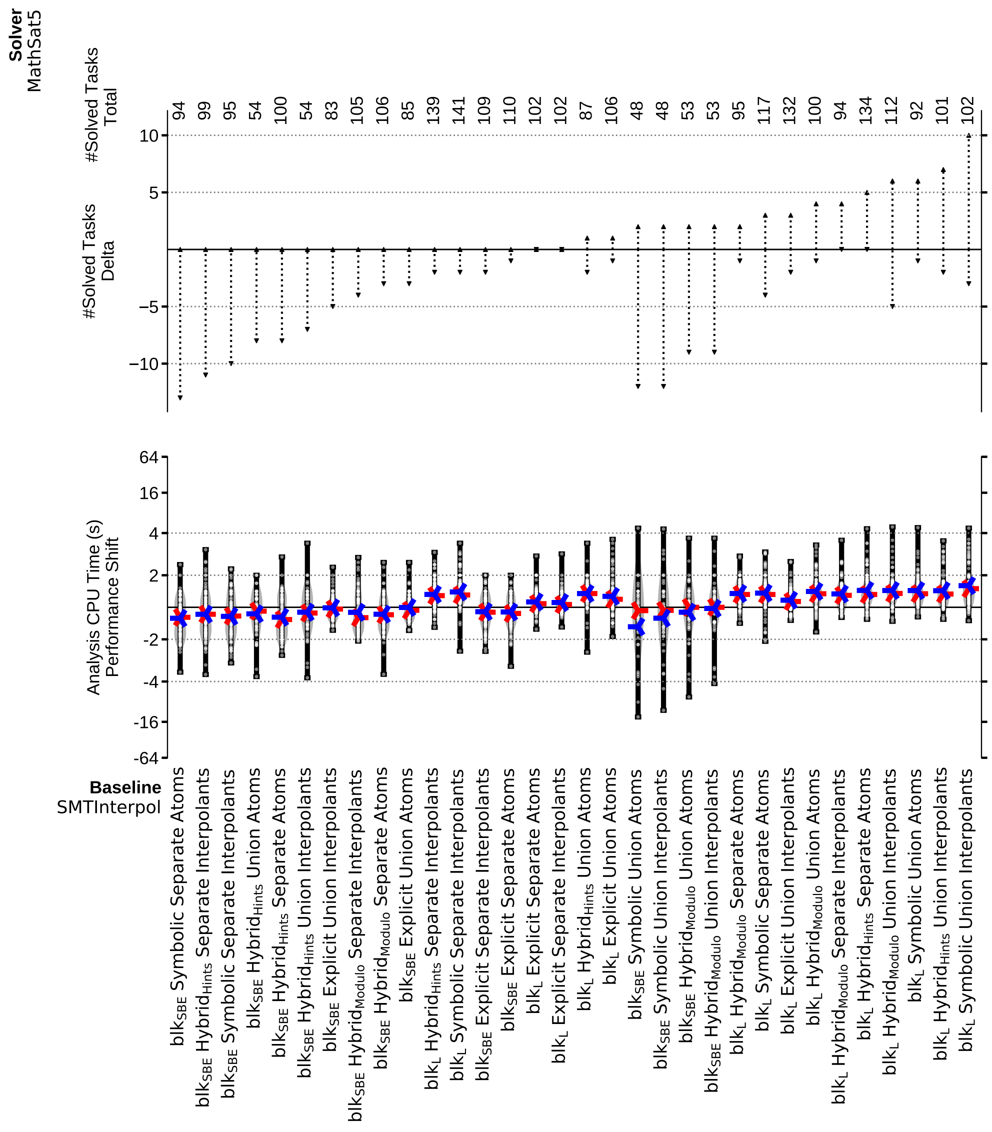


Figure 30: Sensitivity plot that illustrates the sensitivity of different analysis configurations to the choice of the solver for the case study Scenarios.

alyzed tool configurations—that is, could lead to optimizations that might only work well for the given set of programs. Nevertheless, having a small set of programs with a well-known structure (features) helps us to understand and describe the influence of different analysis parameters, and develop techniques that are applicable to a broader set of programs.

To increase the *external validity* of our results, we also studied a set of different analysis configurations on a set of real-world verification tasks that were derived from Linux kernel modules [8]. The experiments were conducted without a pointer aliasing activated, which can lead to different performance characteristics for those tasks for which the satisfaction of the specification would be dependent on aliasing. Nevertheless, since the checked properties are primarily control-dependent, we assume that the validity of our results is not affected dramatically.

Another threat to validity is the possible presence of *bugs* in the tool implementation, solvers, or libraries—see Sect.A.2 for a detailed discussion. Nevertheless, we have tried to ensure the correctness and soundness of the implementation as good as possible, especially to guarantee the correctness of the results for the case study Scenarios. The results for the case study Linux are more likely to be different with a tool that ensures that the semantics of all program operations—for example, those that are needed for low-level operating system development—are reflected soundly.

While we tried to increase the validity of our results by conducting sensitivity analyses, which study the effect of different parameters on our results, our results are only valid for the studied analysis configurations and the used set of verification tasks. The results can be different, for example, for bounded model checking, compositional model checking, or explicit-state model checking. We use an analysis configuration that makes use of an SMT solver and a BDD library. Both components are critical for the performance of our analysis procedure. Already an encoding of the state space using different solver theories can provide different efficiency and effectiveness. Figure 30 presents a sensitivity plot that illustrates how the results are changed if we change from the solver SMTInterpol to MathSAT5—it shows that there is considerable sensitivity to the chosen solver.

4.7 RELATED WORK

This chapter presents an approach for *composing* programs (analysis tasks) *by weaving*, also *on-the-fly*, that is, interleaved with their actual interpretation—for example, during state-space exploration that is conducted by a software model checker.

Task Specification. We formalize the program fragments to compose as Yarn and provide Yarn transducers for sharing and reproducing these sets of control transitions in different contexts of analysis tasks. Using transducers for this purpose is novel; so far, transducers have been used to describe the input–output relations of programs [49, 126, 216]. A Yarn transducer can emit programs that take full advantage of Turing completeness. The notion of Yarn is related to model programs [246], but those do not have an inherent mapping to program concerns.

Several techniques have been proposed to operationalize program specifications from higher-level specifications that are provided in some form of temporal logic or natural language. In this work, we present a specification mechanism that operates on the same level of abstraction as the syntactic model of the program to analyze. Many prominent specification approaches that are applied in practice are on the level of source code—and might add some temporal notion. Examples include the ANSI-C specification language (ACSL) [26], SLIC [22], and the BLAST query language [39, 232]. The specification can be provided as runtime assertions or can be transformed into such [90]. Also, general-purpose aspect languages have been used [13, 145] or extended [108] to allow for expressing the desired properties of a program. Formal specification is also used for test generation, where a specification consists of a set of test goals. Typically, trap properties [114, 127, 221] are specified, possibly in the form of trap monitors or test goal automata [127, 141]. That is, automata are an established mechanism for formally describing parts of a verification task. Based on the concept of abstract transducers, we present Yarn transducers that emit Yarn—sequences of control transitions, annotated with program operations—to weave. The notion of Yarn is not limited to Yarn transducers: Yarn to compose by weaving can be provided by any analysis component in the verification framework.

Composition Period. We distinguish between three different composition periods: *offline*, before the full composed verification task is handed over to the verifier, *initially*, before the state-space exploration is conducted, and *on-the-fly*, that is, along with the state-space exploration. Our Loom analysis is designed for on-the-fly composition by weaving but can be used in all other points of time too. Traditionally, the verification task is composed either offline [22, 90, 241], or initially [141, 194, 252, 254]. A specification that is handed over to the verifier as a set of automata can be composed either initially, that is, by an eager automaton product computation [134], or on-the-fly and by computing the automaton product lazily—as already done for model-driven test generation [33], and in our work on testification of error witnesses [29]. Work in the field of dynamic software updating [62, 203] can be considered an instance of on-the-fly weaving.

Task Composition. Several techniques to compose verification tasks from different parts (aspects, components, modules, features, automata) exist. We now focus on *how* these techniques conduct the composition. Many of these techniques have their roots in the fields of aspect-oriented [160] and feature-oriented programming [6]. Aspect weaving [160] is the standard choice for composing verification tasks offline, as done, for example, by SLIC [26].

For some properties, such as checks for NULL-pointer dereferences or overflows of buffers or numbers, no full aspect weaver is needed, and simpler weaving techniques that traverse and manipulate control-flow graphs or the program's abstract syntax tree can be sufficient [102, 194, 252, 254]. Some approaches encode these properties directly into SMT formulas [194].

Another form of composition is to construct the product of the given automata directly; this can be done eagerly (and initially) [134] or lazily (and on-the-fly). The composition approach that we present in this work can compute a lazy automaton product of transducers with outputs that carry information that is relevant for the verification task. We present the first holistic approach that can perform a composition by weaving on-the-fly, that is, along with state-space exploration, while YARN transducers emit the program fragments to weave—both control-dependent, but also data-dependent properties can be expressed.

Program Composition. The LOOM is a generic concept to compose a program by weaving from given YARNS, that is, from sets of sequences of program operations that provide functionality for specific program concerns. Arbitrary analyses and analysis components that are executed in parallel to a LOOM analysis can provide YARN to weave at different points of the abstract state space, for various purposes.

Concepts like aspect-oriented programming [160], delta-oriented programming [226], or feature-oriented programming [6, 219] are orthogonal to the LOOM and YARN transducers since they describe programming paradigms. Certain code artifacts and composition rules that can be defined based on these paradigms can be operationalized as YARN transducers, where the YARN describes the code artifacts and the composition roles are translated to a corresponding control transition relation of the YARN transducer. Aspects for monitoring temporal properties [5, 12, 13, 98, 145] of program traces are examples of aspects that translate well to YARN transducers: These aspects have to be woven after particular sequences of events have been observed [98]. On the language-theoretic level, rewriting systems for graphs [11] and terms [176] are closely related to our composition approach, which results from the combination of the LOOM and YARN transducers.

Control Encoding. The choice of encoding the current control state of the composed automata is critical for the performance of a verifier. Our approach allows for encoding the control state of the resulting transition relation in a *symbolic, explicit, or a hybrid* fashion.

Classical bounded model checkers [45] encode the full state space into one (big) formula, that is, a symbolic representation of the states of composed automata is used. This is achieved by instrumenting the automata offline into the code, using global state variables that store the current state of the automata [241]. The bounded model checker does not distinguish these control state variables from other program variables. Work in the context of IC3 [66] has shown that a non-full symbolic encoding of the state space, more precisely an explicit encoding of the control locations of the control-flow automaton, leads to a significant performance benefit. Nevertheless, the product of a set of automata is exponential in the number of automata to compose. That is, a verifier that constructs such a product is more likely to have to deal with an exploded abstract state space—please note that also the control-flow automaton on that, for example, the specification automata become composed with, must be included into this calculation. Existing work only considers tracking the control state of the control-flow automaton of the program under analysis either explicitly or symbolically. This work provides this choice also for other types of finite-state machines that should be composed during the analysis process. Our approach provides the possibility to choose from different encodings for different parts of the state space, which follows the idea of Lazy Abstraction [136].

In case the current control states are stored in program variables, a model checker could always conduct a domain-type [7] analysis and identify these state variables as variables that store a current control location or control state, and might then use different abstract domains to track these variables. That is, tools that consume these programs *could* analyze the type and roles of different variables [7, 68, 91] and decide to encode the control state of instrumented automata either explicitly, symbolically, or in a hybrid fashion.

4.8 SUMMARY

152 *On-the-Fly Weaving*

This chapter has presented a holistic approach for *composing* analysis tasks *by weaving*, also *on-the-fly*, that is, during the state space traversal process. We presented a technique to *encode the control* location of the resulting transition relation dynamically in a *symbolic, explicit, or a hybrid* fashion.

153 *Automata Product*

The combinatorial explosion of the abstract state space that results from constructing the product of several automata—which we construct lazily—motivated our work on composing additional transition relations—resulting from the transitions of the automata—symbolically, and lead us to consider encodings of control locations—or control states—that range between an explicit and a symbolic representation.

154 *Loom Analysis*

The Loom *analysis* can take sequences of control transitions—that is, the Yarn—to weave from any analysis component that is executed along with it.

155 *Yarn Transducers*

We introduce Yarn *transducers* as a means to systematically provide Yarn to compose by weaving at different points in the control flow of an analysis task. Yarn transducers are built on top of abstract transducers, which demonstrates the applicability of abstract transducers for sharing syntactic task artifacts for reuse.

156 *Empirical Study*

An empirical study demonstrates the practical applicability of our concepts and techniques. We study different performance characteristics of these techniques based on a set of scenarios that have a well-known structure and based on a set of Linux kernel modules. We show that the different types of control encodings are beneficial depending on the control-flow structure of the analysis task to compose with.

157 *Outlook*

We restricted our discussions on applications in the context of multi-property verification and model-driven test generation, with a focus on safety properties, and on the composition of sequential programs. Nevertheless, our techniques, for example, the Loom, could also contribute to the composition and analysis of concurrent systems, and it could be used for weaving termination arguments [208], that is, conduct the program transformation that is needed for a termination analysis [77, 229].

“*The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.*”

— Donald E. Knuth

5 PRECISION TRANSDUCERS

Keywords: Abstraction Precision, Model Construction, Precision Reuse, Precision Transducer, Existential Abstraction, Continuous Verification

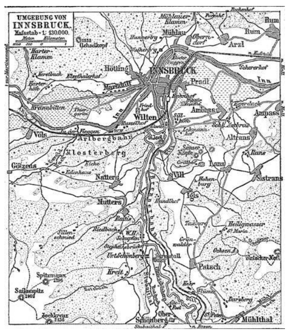


Figure 31:
A map of Innsbruck

Abstraction is the key to construct effective models of a system efficiently—and is considered to be an essential skill in computer science [75, 168, 239]. A model is *effective* if it is sufficiently precise to satisfy its purpose, for example, to reason about a property of the system. Having to model only a few details (abstraction) about a system reduces the effort (*efficiency*), and the modeling process can be finished after reaching a fixed point. The set of facts to model about a system is called the *abstraction precision*.

158 Maps: Models of the World

We start with the creation and use of maps as the first real-world example to discuss the process of creating abstract models and the role of the abstraction precision. A map is created for a specific purpose and audience: There are maps, for example, for hikers, for motorists, or maps that provide information on specific natural resources that are available in a region. Moreover, maps can have different scales, depending on the needed granularity of information to satisfy their purpose. The creation of maps can be cumbersome and cause high costs. Having an existing map as a basis can reduce the effort needed considerably, even if this other map was created for a different purpose. That is, also an old map—as illustrated in Fig. 31—is precious and should be kept for further use. Mental maps are another type of map, with a similar purpose [197]: Programmers construct and use mental maps of source code and its functionality. Experienced programmers have a large set of different mental maps of the source code at their disposal—and can *reuse* them—to solve programming and maintenance tasks. These experts can *share* their experience and mental models—especially, which details of the source code are relevant for which purpose—to other, less experienced, programmers, which can *use this experience* to become more efficient and effective in their job. Let us connect this with thoughts from the introduction of abstract transducers in Chapter 3: We have described books—and the printing press—as an approach for spreading (sharing) information: A (1) text's

159 Mental Maps

structure guides the thoughts of the reader, and (2) its semantics invites the reader to reconsider his or her model of the world, that is, it contributes to the process of refining his or her model of the world. Interestingly, also the ordering in that information is provided to the reader—a form of context—has a crucial role in perceiving information: the effects of priming and recency [17, 118]. 160

Precision Transducers

In this chapter, we propose (and share) precision transducers as a means for software model checkers to share at which parts of the state space which information must be modeled (abstraction precision) to reason about a property—to prove its correctness. That is, precision transducers represent a "mental map" that a model checker constructs to know the abstraction precision that is needed for different parts of the state space. Since an abstract precision can contain arbitrarily complex formulas, precision transducers are also viable for sharing correctness certificates (witnesses), which can reduce the validation effort considerably. 161

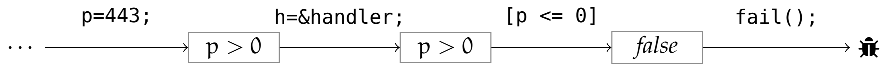
Model Checking

A software model checker constructs finite models of programs with a potentially infinitely large state space. A core problem in model checking is the choice of the level of abstraction—the abstraction precision—for the abstract model to construct from the concrete system. The resulting abstract model must be (1) sufficiently precise such that the absence of a specification violation can be proven without false alarms, but it also has to be (2) as abstract as possible to allow an analysis to converge and to come up with a finite model. Coming up with the right abstraction precision for constructing an abstract model for a given verification task effectively and efficiently is crucial for the performance of a model checker. Therefore, we consider the abstraction precision that a model checker has computed for a given verification task as a *precious intermediate verification result*, and propose to reuse it *within and among (different) verification runs*. Similar verification runs can lead to (many) similar reasoning problems such that sharing these intermediate results for reuse seems promising. 162

CEGAR

A widely used technique to refine the abstraction precision is counterexample-guided abstraction refinement (CEGAR) [67]. It operates *iteratively* by *step-wise refining* the abstraction precision of the model until the model is *sufficiently precise* for the reasoning task at hand. It uses *infeasible counterexamples* to identify additional facts to add to the model. Such infeasible counterexamples can be present in the abstract model only because the abstract model then lacks precision. The refinement iterations stop if the algorithm does not run into infeasible counterexamples anymore. Take the following path segment of a program's execution as an example: Only facts about the variable p have to be tracked, that is, the abstraction precision $\pi = \{p > 0\}$ is sufficient for proving the call fail() infeasible. 163

Example



164

Sharing and Reuse Workflow

In this chapter, we propose, evaluate, and discuss techniques for reducing the number of refinement iterations by sharing and reusing abstraction precisions while keeping the resulting abstract model as small and parsimonious as possible such that the fixed-point iteration of the model checking algorithm converges fast. We discuss how sharing and reusing of abstraction precisions can influence the size of the abstract state space, and the number of refinement iterations, and thus the performance of a verifier. Both the reuse of abstraction precisions within a verification run, but also the reuse among verification runs is motivated by commonalities between different verification (sub-)tasks and the corresponding reasoning problems. Similar solutions can be (re-)used for similar problems. Repeated computation effort can be avoided and thus, *resources*—such as time, energy, and money—saved. Our work on precision sharing and reuse is applied [36] for continuously checking *Linux kernel modules* for their adherence to a set of properties that specify the correct usage of the Linux kernel API [42, 159].

Please note that this chapter has a strong focus on the verification of *safety properties*. We focus the discussions on procedures that discover, share and reuse facts that are needed to prove the unreachability of bad states.

165 Safety Properties

Contributions. This chapter is inspired by paper on "Precision Reuse for Efficient Regression Verification" [36] but presents the following novel contributions:

- *Abstraction Precision.* We *formally* define a notion of an abstraction precision, and form a *lattice* thereof. We distinguish between the *actual* precision and a *candidate* precision and define the *scope* of an abstraction precision. Based on the notion of an *elementary precision*, we introduce the concept of *precision grinding*. We introduce *abstract precisions* to allow for differentiating the abstraction precision by concern.
- *Precision Transducers.* We present *precision transducers* as a flexible means for sharing abstraction precisions on precisely defined scopes: We operationalize *scoped model precisions* based on precision transducers—which is one form of an *abstract transducer*.
- *Sharing Strategies.* We demonstrate how both *existing and novel strategies* for sharing abstraction precisions for reuse can be realized based on precision transducers. A good strategy for sharing precisions aims to *balance* between a small number of precision refinement iterations and narrow precision scopes.
- *Precision Synthesis.* We demonstrate how *precision synthesis* can be implemented based on precision transducers. Parameterized precisions (*precision templates*) become instantiated based on the observations made while traversing the state space.
- *Empirical Study.* We conduct an empirical study to demonstrate the *applicability* and effects of different strategies for sharing abstraction precisions for reuse based on precision transducers. Our novel sharing strategy that takes advantage of the expressiveness of precision transducers yields promising results. Furthermore, we present first results for precision synthesis based on precision transducers.

Outline. We start with an elaborate characterization of the problem of discovering, sharing, and reusing abstraction precisions (Sect 5.1). Which approaches are there for discovering facts to track and to refine an abstraction precision? Do these techniques influence the potential of sharing and reuse?

Which considerations are relevant when sharing abstraction precisions? To which extent is the size of block-abstraction problems relevant for choosing an appropriate strategy for sharing and reuse? How is sharing different from reusing abstraction precisions? What are the appropriate points in time of a verification run for sharing and reuse? Section 5.2 formally defines different notions that are relevant when dealing with abstraction precisions, for example, an abstraction precision lattice, the equality of abstraction precisions, and elementary abstraction precisions. Furthermore, we define the scope of an abstraction precision and the notion of an abstract precision. Section 5.3 introduces precision transducers and a corresponding precision transducer analysis. Section 5.6 first introduces the notion of path precision transducers and tree precision transducers. Then, we describe how different strategies for sharing precisions for reuse can be built on top of precision transducers. Section 5.5 discusses how parameterized precision transducers can be used for synthesizing abstraction precisions. Section 5.6 continues with considerations that are relevant for the reuse of abstraction precisions. How can precision transducers be integrated into the verification workflow? How to choose from the candidates for reuse? Which filtering techniques are appropriate? What granularity can elements of abstraction precisions have? Section 5.7 presents an empirical study to illustrate the practical applicability of the concepts and techniques that we introduce in this chapter. The chapter finishes by summarizing this and related work.

5.1 PROBLEM CHARACTERIZATION

Several factors determine if the process of constructing an abstract model is well suited for efficiently and effectively conducting the verification task at hand. We *focus* on the influence of the *abstraction precision* and its *discovery*, *sharing*, and *reuse*. In the following, we take different conceptual perspectives to discuss and characterize arising problems. Based on these conceptual frameworks, we make more specific contributions throughout the chapter.

5.1.1 Block Abstraction

The idea of abstraction-based model checking [67, 70] is to model only those details of a system that are relevant for the given reasoning task. The abstraction process results in summaries that describe the relevant facts—as specified by the abstraction precision—for the reasoning task. We use the term block abstraction problems to refer to these summarization tasks:

Definition 76: Block Abstraction Problem

The task of computing an abstraction (summary) with a given abstraction precision of a specific fraction (block) of a program and its state space is called a *block abstraction problem*. A *block* defines a set of behaviors and concrete states of a program. Let $D = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ be the abstract domain with the lattice $\mathcal{E}$ of abstract states E. Formally, an abstraction problem $(e, \overline{O}, \pi) \in E \times 2^{Op^*} \times \Pi$ is a tuple that consists of an abstract *block entry state* $e \in E$, the set of *block traces* $\overline{O}$, which is a set of *finite* sequences of *program operations*, and the *candidate abstraction precision* $\pi \in \Pi$ to use. The result of solving this block abstraction problem is a set of abstract successor states $\widehat{e}' \subset E$, where each of them is the result of computing an abstraction $\langle\!\langle\cdot\rangle\!\rangle^{\pi}$ based on an abstraction precision π. We therefore also call these states *summary states* or *abstraction states*. The resulting abstraction must overapproximate the block:

$$\Big(\bigsqcup_{\overline{o} \in \overline{O}} SP(e, \overline{o})\Big) \sqsubseteq \bigsqcup \widehat{e}'.$$

The operator SP computes the strongest postcondition—see Sect.2.3.4.

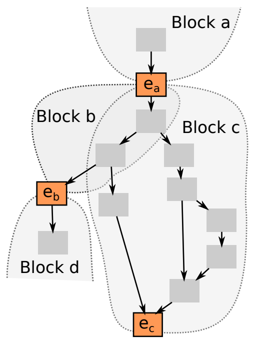


Figure 32: Block abstraction problems

166

Size of Abstraction Problems

Figure 32 provides an illustration of how block-abstraction problems can be [28, 35] reflected in an abstract reachability graph. The orange graph nodes represent abstraction states, the gray nodes represent (intermediate) abstract states that are computed after each control-flow transition, and for that no abstraction is computed. The set of sequences of control-flow transitions between the abstraction states correspond to the set of block traces. The smaller the *size* of block abstraction problems is, the more of them are there for a given verification task, and the more abstractions of these blocks have to be computed during the state-space exploration process.

In the implementation and formalization of the predicate analysis [35], on which this work builds, a block abstraction problem is encoded in a form that can be handed over to an SMT solver easily—see Sect. 2.4.4 for details of the

167 Predicate Block Abstraction Problem

analysis. A *predicate block abstraction problem* is a tuple $(\psi, \varphi, \pi) \in \mathcal{F} \times \mathcal{F} \times \Pi$, where ψ is the *abstraction formula* that represents the initial abstract state of the block abstraction problem, the *block formula* φ represents the semantics of the set of program operation sequences in the block, and π is the candidate abstraction precision for computing the summary.

Abstraction Computation. The predicate analysis [35] summarizes all relevant information from a block abstraction problem by computing a predicate abstraction [122, 177], based on the set of predicates from the candidate abstraction precision π. Each of the resulting abstraction states has an abstraction formula $\psi' \in \mathcal{F}$, which represents a set $[\![\psi']\!]$ of concrete states that can be reached by following sequences of program operation $\overline{o} \in \overline{O}$ starting from one of the concrete states $[\![e]\!]$ that the block entry state e represents. 168 ⚠

Whenever we talk about (block) abstraction problems in the sections that follow, we talk about predicate block abstraction problems and deal with abstraction precisions that consist of *sets of predicates*. A block encoding strategy [28, 35] determines the fraction of a program and its state space that should be encoded into one block. If not stated otherwise, we assume that Boolean predicate abstraction [177] is used to compute abstractions. Boolean predicate abstraction provides the most precise abstraction that is possible based on a given set of predicates. Nevertheless, its costs are high:

Problem 1 (High Abstraction Costs). The costs of computing a Boolean predicate abstraction $\langle\!\langle \vartheta \rangle\!\rangle_{\mathbb{B}}^{\pi}$ of a given formula ϑ (with decidable theories) are double *exponential* in the number of predicates [120, 177] in the precision π.

Abstraction Coverage. Whenever we want to prove a temporal property of a given program, with potentially infinitely many states and behaviors, we have to construct a finite model first.

In unbounded model checking, we create a finite model by employing abstraction [70], which results in an overapproximation that covers all possible states and behaviors. This overapproximation is constructed by an algorithm

169 Fixed-Point Iteration

that conducts a fixed-point iteration: Whenever the algorithm reaches a new state, it checks if this state is covered by the model that has been constructed up to that point. The algorithm continues its state-space exploration from newly discovered states until no new unknown states are discovered. One example of such an algorithm is the CPA algorithm [31]—see Sect. 2.5.1. The check for coverage is typically performed based on the inclusion relation $\sqsubseteq$ of the lattice of abstract states—which is central for the fixed-point iteration.

170 Block Summary Coverage

As motivated previously, block abstraction problems are solved to summarize fractions of a program and its state space. This is a crucial step for unbounded software model checking, for example, to cope with loops and recursion: Loop-free subgraphs (for example, loop bodies) of the control-flow automaton are summarized [28] into abstraction states, and then, coverage is checked between the abstraction states that summarize the subset of the state space that has already been explored.

Example 13. Figure 33 illustrates a reachability graph that is formed of *abstraction* states (the nodes of the graph) *only*; each abstraction state was computed by solving a block-abstraction problem. In this example, abstraction state e_1 is the initial state, and abstraction state e_2 is the result of solving the block abstraction problem $(e_1, \overline{O}_2, \pi)$. Let the set $\overline{O}_3$ be the set of sequences of program operations that represent a loop body, and let the set $\overline{O}_2$ be the sequences of program operations reach this loop body. Since we assume that the summaries that are represented by the abstraction states e_2 and e_3 are loop invariant, coverage is achieved, that is, $e_3 \sqsubseteq e_2$.

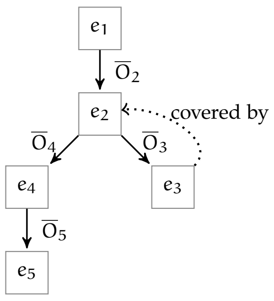


Figure 33: Reachability graph of block abstractions

The information to encode into a summary is determined by the abstraction precision π of a block-abstraction problem. Having an appropriate abstraction precision—for example, a set of predicates that can be combined by predicate abstraction to form a loop invariant—determines the efficiency and effectiveness of a verification procedure.

171 *Role of the Abstraction Precision*

We prefer constructing models based on abstraction precisions for that the analysis can *converge* to a fixed point *fast*. In the case of predicate abstraction, general predicates increase the chance of ending up in such abstractions. Having too specific predicates in the abstraction precision reduces this chance considerably: A Boolean predicate abstraction $\langle\langle\vartheta\rangle\rangle_{\mathbb{B}}^{\pi}$ of a formula ϑ is the strongest Boolean combination of a set of predicates π that is entailed by ϑ—see Sect. 2.4.4. One of its properties is that it does not have any notion of generalization, that is, it takes *all given* predicates into account and does not manipulate any of them, for example, weaken, strengthen, or remove some of them. This characteristic is critical for precision sharing and reuse: Having, without reason, too narrow predicates in the precision can contribute to a diverging state space.

Example 14. Given a formula $\vartheta \equiv (i_1 = 1) \wedge (i_2 = i_1 + 1)$ to compute the Boolean predicate abstraction $\langle\langle\vartheta\rangle\rangle_{\mathbb{B}}^{\pi}$ with precision $\pi = \{i <= 16, i = 1, i = 2\}$. Each predicate is represented by a propositional variable $\{\rho_1, \ldots, \rho_n\}$ based on that we construct the AllSat problem $\vartheta \wedge (\rho_1 \Leftrightarrow i \leqslant 16) \wedge (\rho_2 \Leftrightarrow i = 1) \wedge (\rho_3 \Leftrightarrow i = 2)$ that is used for computing the abstraction. In the end, we get $\langle\langle\vartheta\rangle\rangle_{\mathbb{B}}^{\pi} \equiv \rho_1 \wedge \rho_3 \equiv (i = 2) \wedge (i \leqslant 16)$, which can be further simplified to $i = 2$. That is, while there would have been a general predicate that could have acted as a loop invariant, the abstraction procedure returned an abstraction with the most precise predicate.

We can see from the previous example that a high precision is not always better—the highest possible precision $\pi_{\top}$ would not cause any abstraction and result in the formula ϑ—given that ϑ is fully satisfiable and does not contain unsatisfiable conjunctive clauses.

Problem 2 (Overspecific Predicates). A too high abstraction precision, for example, a set of fine-grained predicates that maps to a small set of concrete states, can reduce the performance of a verification procedure considerably by reducing the chance of arriving on a fixed point early.

172 *Implications for Sharing and Reuse*

The implications for sharing and reusing predicates as elements of abstraction precisions can be manifold. For example, prefer to share general predicates, which are more likely to lead to a converging state-space exploration process, reduce the scope of less general predicates, and take strategies into account that generalize predicates before actually (re-)using them.

5.1.2 Discover-Share-Reuse

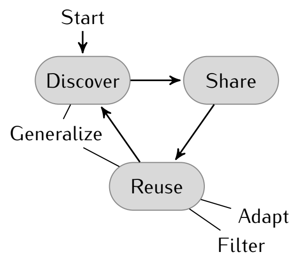


Figure 34: Discover-Share-Reuse (DSR) Scheme

After we have discussed how an abstraction is computed based on a given abstraction precision—for example, a set of predicates—we now focus on how these abstraction precisions are discovered, shared, and reused. We introduce the *Discover-Share-Reuse Scheme* as one perspective on abstraction-based model checking to deduce several principles for precision sharing and reuse of abstraction precisions. Figure 34 illustrates the scheme.

Discover. The phase *discover* is used to identify additional facts to model about a program to conduct the given verification task. Verification techniques that rely on abstraction are dependent on procedures that discover the facts to model for (1) ruling out spurious (infeasible) counterexamples, and (2) converging to a fixed point in the analysis process fast (and not to diverge), for example, by discovering appropriate loop invariants. A procedure for discovering the facts to track is said to be *complete* [152] if it guarantees to eventually provide a refined abstraction precision that is sufficiently precise to conduct a given verification task. The most primitive but complete approach for discovering predicates is [152] to enumerate all predicates (a countable set) on the parts—for example, data locations or control locations—of a task and refine the abstraction precision by adding one more predicate from the enumeration in each iteration.

173 *Completeness of Discovery*

174 *Approaches for Discovery*

Different approaches (many incomplete ones) have been proposed to discover the facts to track and are available for use: An abstraction precision can be provided manually [122] by the verification engineer. Candidates can be derived from branching conditions [59], infeasible program paths [51], program path slices [150], counterexamples [67], from matching templates [153, 238], from the cone-of-influence [148], or from weakest preconditions or strongest postconditions [25]. Some discovery techniques are only applicable to constraint systems that are either unsatisfiable or satisfiable, whereas this is not relevant for other techniques. Techniques based on Craig interpolation [190] and unsat cores [132, 183] operate on unsatisfiable problems and have a huge impact on the verification community. Abduction and the weakest precondition calculus [59, 183, 231] are applied to satisfiable problems to identify relevant details or even loop invariants. Other fields of research also studied the discovery of predicates: *Predicate invention* from the field of inductive logic programming denotes the automatic process of introducing new relationships (predicates) [198]. The fields of *inductive inference* and *logic of generality* provide additional (or auxiliary) directions to come up with relational predicates [225]. The refinement procedure might include an *acceleration* [27] step that generalizes the predicates to track.

175 *Focus on CEGAR with Craig Interpolation*

This work focuses on sharing and reusing abstraction precisions that are discovered based on counterexample-guided abstraction refinement [67] with Craig interpolation [84, 137, 190, 191], and are used for predicate abstraction [28, 35, 122]. We address the following problem:

Problem 3 (Number of Refinement Iterations). Several refinement iterations are necessary until a CEGAR-based verifier has constructed an abstract model with all the facts that are sufficient for conducting the verification task at hand. The number of iterations increases with the number of facts to model.

Share. Before abstraction precisions can be reused, candidates for reuse have to be shared to the different block abstraction problems in a verification run. An abstraction precision that either helped to solve a verification task efficiently and effectively, or helps likely to increase the performance of a verification run, is a precious intermediate verification result that should be shared for reuse.

176 *Sharing*

We use the term *sharing* to denote the decision and act, whether or not to provide an abstraction precision for a block abstraction problem in a specific context. The same abstraction precision can be provided for multiple block abstraction problems, within or among verification runs, and might be relevant to rule out several infeasible counterexamples at different points of the state space. The *context* for sharing an abstraction precision can be, for example, a specific unwinding of a loop, a sequence of control-flow transitions, or a specific function call stack. A *sharing strategy* is a systematic approach to share abstraction precision for reuse and might be tightly integrated with the precision discovery procedure.

177 *Sharing Strategy*

Different sharing strategies have been proposed in the literature. Sharing abstraction precisions *as local as possible* [137] helps to mitigate the state space explosion problem, whereas sharing the same abstraction precision at all points of the state space *reduces the number of refinement iterations*, but may increase both the size of the abstract model (coverage less likely) and the costs for abstraction computations [177]. Lazy abstraction [136] maps an abstraction precision to each abstract state; it aims at a parsimonious abstract state space [137] by reducing sharing and making abstraction precisions as local as possible. Other strategies [147] choose to use flat maps of functions or control locations to abstraction precisions, or sets of globally shared predicates, and can keep the sharing information among iterations, but cannot express complex contexts for sharing.

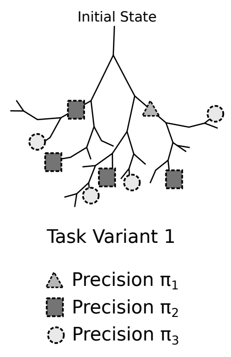


Figure 35: Different abstraction precisions among the ARG

Example 15. A refiner that follows the pure idea of lazy abstraction [136] keeps the refinements of abstraction precisions local to the states that have no reachable counterparts in the real system (the last state in a sequence that has reachable counterparts is called the *pivot states*), and would not share the precision among different abstract states that are not part of the affected subgraph (below the pivot states). Take, for example, task variant 1 in Fig. 35: The abstraction precision π_2 would be discovered four times because the pivot states are in different subgraphs of the ARG. In total, at least 9 refinements would have to take place to identify only three different sets of facts—reflected in the abstraction precisions π_1, π_2, and π_3—to track.

Problem 4 (Generic Sharing Mechanism Missing). There is no generic mechanism that allows expressing different strategies for sharing abstraction precisions for reuse. Existing approaches either allow only a flat mapping and are rather insensitive to the context or rely (as lazy abstraction) on a copy of the state space to not lose the context after an abstraction refinement.

This chapter introduces precision transducers that allow expressing different sharing strategies, for example, strategies that are close to the sweet spot between a low number of refinement iterations and a compact abstract state

```
int n, m, i, j; // non det.
int k = 0;
assume(10 <= n && 10 <= m);
for (i = 0; i < n; i++)
  for (j = 0; j < m; j++)
    k ++;
assert(k >= 100);
```

Figure 36: Context

space. Precision transducers provide expressiveness to share precisions depending on the (call) context, the program flow of the program analysis, or even on specific program paths, in a path-sensitive manner.

Example 16. Figure 36 shows a program with nested loops, for which considering the context is crucial when sharing predicates to obtain the verification result efficiently: Considering the context reduces the time for verification from 380 s to 30 s, and a model with 957 states instead of 39 117. See the appendix for details on our benchmarking environment.

178

Filter, Generalize

Reuse. The reuse of abstraction precisions can take place after they have been shared. Different abstraction precisions can be shared as candidates for reuse. Further processing steps, for example, filtering or generalization, might be applied to these candidate precisions. These steps can be necessary to keep the abstraction costs low and to ensure the effectiveness and efficiency of the verification process. For example, an abstraction procedure based on Boolean predicate abstraction [28, 122] might [72] take only a subset of the shared predicates into account to compute an abstraction. The set of predicates in the candidate precision might be large, and many of these predicates could be removed while still being able to achieve the same or a better verification performance; it is not always good to have the most precise abstraction [234].

The abstraction procedure can decide to synthesize more general abstraction precisions from the candidate precisions. The generalized precision must exclude at least the same (infeasible) counterexamples than the less general precision to ensure *progress* of the verification procedure. Alternatively, the procedure must provide a mechanism to detect that an abstraction was too general and backtrack [97] to a less general abstraction precision. A classification [7] of variables can be taken into account to choose an appropriate generalization (or minimization) procedure.

Problem 5 (Several Candidates for Reuse). Using the full abstraction precision that has been shared as a candidate for computing an abstraction can lead to an abstract model that diverges, or it can increase the cost for computing the abstraction unnecessarily.

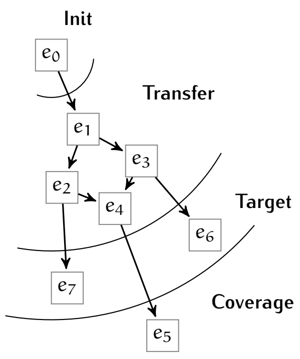


Figure 37: Init-Transfer-Target-Coverage Scheme, Forwards Analysis

5.1.3 Init-Transfer-Target-Coverage

Sharing, reuse, and the discovery of abstraction precisions can take place in different phases of a verification run. We introduce the *Init-Transfer-Target-Coverage Scheme* (ITTC) as a model for categorizing, explaining, and discussing different model checking techniques. We use this scheme to discuss how and where discovery, sharing, and reuse of abstraction precisions can be integrated into a verification tool. Figure 37 illustrates the schema. Please note that our discussion of the scheme focuses on a forwards analysis, but the scheme can also be adopted to backward analyses. The scheme models four phases of a model construction algorithm:

Init. The *init phase* covers the time before the first successor state for the initial state is computed. An *initial abstraction precision* might be provided or discovered in this phase, for example, from the verification engineer, or a mining heuristic that is independent of the actual state-space traversal. We use this phase [36] to provide an initial precision (transducer) for regression verification, that is, to share the abstraction precision that was used for a previous version of the verification task.

In the init phase of a verification run, we can use (1) information that is provided by the user as an argument to the verifier, or (2) information that the verifier has learned from previous verification runs and provided in persistent storage. Let us assume that there is already a history of verification runs for which the precisions have been stored. We can define a similarity (or distance) measure between different verification tasks. We use this measure to identify the task with the highest *similarity to the verification task* at hand and take its abstraction precision for the task at hand as the *initial* value.

179 Task Similarity

An obvious distance measure can be defined based on information from the version-control systems: We can reuse the union of precisions that were stored for the versions that precede directly in the graph of commits, that is, the distance is defined based on the number of edges between two versions in the commit-graph. We take this approach for regression verification [36].

Transfer. The *transfer phase* covers all abstraction (summary) computations and the exploration of (abstract) successor states. This phase is central to the concept of dynamic precision adjustment [32] where the abstraction precision can get adjusted during the state space traversal. The transfer phase might conduct further widenings or strengthenings [31] of abstract states, that is, adjust the abstraction precision of abstract states and with it the abstraction precision of the model.

180 Dynamic Precision

Target. The *target phase* covers all points in time in which a target state, which represents a potential violation of the specification, is entered. The classical counterexample-guided abstraction refinement [67] operates in this phase to check if the target state is reachable, and to refine the abstraction precision in case of spurious counterexamples. The analysis is typically resumed after the abstraction precision has been refined and the infeasible part of the state space removed.

Coverage. Specific techniques [59] refine the abstraction precision after the model checking algorithm has reached a fixed point by *covering* all potential states of the system under analysis. This class of techniques performs a *global refinement*, that is, the set of all reached target states, where each of them represents a potential violation of the specification, is considered for choosing an appropriate refinement of the abstraction precisions for the next iteration of the verification algorithm. Especially techniques for multi-property verification [8] and model-driven test generation might benefit from such a refinement approach.

181 Global Refinement

5.2 ABSTRACTION PRECISION

Before we continue our discussions on sharing abstraction precisions for reuse, we formally define our notion of abstraction precision and introduce further notions that are built around the concept of abstraction precision.

5.2.1 Types of Abstraction Precisions

Precision is an established concept in program analysis and verification [32, 67, 89, 119, 136]. We provide a formal definition that reflects our perspective on abstraction precisions:

Definition 77: Abstraction Precision

An *abstraction precision* (or *concrete precision*) $\pi \in \Pi$ characterizes the set of details of a state or system that should be maintained by an abstraction process. The abstraction precision denotes $[\![\pi]\!] \subseteq C \to 2^C$ a function that defines the set of states that are considered equivalent with respect to the fraction of information to maintain. It defines which information from a concrete state is maintained in a corresponding abstract state. This corresponds to an existential abstraction [70] and either maintains the full information or leads to an overapproximation. The set of all abstraction precisions is denoted by Π.

That is, we take a similar perspective as Nayak and Levy [202] that view abstractions as *model level mappings*. They define a function $\pi : [\![B]\!] \to [\![A]\!]$, where $[\![B]\!]$ is the set of interpretations of words in a language "base" and set $[\![A]\!]$ is the set of interpretations of words in a language "abstract". A prominent notion of precision was described by Dams, Gerth, and Grumberg [89]. They use the term *precision* to describe the *information content* of an abstract model: An entity a is said to contain more information than an entity b if their interpretations are in a subset relation, that is, iff $[\![b]\!] \subseteq [\![a]\!]$.

182 Related Work

From the definition of an abstraction precision follows the definition of the equality of two given abstraction precisions:

Definition 78: Precision Equivalence

Two abstraction precisions π_1 and π_2 are said to be *equivalent*, that is, $\pi_1 \equiv \pi_2$, if and only if they result in the same abstraction for all given inputs. That is, for each state $c \in C$ it is true that $[\![\pi_1]\!](c) = [\![\pi_2]\!](c)$.

Example 17. Let's assume that the set of concrete states $C \subseteq 2^{X \to \mathbb{B}}$ is defined as a set of maps that map from data locations to bit values. Furthermore, assume that the set of data locations X consists of the set of program variables $\{a, b, c\}$ only. The precision π_1 defines that the value of data location a and b should be maintained. For a concrete state $c_1 = \{a : 0, b : 1, c : 0\}$, precision π_1 defines the abstraction $[\![\pi_1]\!](c_1) = \{\{a : 0, b : 1, c : 0\}, \{a : 0, b : 1, c : 1\}\}$, that is, it results in an overapproximation that assumes any possible value for data location c.

Previous example makes use of a type of abstraction precision that defines the information to maintain based on the set of data locations; we call such a precision a data-location precision:

Definition 79: Data Location Precision

A *data-location precision* $\pi \in \Pi_X$ defines a set of data locations for which an abstraction procedure should maintain the full information, that is, $\pi \subseteq X$, and $[\![\pi]\!] \subseteq C \to 2^C$. The set of all data-location precisions is denoted by Π_X.

Another important type of precision is the predicate precision:

Definition 80: Predicate Precision

A *predicate precision* $\pi \in \Pi_{\mathcal{P}}$ is an abstraction precision that is formed as a set of predicates $\mathcal{P}$, that is, $\pi \subseteq \mathcal{P}$ and $[\![\pi]\!] \subseteq C \to 2^C$. The set of all predicate precisions is denoted by $\Pi_{\mathcal{P}}$. One predicate $\rho : C \to \mathbb{B}$ is a function on the set of concrete states C and denotes a subset $[\![\rho]\!] \subseteq C$ of those—it is a Boolean formula in predicate logic.
Semantically, a predicate precision $\pi \in \Pi_{\mathcal{P}}$ defines an abstraction of a given concrete state $c \in C$ as the *conjunction* of all predicates from the set π that evaluate to *true* for the state, that is, $[\![\pi]\!] = \{(c, \bigwedge\{\rho \in \pi : \rho(c) = \mathit{true}\}) \mid c \in C\}$.

Note that also a set of program invariants can form a predicate precision but *predicates from a predicate precision are not necessarily program invariants*.

Example 18. Given the predicate precisions $\pi_1 = \{b = 1, x = 2, i \leqslant 90\}$, and $\pi_2 = \{x = 2\}$, which have been used in Sect. 2.4.4 to discuss the difference between Boolean and Cartesian predicate abstraction. The precisions are *not* equal to another, that is, $\pi_1 \neq \pi_2$, because there exists at least *one* concrete state $c \in C$ for that $[\![\pi_1]\!](c) \neq [\![\pi_2]\!](c)$. An example for such a state is $c_x = \{x : 2, b : 1, i : 0\}$. If we assume that all concrete values are in the interval $[-128, 127]$, then $\{x : 2, b : 1, i : 117\} \in [\![\pi_2]\!](c_x)$ but $\{x : 2, b : 1, i : 117\} \notin [\![\pi_1]\!](c_x)$.

We distinguish between candidate precisions and actual precisions:

Definition 81: Candidate and Actual Precision

A *candidate precision* $\pi_c \in \Pi$ is shared to an abstraction procedure for computing an abstraction. The *actual precision* $\pi_a \in \Pi$ is the abstraction precision that the resulting abstraction actually has. A discrepancy between candidate precision and actual precision can be the result of adapting, filtering, or generalizing the given candidate precision.

5.2.2 Concrete Precision Lattice

Abstraction precisions form a lattice, which is important to ensure progress of refinement-based verification techniques:

Definition 82: Concrete Precision Lattice

The *concrete precision lattice* is defined by tuple $\sqcap = (\Pi, \sqsubseteq_\pi, \sqcap_\pi, \sqcup_\pi, \top_\pi, \bot_\pi)$. A pair of abstraction precisions is in the *inclusion relation* $\sqsubseteq_\pi \subseteq \Pi \times \Pi$, that is, $(\pi_1, \pi_2) \in \sqsubseteq_\pi$, if and only if $\forall c \in C : [\![\pi_1]\!](c) \supseteq [\![\pi_2]\!](c)$. Let $c \in C$ be an arbitrary concrete state. The *join* $\sqcup_\pi : \Pi \times \Pi \to \Pi$ of two abstraction precisions $\pi_1 \sqcup_\pi \pi_2$ defines a new abstraction precision $\pi_\sqcup$ that aims to maintain at least the information that is intended to be maintained by precision π_1 or precision π_2, that is, $[\![\pi_\sqcup]\!](c) = [\![\pi_1]\!](c) \cap [\![\pi_2]\!](c)$. The *meet* $\sqcap_\pi : \Pi \times \Pi \to \Pi$ of two abstraction precision $\pi_1 \sqcap_\pi \pi_2$ defines a new abstraction precision $\pi_\sqcap$ that aims to maintain the information that is intended to be maintained by both precision π_1 and π_2, that is, it is the union $[\![\pi_\sqcap]\!](c) = [\![\pi_1]\!](c) \cup [\![\pi_2]\!](c)$—less information is maintained, one state maps to a larger set of states. The *top precision* $\top_\pi$ defines that the full information should be maintained: $[\![\top_\pi]\!](c) = \{c\}$. The *bottom precision* $\bot_\pi$ defines that no information should be maintained: $[\![\bot_\pi]\!](c) = C$. The bottom precision is the *neutral precision* regarding the join of precisions. A *precision subtraction* $\pi_1 \setminus_\pi \pi_2$ defines a new abstraction precision $\pi_\setminus$ that aims at keeping all information that is specified in precision π_1 except if it is specified by π_2. The *complement* $\neg\pi$ of a given abstraction precision π defines the information *not* to maintain, that is, $[\![\neg\pi]\!](c) = [\![\pi_\top]\!](c) \setminus [\![\pi]\!](c)$.

Please note that lattices of predicate precisions and data-location precisions are typically implemented as *powerset lattices*.

183 ⚠ We assume that a concrete precision lattice $\sqcap$ is one component of every abstract domain with widening, that is, an abstract domain with widening is defined by the tuple $D = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle, \sqcap, \langle\!\langle\cdot\rangle\!\rangle^\pi)$. Whenever an abstract domain is composed to form a product domain, or composite domain [31], also a product of the precision lattice is constructed to form a product lattice. Along with the work that introduced dynamic precision adjustment [32], the notion of composite precision was introduced, but no lattice has been used to describe the relationship between precisions.

Example 19. Given two concrete predicate precisions $\pi_1 = \{a > 7\}$ and $\pi_2 = \{a < 23\}$. A new precision $\pi_3 = \pi_1 \sqcup_\pi \pi_2$ is the result of their join. The resulting precision $\pi_3 = \{a > 7, a < 23\}$ maintains more information, that is, the join denotes the intersection $[\![\pi_3]\!](c) = [\![\pi_1]\!](c) \cap [\![\pi_2]\!](c)$.

The following example clarifies the difference between a lattice of predicates—where an element in the lattice is a predicate—and our concrete precision lattice:

Example 20. Given two predicate precisions $\pi_1 = \{a > 7, a < 23\}$ and $\pi_2 = \{a > 7 \wedge a < 23\}$. Precisions π_1 and π_2 are *not equivalent* to another:

There exists at least one concrete state for which these precisions result in different abstractions. An example is the state $c = \{a : 1\}$ for which $\pi_2(c) = C$, whereas $\pi_1(c) = [\![a < 23]\!]$.

After we have defined the notion of precision lattice, we can define the notion of elementary precision:

Definition 83: Elementary Precision

A concrete abstraction precision π_e is called *elementary* if there is no other concrete abstraction precision π_l that is in the inclusion relation $\pi_l \sqsubseteq \pi_e$ but the bottom precision $\bot_\pi$. We use the symbol $\Pi_\triangle$ to denote the set of *non-elementary precisions*, which we define as $\Pi_\triangle = \{\pi \,|\, \pi \in \Pi \wedge \pi_l \sqsubseteq \pi \wedge \pi_l \neq \bot_\pi\}$. The symbol $\Pi_\square$ denotes all *elementary precisions*, that is, $\Pi_\square = \Pi \setminus \Pi_\triangle$.

Example 21. Let $\Pi_\mathcal{P}$ be the set-based predicate precisions (which are elements of a corresponding powerset lattice), that is, each concrete abstraction precision $\pi \in \Pi_\mathcal{P}$ is a set $\pi \subseteq \mathcal{P}$ of predicates, with $\pi_1 \sqsubseteq_\pi \pi_2 \Rightarrow \pi_1 \subseteq \pi_2$. Each predicate precisions $\pi \in \Pi_\mathcal{P}$ that consists of exactly one predicate, that is, $|\pi| = 1$, is an elementary precision.

5.2.3 Precision Scope

Precision sharing denotes the *process* of providing one specific abstraction precision for different abstraction tasks, which *results* in the scope of an abstraction precision. This type of sharing reduces [36] the effort for conducting precision refinements [67] to exclude spurious counterexamples. That is, sharing an abstraction precision then influences the complexity of the abstraction problems that have to be solved to conduct a verification task.

Definition 84: Precision Scope

The *scope* of an abstraction precision $\pi \in \Pi$ defines the context and the extent of sharing the abstraction precision for reuse. The abstraction precision is shared if the context is satisfied and as long the analysis stays in the extent of the scope. Formally, the scope scp_π of a given abstraction precision π denotes $[\![\mathrm{scp}_\pi]\!] \subseteq Op^* \times 2^{Op^*}$ a set of pairs of context and lookahead. That is, the precision π is shared if $(\bar{\sigma}, \widehat{\sigma}) \in \mathrm{scp}_\pi$ and the analysis reaches the end of word $\bar{\sigma}$ and the remaining postfix ahead to process is covered by the lookahead $\widehat{\sigma}$—see Sect. 3.2.1 for details on the lookahead.

We also define the precision of the full abstract model based on scopes:

Definition 85: Scoped Model Precision

A *scoped model precision* $\mathring{\pi}$ denotes $[\![\mathring{\pi}]\!] \subseteq Op^* \times 2^{Op^*} \times (H \to 2^\Pi)$ a mapping of particular abstraction precisions to different scopes, separated by concern. A concern $h \in H$ can be, for example, a property to check in a verification run.

Proposition 14. A scoped model precision has equivalent expressiveness than mapping abstraction precisions to abstract states if the abstract reachability graph is deterministic. An abstract reachability graph is called deterministic if there are not two more abstract successor states $E' \subseteq E$ for a given abstract state e with the same labeling, that is, if for any program operation op $\in$ *Op* we have $|\{e' \mid (e, e') \in \overset{\text{op}}{\rightsquigarrow}\}| \leqslant 1$.

Proof. Mapping the abstraction precision to abstract states would be more expressive in case of determinism if two different abstract states in the abstract reachability graph would be reachable based on the same sequence of program operations but this contradicts to the definition of deterministic reachability graphs. □

5.2.4 Abstract Precision

After we have discussed the notion of *concrete abstraction precision* in the last sub-sections, we now introduce the notion of *abstract* precision, which aids in following objectives:

- *Clear mapping* of different abstraction precisions to different concerns to analyze: This is helpful in case the state space is separated by concern to analyze at certain points. While intended to be used in future work, already existing state-space abstraction approaches that conduct some sort of state-space partitioning, for example, for inferring weakest preconditions [230, 231], could benefit.
- *Possibility to disable* the modeling of certain concerns: We presented the idea of dynamically disabling the analysis for particular concerns in our work on multi-property verification [8]. Mapping the empty set of concrete abstraction precisions to a concern disables its analysis.
- *Alternatives* for concrete precisions to choose from can be provided: This is a result of mapping a *set* of precisions to a concern. For this work, we assume that at most one abstraction precision is mapped to a concern and keep alternative precisions for future work.

Based on these objectives, we define an abstract precision as follows:

Definition 86: Abstract Precision

A *abstract precision* $\mathfrak{p} \in \mathfrak{P}$ maps a set of concrete abstraction precisions to each concern of a program analysis task, that is, it is a function $\mathfrak{p} : H \to 2^{\Pi}$. The set of all abstract precisions is denoted by $\mathfrak{P}$.

As we proceed in this section, we will define several operators on abstract precisions. The formalisms that we use here to deal with (map-)lattices and their elements are defined in Sect. 37.

Example 22. Given the set $H_{\overline{P}} = \{h_1, h_2, h_3, h_4, h_5\}$ of concerns that are relevant for a given analysis task, and let each concern $h \in H_{\overline{P}} \subset H$ be a property to check (a specification is a set of properties). The abstract precision $\mathfrak{p} = \{(h_1, \{\pi_1\}), (h_3, \bot_P), (h_4, \{\bot_\pi\})\}$ states that concern h_1 should be tracked with precision π_1, concern h_2 should not be tracked because the bottom element $\bot_P$ is implicitly mapped to it (see the definition of a map lattice), concern h_3 is also disabled, concern h_4 is enabled to be tracked with the lowest possible precision $\bot_\pi$, and the tracking of facts for concern h_5 is also disabled.

We continue with the definition of a lattice of abstract precisions and continue with its join and meet:

Definition 87: Abstract Precision Join

The *join* $\sqcup_\mathfrak{p} : \mathfrak{P} \times \mathfrak{P} \to \mathfrak{P}$ of a pair of abstract precisions is the *pairwise join* of the represented concrete precisions, separated by concern. The pairwise join of concrete precision sets is defined by $\times_\sqcup(S_1, S_2) = \{s_1 \sqcup_\pi s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}$. That is, the concatenation of a pair $\mathfrak{p}_1, \mathfrak{p}_2$ of abstract precisions is defined as

$$\sqcup_\mathfrak{p}(\mathfrak{p}_1, \mathfrak{p}_2) = \{(h, \mathfrak{p}_1(h)_\bot \times_\sqcup \mathfrak{p}_2(h)_\bot) \mid h \in \mathsf{keys}(\mathfrak{p}_1) \cup \mathsf{keys}(\mathfrak{p}_2)\}.$$

Example 23. Given the abstract precisions $\mathfrak{p}_1 = \{(h_1, \{\pi_1\}), (h_2, \{\pi_2\}), (h_4, \{\pi_4, \pi_5\})\}$ and $\mathfrak{p}_2 = \{(h_2, \{\pi_7\}), (h_3, \{\pi_3\}), (h_4, \{\pi_6\})\}$. The join $\mathfrak{p}_1 \sqcup_\mathfrak{p} \mathfrak{p}_2$ results in the abstract precision $\mathfrak{p}_3 = \{(h_1, \{\pi_1\}), (h_2, \{\pi_2 \sqcup_\pi \pi_7\}), (h_3, \{\pi_3\}), (h_4, \{\pi_4 \sqcup_\pi \pi_6, \pi_5 \sqcup_\pi \pi_6\})\}$.

Definition 88: Abstract Precision Meet

The *meet* $\sqcap_\mathfrak{p} : \mathfrak{P} \times \mathfrak{P} \to \mathfrak{P}$ of a pair of abstract precisions is the *pairwise meet* of the represented concrete precisions, separated by concern. The pairwise meet of concrete precision sets is defined by $\times_\sqcap(S_1, S_2) = \{s_1 \sqcap_\pi s_2 \mid s_1 \in S_1 \wedge s_2 \in S_2\}$. That is, the meet of a pair $\mathfrak{p}_1, \mathfrak{p}_2$ of abstract precisions is defined as

$$\sqcap_\mathfrak{p}(\mathfrak{p}_1, \mathfrak{p}_2) = \{(h, \mathfrak{p}_1(h)_\bot \times_\sqcap \mathfrak{p}_2(h)_\bot) \mid h \in \mathsf{keys}(\mathfrak{p}_1) \cup \mathsf{keys}(\mathfrak{p}_2)\}.$$

The abstract precisions form a lattice:

Definition 89: Abstract Precision Lattice

The *abstract precision lattice* is a lattice $\dddot{\mathfrak{P}} = \mathsf{ml}(H, \mathsf{pw}(\Pi)) = (\mathfrak{P}, \sqsubseteq_\mathfrak{p}, \sqcap_\mathfrak{p}, \sqcup_\mathfrak{p}, \top_\mathfrak{p}, \bot_\mathfrak{p})$ over the set of abstract precisions. That is, each element of this lattice is a map from concerns to sets of concrete abstraction precisions. The *abstract top precision* $\top_\mathfrak{p}$ signals to track all concerns with full precision. The *abstract epsilon transition* $\mathfrak{p}_\epsilon$ signals that all concerns are enabled for tracking but with the lowest possible precision, that is, $[\![\mathfrak{p}_\epsilon]\!] = \{(h, \{\bot_\pi\}) \mid h \in H\}$. The *abstract bottom precision* $\bot_\mathfrak{p}$ signals that no concern is enabled for tracking. The definitions of the inclusion relation $\sqsubseteq_\mathfrak{p}$ correspond to the generic definitions for the map lattice—see Sect. 37.

5.3 PRECISION TRANSDUCER

This section presents precision transducers as means to share abstraction precisions for reuse. Generally, all those finite-state machines that emit abstraction precisions on states or transitions can be considered to be precision transducers. A precision transducer is an operationalization of a scoped model precision $\mathring{\pi}$. We define a precision transducer as follows:

Definition 90: Precision Transducer

A *precision transducer* (or *π-transducer*) is a transducer that emits abstraction precisions along given sequences of program operations. The emitted abstraction precisions define the details to model at the different points of the abstract state space. Precision transducers are a means to define the *scope* of abstraction precisions. We instantiate an abstract transducer to work as a precision transducer and get the tuple

$$P = (Q, D_{\mathrm{in}}, D_{\mathrm{out}}, \iota_0, F, \delta).$$

The definitions of the set of control states Q, the initial transducer state ι_0, the set of final control states F, and the transition relation δ, are equivalent to those of the generic abstract transducer—see Sect. 3.2. The word domains for input and output are defined to provide the functionality of precision transducers. An abstract input word maps to a set of sequences of control-flow transitions, and an abstract output word maps to a set of sequences of abstraction precisions:

- Input Word Domain D_{in}. The *abstract input domain* is an abstract word domain $D_{\mathrm{in}} = (\mathsf{fl}(G^*), \dddot{\mathfrak{I}}, [\![\cdot]\!]_{\mathrm{in}}, \langle\!\langle\cdot\rangle\!\rangle_{\mathrm{in}})$, with the lattice $\dddot{\mathfrak{I}}$ of abstract input words. One *abstract input word* denotes a set of finite sequences of control-flow transitions, which is reflected by the semantic denotation function $[\![\cdot]\!]_{\mathrm{in}} : \mathfrak{I} \to 2^{G^*}$. The abstraction function $\langle\!\langle\cdot\rangle\!\rangle_{\mathrm{in}} : 2^{G^*} \to \mathfrak{I}$ provides the inverse mapping.
- Output Word Domain D_{out}. The *output word domain* $D_{\mathrm{out}} = (\mathsf{pr}(\Pi^\infty), \dddot{\mathfrak{P}}, [\![\cdot]\!]_{\mathrm{out}}, \langle\!\langle\cdot\rangle\!\rangle_{\mathrm{out}})$, or also *precision word domain*, is an abstract word domain that defines the output alphabet of control transitions. One symbol of the output alphabet is an abstract precision $\mathfrak{p}$, which denotes $[\![\mathfrak{p}]\!]_{\mathrm{out}} \subseteq H \times \Pi^\infty$ a set of concrete words over the set of *abstraction precisions* Π. The denotation function $[\![\cdot]\!]_{\mathrm{out}} : \mathfrak{P} \to 2^{H \times 2^{\Pi^\infty}}$ maps from a given abstract precision to a map from concerns to sets of precision words, that is, $[\![\mathfrak{p}]\!]_{\mathrm{out}} = \{(h, \{\bar{\pi} \mid \pi_1 \sqcup \ldots \in \widehat{\pi} \wedge \bar{\pi} = \langle \pi_1, \ldots \rangle \in \Pi^\infty\}) \mid (h, \widehat{\pi}) \in \mathfrak{p}\}$. The abstraction function $\langle\!\langle\cdot\rangle\!\rangle_{\mathrm{out}} : 2^{H \to 2^{\Pi^\infty}} \to \mathfrak{P}$ takes the inverse role. Emitting the abstract bottom precision $\bot_\pi$ signals that the state-space modeling process should be stopped at this point.

The infinite *set of all precision transducers* is denoted by $\mathbb{P}$, which is a subset $\mathbb{P} \subset \mathbb{T}$ of abstract transducers.

Since we use abstract precisions as abstracts words, we also must provide means to concatenate them:

Definition 91: Abstract Precision Concatenation

The concatenation $\circ_{\mathfrak{p}} : \mathfrak{P} \times \mathfrak{P} \rightarrow \mathfrak{P}$ of a pair of abstract precisions is the pairwise join of the represented concrete precisions, separated by concern. That is, the concatenation $\mathfrak{p}_1 \circ \mathfrak{p}_2$ of a pair of abstract precisions is equivalent to their join $\sqcup_\pi$.

To deal with precision transducer within an analysis framework, also a neutral precision transducer is needed. This transducer is the neutral element regarding the union and concatenation of precision transducers:

Definition 92: Neutral Precision Transducer

The *neutral precision transducer* P_ϵ is the most abstract precision transducer. It emits the abstract epsilon precision $\mathfrak{p}_\epsilon$ on all its transitions. This precision transducer is defined by the tuple $P_\epsilon = (Q, D_{\text{in}}, D_{\text{out}}, \iota_0, F, \delta)$, with the set of control states $Q = \{q_0\}$, an empty set of final states $F = \emptyset$, a transition relation with a self transition $\delta = \{(q_0, \top, q_0, \mathfrak{p}_\epsilon)\}$, and a set of initial transducer states $\iota_0 = \{(q_0, \mathfrak{p}_\epsilon)\}$.

5.3.1 Precision Transducer Analysis

We now describe an analysis for running precision transducers within the framework of configurable program analysis [31]. A *precision transducer analysis* is a configurable program analysis that runs in parallel—by means of a component of a composite analysis, see Sect. 2.5.3—to other analyses. It performs state transitions of a precision transducer and is responsible for instantiating and providing precisions that are emitted on the matching transducer transition. A precision transducer analysis is defined by the tuple

$$\mathbb{D}_P = (D_P, \rightsquigarrow_P, \downarrow_T, \mathsf{merge}_P, \mathsf{stop}_P, \mathsf{prec}_T, \mathsf{target}_T)$$

and is used to run a given precision transducer $P = (Q, D_{\text{in}}, D_{\text{out}}, \iota_0, F, \delta)$. The analysis has following specific components—the other components are equal to those that are defined for generic abstract transducers:

Abstract Domain D_P. The abstract domain $D_P = (C, \mathcal{E}, [\![\cdot]\!], \langle\!\langle\cdot\rangle\!\rangle)$ is defined based on a map lattice $\mathcal{E} = (J, \top, \bot, \sqsubseteq, \sqcup, \sqcap)$, with $J = 2^{Q \rightarrow \mathfrak{P}}$, where each element $\iota \in J$ of the lattice is a transducer state. One transducer state $\iota = \{(q, \mathfrak{p}), \ldots\} \in J$ is a mapping $\iota : Q \rightarrow \mathfrak{P}$ from control states to abstract precisions. The precision transducer analysis starts with the initial transducer state ι_0 of the precision transducer P to conduct runs for.

Operator $\rightsquigarrow_P$. The transfer relation $\overset{g}{\rightsquigarrow}_P \subseteq J \times G \times J$ defines the set of abstract successor states of an abstract transducer state $\iota = \{(q, \mathfrak{p}), \ldots\} \in J$, for a given control-flow transition $g \in G$. To deal with ϵ-moves, we use the output closure operator that maintains the set of concrete output words, similar to a regular closure operator $\mathsf{abstclosure}_\infty$ as defined in Sect. 3.2.3. Output of consecutive control transitions is combined with the precision concatenation operator. The transfer relation $\rightsquigarrow_{\mathsf{P}}$ is in other respects equivalent to the transfer relation $\rightsquigarrow_{\mathsf{T}}$ of the generic abstract transducer analysis—see Sect. 3.5.1.

OPERATOR $\mathsf{merge}_{\mathsf{P}}$. The merge operator controls if two abstract states should be combined or if they should be explored separately, and separate the state space. We always merge: $\mathsf{merge}_{\mathsf{P}}(\iota, \widehat{\iota}, \cdot)$ returns $\iota \sqcup \widehat{\iota}$.

OPERATOR $\mathsf{stop}_{\mathsf{P}}$. The stop operator $\mathsf{stop}_{\mathsf{P}}$ decides if a given abstract state is covered by a reached state. Because merge_{Π} always merges, all states can be considered covered, that is, $\mathsf{stop}_{\mathsf{P}}$ returns always *true*.

The behavior of the analysis can be configured by choosing different variants of its operators. For example, varying the operator merge can configure the analysis to operate path sensitive, or only context sensitive and flow sensitive [31]. Please note that even if the precision transducer is configured to merge all states, other analyses that run in parallel to this analysis often disagree to this decision and no merge is performed for the composite state. That is, precision transducers can still be executed in a path-sensitive manner if the other analyses that run in parallel enforce to construct such an abstract reachability graph.

Program analyses that run in parallel to the precision transducer analysis, can read, filter, adapt, and (re-)use the emitted precisions in their precision adjustment operator prec [32] for computing abstractions.

5.4 GUIDED PRECISION SHARING

We use precision transducers to express strategies for scoping and sharing of abstraction precisions. The objective of scoping a precision is to keep a set of predicates as local as necessary to *mitigate state space explosion* (and to keep the *costs for computing abstractions* low), but as wide as possible to *reduce the number of refinement iterations*. The number of abstraction refinement iterations is a major cost factor for refinement-based model checking [36, 67]. We provide one precision not only for one specific abstraction task but for several other abstraction tasks as well.

184 Goals

Precisions can be shared *within and among* a verification run: The same precision might be relevant for different parts of the state space, or the verification of different variants [36] of a program. Depending on the goal of a verification run and the structure of a verification task, different strategies for scoping and sharing are appropriate. In software verification, the goal is to verify as many verification tasks as possible within a given resource budget, whereas in software certification the goal is to end up with compact and readable proofs—by increasing the locality of predicates [40, 135, 149]. We use precision transducers to share precisions as candidates for reuse *speculatively*: Not all candidate precisions might contribute to an effective and efficient abstraction process. The *reuse* process includes filtering, adapting, and generalizing the shared abstraction precision—as discussed in Sect. 178. Such heuristics can reduce adverse effects of sharing candidate precisions too liberal, especially if the formula (to abstract) encodes only a small fraction of a program.

185 Reuse

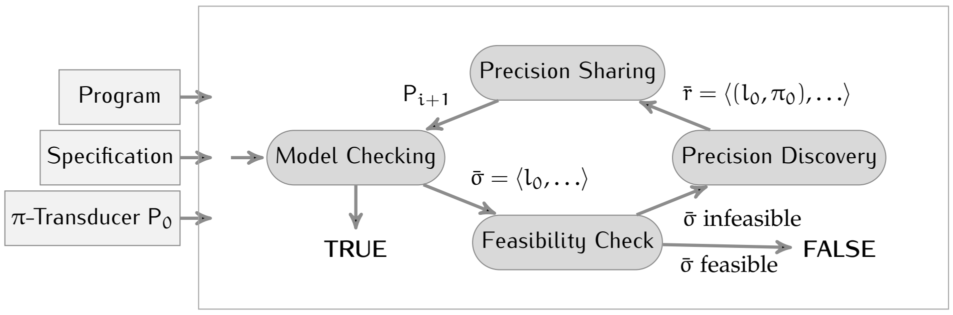


Figure 38: The counterexample-driven abstraction precision refinement workflow with scoping and sharing based on precision transducers.

5.4.1 Precision Transducers from Refinements

Earlier in this chapter, we have outlined a scheme that characterizes the process of discovering, sharing, and reusing abstraction precisions—see Fig. 34. Furthermore, we have provided a general perspective on how the different steps of such a process can be integrated into the overall workflow of a model checking procedure—see Sect. 5.1.3, and Fig. 37. We now outline how we integrate precision transducers into the analysis flow of a model checking procedure that is based on counterexample-guided abstraction refinement (CEGAR) [67].

A CEGAR-based model checker iteratively refines the abstraction precision used to construct an abstract model until this model is sufficiently precise such that no more infeasible counterexamples can be found—or it stops if one of them is feasible. The precision refinement to conduct is determined based on information from the infeasible counterexample, for example, from the Craig interpolants [84, 137, 190] that are available for it.

Workflow. We integrate precision transducers into this workflow as (1) *data structure* to store the abstraction precision to share, we (2) *abstract* the precision transducers to *increase the scope* of sharing, and (3) use the corresponding precision *transducer analysis* as technique to *share* abstraction precisions at different points in the state-space as defined in the precision transducer.

Figure 38 illustrates the refinement workflow with precision transducers integrated: The overall abstraction precision that is taken into account by the model construction (and checking) procedure is determined by the precision transducer P. We start with the initial precision transducer P_0. We refine the precision transducer from iteration to iteration until we end up, after n iterations, with the precision transducer P_n that describes a precision that is sufficient to prove the specification for the given program. The refinement procedure $\mathsf{refine} : L^* \rightarrow (L \times \Pi)^*$ takes as input an infeasible program path $\bar{\sigma} = \langle l_0, \ldots, l_n\rangle \in L^*$ and returns a refinement $\bar{r} \in (L \times \Pi)^*$ that assigns each position (prefixes) of the path the precision needed to rule out this (infeasible) path [137]. From a refinement $\bar{r}$, we construct an intermediate precision transducer P_r that is annotated with precisions that are sufficient to exclude the infeasible path $\bar{\sigma}$ in the next version of the model. Two precisions have a unique role: The bottom precision $\bot_\pi$ is mapped to a point in the path if no information prior this point is relevant to rule out the

186 refine

187 *Precisions $\top_\pi$ and $\bot_\pi$*

infeasible counterexample. The top precision $\top_\pi$ is assigned (by convention) if the corresponding point in the path is considered unreachable—which is dual to the behavior of refinement procedures based on Craig interpolation that emit the interpolant *false* for such points. The intermediate transducer is joined with the last precision transducer P_i. Before we continue with the model checking procedure, we reduce the precision transducer and get $P_{i+1} = \mathsf{reduce}(P_i \cup P_r)$. This step reduces the number of states and transitions of the transducer while maintaining the set of transductions; it is an optimization to reduce the costs for the precision transducer analysis—see Sect. 3.4 for details on reducing abstract transducers. 188

Path Concerns

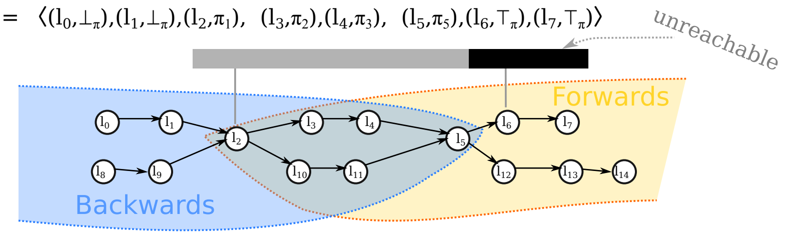
Each program path $\bar{\sigma}$ that is handed over to the refinement procedure refine represents an infeasible violation of a set of properties. Since each of these properties is a program concern, we can map a set $H_{\bar{\sigma}} \subseteq H$ of concerns (the *path concerns*) to the path $\bar{\sigma}$. We map the abstraction precisions that are discovered to rule out an infeasible counterexample $\bar{\sigma}$ to the corresponding set of concerns $H_{\bar{\sigma}}$.

Path Precision Transducers. Given an infeasible program path $\bar{\sigma}$ and the corresponding refinement $\bar{r}$, we construct a precision transducer $P(\bar{r})$ that emits alongside the path $\bar{\sigma}$ exactly those precisions that are needed to rule out this infeasible counterexample. We operationalize a refinement $\bar{r} = \mathsf{refine}(\bar{\sigma}) \in (L \times \Pi)^*$ as a *path precision transducer* $P(\bar{r}) = (Q, D_{\text{in}}, D_{\text{out}}, \iota_0, F, \delta)$. The set Q of control states is constructed from the set of prefixes of the path, such that $Q = \mathsf{prefixes}(\bar{r})$, where the empty prefix ϵ corresponds to the initial state $q_0 \in Q$. A one-to-one mapping of control locations to control states is not possible because a specific control location can be found several times on a given program path. Given a refinement $\bar{r}$, pairings of prefix and abstraction precision are provided by the operator $\mathsf{prefixes} : (L \times \Pi)^* \to 2^{L^* \to \Pi}$, that is, $\mathsf{prefixes}(\bar{r}) \subseteq (L^* \to \Pi)$. The transition relation δ describes the transition between states that correspond to the prefixes $\delta = \{\, (\mathsf{pre}(\bar{l}), \mathfrak{v}, \bar{l}, \mathfrak{p}) \mid (\bar{l}, \pi) \in \mathsf{prefixes}(\bar{r}) \wedge [\![\mathfrak{v}]\!] = (\mathsf{last}(\mathsf{pre}(\bar{l})), \cdot, \mathsf{last}(\bar{l})) \in G \wedge h \in H_{\bar{\sigma}} \wedge [\![\mathfrak{p}]\!] = \{(h, \{\pi\})\} \,\}$.

Expansion to Tree Precision Transducers. We optionally conduct an *expansion* of a refinement $\bar{r} \in (L \times \Pi)^*$ from a single refinement to a set of refinements $R \subseteq (L \times \Pi)^*$ that covers several program paths, which form an annotated (prefix) tree. We introduce this heuristic for increasing the scope of the abstraction precision that has to be added based on the refinement, and thus to reduce the number of refinement iterations. The result is a *tree precision transducer* $P(R) = \bigcup_{r \in R} P(r)$, which is a precision transducer that is constructed as the union of the precision transducers for a set of refinements.

The expansion process is implemented in the operator $\mathsf{expand} : (L \times \Pi)^* \to 2^{(L \times \Pi)^*}$, which takes as input a refinement $\bar{r}$ and provides a set of refinements R as its result. Figure 39 illustrates this expansion. At the heart of our implementation of the operator expand is a flow analysis that identifies additional path segments—which are connected to the program path described by $\bar{r}$—for which an annotated precision might be equally relevant. We conduct a forwards-flow analysis from the first control location, for which the precision is *not* equal to the bottom precision $\bot_\pi$, and collect all reachable control locations $L_F \subseteq L$. We conduct a backward-flow analysis from the first location with the top precision $\top_\pi$ that signals the unreachability of

$\bar{r} = \langle(l_0,\bot_\pi),(l_1,\bot_\pi),(l_2,\pi_1),\ (l_3,\pi_2),(l_4,\pi_3),\ (l_5,\pi_5),(l_6,\top_\pi),(l_7,\top_\pi)\rangle$



$$\begin{aligned} R = \{ & \langle(l_0,\bot_\pi),(l_1,\bot_\pi),(l_2,\pi_1),\ (l_3,\pi_2),(l_4,\pi_3),\ (l_5,\pi_5),(l_6,\top_\pi),(l_7,\top_\pi)\rangle\ , \\ & \langle(l_0,\bot_\pi),(l_1,\bot_\pi),(l_2,\pi 1),\ (l_{10},\pi_1 \sqcup \pi_5),(l_{11},\pi_1 \sqcup \pi_5),\ (l_5,\pi_5),(l_6,\top_\pi),(l_7,\top_\pi)\rangle\ \} \end{aligned}$$

Figure 39: Expansion of a single refinement $\bar{r}$ to a set of refinements R by identifying the *intersection* of the control locations that are reachable forwards and those that are reachable backwards.

this point, which results in the set $L_B \subseteq L$. We construct the expanded set of refinements R based on those path segments that are covered by intersection $L_F \cap L_B$ and connected to the path described by $\bar{r}$. We get from a path precision transducer $P(\bar{r})$ for a refinement $\bar{r}$ to a tree precision transducer by $P(\text{expand}(\bar{r})) = P(R)$.

5.4.2 Sharing Strategies

An abstract transducer emits different abstraction precisions alongside a specific set of program paths. One form of a precision transducer is a path precision transducer, which emits the abstraction precisions for exactly one program path. Nevertheless, providing abstraction precisions to rule out only one infeasible counterexample is inefficient since we deal with a class of program analysis tasks that are known to be prone to the path explosion problem—see Sect. 2.4.1. That is, this form of precision sharing would require an exponential number of precision refinement iterations. Abstracting a precision transducer makes this process more efficient. Abstraction introduces non-determinism [14] to a precision transducer, and the output that was previously emitted for only one input word gets emitted for a broader set of input words.

189 *Role of Abstraction*

The most abstract instance of a precision transducer that we consider in this work is the neutral precision transducer, as defined in Sect. 5.3. The utmost useful abstraction that can be computed of a given precision transducer is the result of merging all its control states and all its transitions and with it the emitted abstraction precisions, and finally applying an input alphabet abstraction such that the remaining self-transition matches for all possible inputs. We later define this form of abstraction as the global sharing strategy. A discussion on abstracting abstract transducers can be found in Sect. 3.3.

The requirement on all approaches for abstracting abstract transducers (and with it precision transducers) is that both the set of transductions and the accepted input language must be overapproximated. A procedure for abstracting a precision transducer must only ensure to overapproximate the set of transductions: We do not make use of the notion of accepted words for precision transducers, that is, the set of final control states F is always empty. Please note that we do not consider generalizing or filtering of shared abstraction precisions in this step.

190 *Overapproximation*

Precision transducers are a powerful concept for operationalizing strategies for scoping and sharing abstraction precisions. We tightly integrate this notion into the workflow that is illustrated in Fig. 38, and define:

Definition 93: Scoping and Sharing Strategy

A *scoping and sharing strategy* defines the scope for that the abstraction precisions that are discovered in a verification run become shared for reuse. The operator $\mathsf{scopeshare} : \mathbb{P} \times (L \times \Pi)^* \to \mathbb{P}$ defines the signature of such a strategy. The operator takes two arguments: The precision transducer P that represents the scoped abstraction precision that was used to create the present abstract model, and a refinement $\bar{r} \subseteq (L \times \Pi)^*$ that defines the abstraction precisions to add to the resulting precision transducer P'. To ensure the progress of the model construction process, we require that the resulting transducer does not lose any abstraction precision, that is, $\mathcal{T}(P') \subseteq \mathcal{T}(P \cup P(\bar{r}))$.

The objectives of scoping and sharing strategies can be counteractive: (1) Keeping a specific level of abstraction precision local can help to prevent an explosion of the abstract state space. (2) Increasing the scope of sharing an abstraction precision as much as possible reduces the number of refinement iterations. (3) Too many facts to track can cause an explosion of abstraction costs—which can be doubly exponential in the number of predicates to track [177].

We now define several (existing and novel) strategies for scoping and sharing abstraction precisions based on precision transducers. We consider only strategies that are applicable for analysis configurations based on predicate abstraction [20, 28, 122] and that use Craig interpolation [84, 190] to discover new abstraction precisions (predicates). In general, any verification technique based on counterexample-guided abstraction refinement [67] could benefit from using (or a study on) one of these strategies [36].

We define the strategies that we study based on the operator $\mathsf{scopeshare}_{\mathsf{care}}$, which is parameterized by the operator care:

$$\mathsf{scopeshare}_{\mathsf{care}}(P, \bar{r}) = \mathsf{reduce}(P \cup \mathsf{care}(\bar{r})).$$

191 *Operator* care

The operator $\mathsf{care} : R \to \mathbb{P}$ takes a refinement and constructs a precision transducer P' that subsumes a given refinement $\bar{r}$ and rules out the corresponding infeasible counterexample. The operator $\mathtt{reduce} : \mathbb{T} \to \mathbb{T}$ reduces—see Sect. 3.4—a given transducer, that is, it reduces the numbers of states and transitions while keeping the resulting transducer equivalent to the input; it joins, for example, all transitions between a pair of control states to one transition if these transitions have the same abstract input word.

Global. The first published strategy for precision sharing in the context of predicate abstraction is simple [122]: All predicates are shared in *one global set of predicates* $\pi_{\mathsf{global}} \subseteq \mathcal{P}$ and are then reused for every abstraction computation. This is the most efficient way to avoid refinement iterations that discover the same predicates repeatedly for different parts of the state space. Nevertheless, this strategy has several drawbacks: (1) details of the state space become modeled at points in the state space where they are not rele-

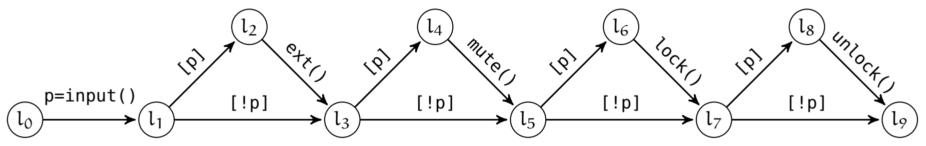


Figure 40: Example for that *global sharing* of predicates on variable p would lead to an *explosion* of the abstract state space.

vant for the verification task at hand, (2) abstraction computation costs are increased needlessly, and (3) it might result in an explosion of the abstract state space—as illustrated by following example:

Example 24. Take the control-flow automaton in Fig. 40 as example. The task is to verify that there is no `unlock()` without a previous `lock()`. Tracking variable p is only (in a path sensitive manner) *beginning* from location $l_5 \in L$ on all flows to location $l_8 \in L$ is sufficient for this proof—beside the current state of a specification automaton that keeps track of the state of the lock. A *global precision sharing* tracks variable p on all locations, that is, beginning after its initialization at location $l_1 \in L$: It causes the abstract state space to explode very early needlessly.

This strategy [122] shares all predicates that have, so far, been identified to be relevant, on each point in the state space for computing abstractions. We use to operator $\mathsf{care}_{\mathsf{Global}}$ for generating a precision transducer from a given refinement $\bar{r}$. The operator is defined as $\mathsf{care}_{\mathsf{Global}}(\bar{r}) = (Q, D_{in}, D_{out}, \iota_0, F, \delta)$, with the set of control states $Q = \{q_0\}$, an empty set of final states $F = \emptyset$, a transition relation with a self transition $\delta = \{(q_0, \top, q_0, \mathfrak{p}_{\bar{r}})\}$, and a set of initial transducer states $\iota_0 = \{(q_0, \mathfrak{p}_{\bar{r}})\}$. The abstract precision $\mathfrak{p}_{\bar{r}}$ is the result of joining all abstraction precisions from a refinement for the set of path concerns $H_{\bar{\sigma}}$, that is, $\mathfrak{p}_{\bar{r}} = \{(h, \bigsqcup\{\pi \mid (l, \pi) \in \bar{r}\}) \mid h \in H_{\bar{\sigma}}\}$. That is, the precision transducer is the result of abstracting a path precision transducer $P(\bar{r})$ by merging all its states and transitions, and conducting an alphabet abstraction such that the remaining transition matches for all inputs.

Counterexample Locations. Another strategy [137] is to localize precisions based on the control locations they are mapped to in a refinement $\bar{r} \in (L \times \Pi)^*$. This idea was presented in work on deriving facts to track from proofs of unsatisfiability [137]. A corresponding precision transducer is derived by the operator $\mathsf{care}_{\mathsf{Location}} : (L \times \Pi)^* \to \mathbb{P}$. Given a refinement $\bar{r} = \langle (l_1, \pi_1), \ldots, (l_n, \pi_n) \rangle$, a call $\mathsf{care}_{\mathsf{Location}}(\bar{r})$ returns a precision transducer with the set of control states $Q = \{q_0\}$, the initial transducer state $\iota_0 = \{(q_0, \mathfrak{p}_\epsilon)\}$, and a transition relation $\delta = \{(q_0, \mathfrak{v}, q_0, \mathfrak{p}) \mid (l_i, \pi_i) \in \bar{r} \land h \in H_{\bar{\sigma}} \land \mathfrak{p} = \{(h, \{\pi_i\})\} \land [\![\mathfrak{v}]\!]_{in} = \{g \mid g = (l, \cdot, l_i) \in G\}\}$ that has a transition for each of the control locations in the refinement.

Counterexample Functions. This strategy is similar to previous strategy (locations) except that the precisions are mapped to all locations of a function. A function-wide sharing [20, 191] is typically applied to reduce the number of refinement iterations. We define the function $\mathsf{care}_{\mathsf{Function}}$ to produce a corresponding precision transducer. Given a refinement $\bar{r} = \langle (l_1, \pi_1), \ldots, (l_n, \pi_n) \rangle$, a call $\mathsf{care}_{\mathsf{Function}}(\bar{r})$ returns a precision transducer with the set of control states $Q = q_0$ and an initial transducer state $\iota_0 = \{(q_0, \mathfrak{p}_\epsilon)\}$.

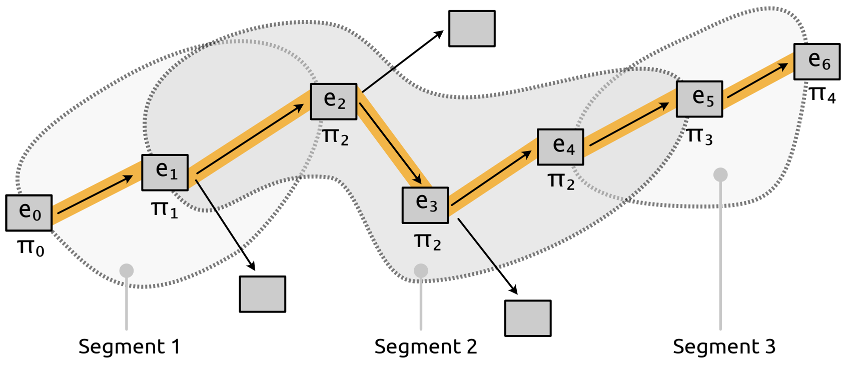


Figure 41: Given a sequence $\bar{e} = \langle e_0, e_1, e_2, e_3, e_4, e_5, e_6 \rangle$ of *abstraction* states along an infeasible counterexample in the abstract reachability graph. The refinement to eliminate this counterexample is given by $\bar{r} = \langle (l_0, \pi_0), (l_1, \pi_1), (l_2, \pi_2), (l_3, \pi_2), (l_4, \pi_2), (l_5, \pi_3), (l_6, \pi_4) \rangle$. The sharing strategy *Context* identifies the refinement segments 1 to 3, expands these segments, and shares corresponding abstraction precisions as soon as the expanded segments are reached. Note that before any control location of the second expanded segment is considered reached, a control location of the first expanded segment must be active before.

The set of transducer transitions is defined by $\delta = \{ (q_0, \mathfrak{v}, q_0, \mathfrak{p}) \,|\, (l_i, \pi_i) \in \bar{r} \wedge h \in H_{\bar{\sigma}} \wedge \mathfrak{p} = \{(h, \{\pi_i\})\} \wedge [\![\mathfrak{v}]\!]_{in} = \{g \,|\, g = (l, \cdot, l_f) \in G \wedge F(l_i) = F(l_f)\}\}$, where $F(l)$ denotes the function to that the given control location $l \in L$ belongs to. That is, the resulting precision transducer is equivalent to a precision transducer that is created by merging all states of a path precision transducer and conducting an alphabet abstraction—from a single control-flow transition to all control-flow transitions in the same function.

Context. With the strategy *Context*, we describe a novel approach that aims at reducing the number of refinement iterations while keeping abstraction precisions as local as possible. From a given refinement $\bar{r}$, we derive (using an operator segment) a sequence of refinement segments: A new refinement segment is started each time the abstraction precision changes alongside a given refinement. Each control state in the resulting precision transducer corresponds to one refinement segment. Figure 41 illustrates the segments alongside a refinement. Each refinement segment is expanded using the expansion operator expand, which widens the scope for that the corresponding abstraction precision is emitted—see Sect. 189, and Fig. 39.

The strategy is implemented in the operator $\mathsf{care}_{\mathsf{Context}}$. We use the *segmentation operator* segment : $(L \times \Pi)^* \to (\mathfrak{I} \times (L \times \Pi)^*)^*$ to derive a sequence $\bar{t} = \langle t_1, \ldots, t_n \rangle$ of pairs from a given refinement $\bar{r}$, where one pair $t_i = (\mathfrak{v}, \bar{s})$ consists of a start trigger condition $\mathfrak{v}$ and a refinement segment $\bar{s}$. Each pair $t_i \in \bar{t}$ has a corresponding state $q = Q(t_i)$ in the set of control states Q of the resulting precision transducer $P = (Q, D_{in}, D_{out}, \iota_0, F, \delta)$. The transducer starts in the initial transducer state $\iota_0 = \{(q_0, \mathfrak{p}_\epsilon)\}$, with $q_0 = Q(t_1)$. The transition relation is defined by $\delta = \delta_{next} \cup \delta_{stay} \cup \delta_{stop}$, with

$$\delta_{next} = \bigcup_{t_i = (\mathfrak{v}, \bar{s}) \in \bar{t}} \{ (Q(t_{i-1}), \mathfrak{v}, Q(t_i), \mathfrak{p}(\bar{s})) \}$$

$$\delta_{stay} = \bigcup_{t_i=(\mathfrak{v},\bar{s})\in\bar{t}} \{ (Q(t_{i-1}), \mathfrak{v}_{\mathsf{expand}(\bar{s})}, Q(t_i), \mathfrak{p}(\bar{s})) \}$$

$$\delta_{stop} = \bigcup_{t_i=(\mathfrak{v},\bar{s})\in\bar{t}} \{ (Q(t_{i-1}), \neg\mathfrak{v}_{\mathsf{expand}(\bar{s})}, q_0, \mathfrak{p}(\bar{s})) \}$$

where $\mathfrak{p}(\bar{s}) = \{(h, \bigsqcup_{(l,\pi)\in\bar{s}} \pi) \mid h \in H_{\bar{\sigma}}\}$ is an abstract precision that summarizes all abstraction precisions along a refinement segment for the path concerns $H_{\bar{\sigma}}$, and $[\![\mathfrak{v}_{\mathsf{expand}(\bar{s})}]\!]$ matches the set of control-flow transitions in the expanded segment.

5.5 GUIDED SYNTHESIS

An observation that we made during our work on the verification of Linux kernel models [8, 36] was that it would be possible to save many of the refinement iterations of a CEGAR-based model checker if we employ some precision synthesis (precision mining or predicate invention). A closer look at the discovered predicates reveals that a considerable number of them refer to the specification or the environment model that is woven into the program to analyze, to form an analysis task. That is, predicates to track can be derived directly from the specification and the environment model. Other researchers made a similar conclusion along with the work on the SLAM project [24], which aims at verifying Windows device drivers.

192 Related Work

We briefly outline how existing and novel approaches for precision mining and synthesis can be realized based on the building blocks that are presented in this work. These building blocks include: (1) YARN transducers, which provide a clear separation of the code that is related to the specification (or the environment model) and the code of the program to check, (2) the generic concept of abstract transducers and the corresponding abstract transducer analysis, which provide means to handle parameterized output words, for example, precision templates, and (3) techniques for filtering shared abstraction precisions, which can be employed by abstraction procedures to reduce potential adverse effects of precision sharing. Precision synthesis can be designated to the phase Discover of our Discover-Share-Reuse scheme—see Fig. 34 on page 130.

193 Building Blocks

Approaches for invariant generation [124, 148, 156, 166] can be used to synthesize abstraction precisions while a procedure for synthesizing precisions cannot necessarily replace a procedure for invariant generation. This is because abstraction precisions are a more generic concept—for example, a data-location precision cannot be described as an invariant or invariant candidate.

194 Invariant Generation

Analyses with Precision Synthesis. Any analysis that is integrated into the analysis process (in the form of a component of a composite analysis) can emit arbitrary abstraction precisions for reuse. This is the universal approach for synthesizing abstraction precisions. These shared precisions can have various origins. By running along with the state-space exploration process, the history of observed states and corresponding behaviors can be taken into account, for example, to invent new abstraction precisions.

We use the term *on-the-fly precision synthesis* to denote a precision synthesis approach that operates during the phase *Transfer* of an analysis—see Fig. 37

195 On-the-Fly

for the ITTC scheme. The strengthening $\downarrow$ step of the CPA algorithm, which is performed immediately after all component analyses have computed their successor states, is most appropriate to integrate such a synthesis approach: The information of all other analyses that are executed in parallel are accessible at that point—and should be consistent to another.

Existing analyses, such as the Yarn transducer analysis (see Sect. 4.4.4) can be extended to emit abstraction precisions for reuse. The following example illustrates how such an extended Yarn transducer analysis produces abstraction precisions based on the matching (and possibly parameterized) transitions of a Yarn transducer. 196

Extended Yarn Analysis

Example 25. Given a transition $e \overset{g}{\rightsquigarrow} \{e'\}$ between two abstract states e to e', based on the control-flow transition $g = (l, \texttt{fp} = \texttt{fopen}(\texttt{path}, \texttt{mode}), l') \in G$. The Yarn transducer transition $(q, \mathfrak{v}, q', \mathfrak{y}) \in \delta$, from the control state q to control state q', with $\mathfrak{v} = \{$`$1 = fopen($?,$?)`$\}$ and the parameterized output word $\mathfrak{y}$, with $\mathfrak{y} = \{(\cdot, \{$`$1 != null`$\})\}$, matches. As its result, the strengthening operator $\downarrow_T$ produces an abstract precision $\mathfrak{p}$, and along with it the set of predicates $\{$`fp != null`$\}$, with the expression `fp` bound to parameter `$1`, which is emitted as precision for reuse.

Extending the Yarn transducer analysis with functionality to synthesize abstraction precisions has the advantage that also dynamically created Yarn—such as Yarn to encode the current control state of the transducer, see Sect. 4.5—can be taken into account.

Precision Transducers from Yarn Transducers. The conceptually elegant approach to synthesize abstraction precisions based on a given Yarn transducer is to translate this transducer into a precision transducer. States and transitions of the Yarn transducer have a one-to-one correspondence in those of the resulting precision transducer.

5.6 PRECISION REUSE

Previous sections have outlined how abstraction precisions can be *shared* within a verification run. In the following, we discuss how shared candidate precisions can be reused while mitigating potential adverse effects, and the role of the granularity of predicates from predicate precisions for the performance of an abstraction procedure.

5.6.1 Integration

Precision reuse is conducted whenever an abstraction should be computed, and we have to decide on the desired level of abstraction of the outcome. In the context of a predicate analysis, an abstraction is typically computed based on Boolean predicate abstraction. The precision adjustment operator prec is the operator of a configurable program analysis that typically is used to conduct an abstraction (widening) [31, 32]. We need means to pass the abstract precisions that were shared for reuse to the operator prec.

Algorithm 7 $\mathsf{CPAalg{+}{+}}_{\mathbb{D}}(R_0, W_0)$, based on [31, 32]

Input: a CPA $\mathbb{D} = (D, \rightsquigarrow, \mathsf{merge}, \mathsf{stop}, \mathsf{prec}, \mathsf{target})$,
a set $R_0 \subseteq E$ of abstract states,
a set $W_0 \subseteq R_0$ of abstract frontier states,
where E denotes the elements of the lattice of D

Output: pair of reached abstract states and remaining abstract frontier states

Variables: a set $\mathsf{reached} \subseteq E$, a set $\mathsf{waitlist} \subseteq E$

```
1:  $\mathsf{reached} := R_0$; $\mathsf{waitlist} := W_0$
2:  while $\mathsf{waitlist} \neq \emptyset$ do
3:      $e := \mathsf{choose}(\mathsf{waitlist})$
4:      for each $e'$ with $e \rightsquigarrow (e', \mathfrak{p})$ do
5:          $\widehat{e} := \mathsf{prec}(e', \mathfrak{p}, \mathsf{reached})$
6:          for each $e'' \in \mathsf{reached}$ do
7:              $e_{new} := \mathsf{merge}(\widehat{e}, e'')$
8:              if $e_{new} \neq e''$ then
9:                  $\mathsf{waitlist} := \big(\mathsf{waitlist} \cup \{e_{new}\}\big) \setminus \{e''\}$
10:                 $\mathsf{reached} := \big(\mathsf{reached} \cup \{e_{new}\}\big) \setminus \{e''\}$
11:         if $\neg\, \mathsf{stop}(\widehat{e}, \mathsf{reached})$ then
12:             $\mathsf{waitlist} := \mathsf{waitlist} \cup \{\widehat{e}\}$
13:             $\mathsf{reached} := \mathsf{reached} \cup \{\widehat{e}\}$
14:             if $\mathsf{target}(\widehat{e}) \neq \emptyset$ then
15:                 return $(\mathsf{reached}, \mathsf{waitlist})$
16: return $(\mathsf{reached}, \emptyset)$
```

CPA ***Algorithm with Abstract Precisions.*** The most *consequent integration* of precision transducers and abstract precisions into the CPA framework is to re-define the CPA algorithm. For example, to only hand over an abstract precision between the transfer relation and the precision adjustment operator. Algorithm 7 reflects this adjustment: The modified transfer relation $\rightsquigarrow\, : E \to 2^{E \times \mathfrak{P}}$ produces pairs of abstract states and abstract precisions, which are consumed by the modified precision adjustment operator $\mathsf{prec} : E \times \mathfrak{P} \times 2^E \to E$. This modification removes lots of the noise that was introduced by pairing each abstract state with an abstract precision, but some approaches for dynamic precision adjustment [32] might not fit into this formalization. The CPA+ algorithm [32] can be *lifted* to operate with abstract precisions to keep the flexibility of dynamic precision adjustment. The algorithm is lifted by changing all operators (and data structures) to operate on abstract precisions $\mathfrak{P}$ instead of concrete abstraction precisions Π.

5.6.2 Filtering

At some point in the analysis, an abstraction must be computed based on a given abstraction precision $\pi \in \Pi$. This given abstraction precision is the result of a sharing process, that is, not every detail that is asked to be modeled (included in the abstract model) for can be modeled on a given point of the state space or a modeling thereof is not relevant.

Definition 94: Precision Filtering

Precision filtering is the act to decide on a fraction π_f of a given abstraction precision π to solve a given block abstraction problem with, where this fraction asks to model less information, that is $\pi_f \sqsubseteq \pi$. Precision filtering subtracts only *elementary* abstraction precision from a given concrete abstraction precision.

Typically, the filtering process eliminates a set of elementary abstraction precisions, for example, a set of predicates, from a given precision. Not all of these predicates might apply to the current block abstraction problem. We employ ideas that are similar to those that are used for Craig interpolation—see Sect. 2.4.3: Filtering of predicates based on the vocabulary that they have in common with the block to compute the abstraction for and all blocks that can follow when proceeding the analysis along from the abstraction state. We illustrate this based on Fig. 32, which shows the structure of several block-abstraction problems. Let us assume that the vocabulary of all block-abstraction problems is known upfront—before the phase Transfer starts. Assume that the abstraction for Block a, that is, abstract state $e_a \in E$ should be computed. An abstraction formula must maintain sufficient information to exclude all spurious counterexamples (paths) on that it can be found. That is, it could be relevant to model a predicate in an abstraction state if it can be relevant for one of the operations that can follow in the control flow. For our example, the common vocabulary is defined by the set $\nu_s = \mathsf{vocab}(\varphi_a) \cap (\mathsf{vocab}(\varphi_b) \cup \mathsf{vocab}(\varphi_c) \cup \mathsf{vocab}(\varphi_d))$. A block is encoded as a block formula, for example, Block a by the block formula φ_a. Given a set of predicates π (a predicate precision) to filter, we derive a filtered set $\pi' = \{\rho \mid \rho \in \pi \land \mathsf{vocab}(\rho) \setminus \nu_s = \emptyset\}$, that is, no predicate must use a vocabulary that has no *full overlap* with the common vocabulary ν_s. [197] Common Vocabulary

Dually, the relevant vocabulary can be identified by a live-variables analysis [204], which has been conducted upfront. Then, those predicates that refer to live variables only should be taken into account to compute the abstraction. [198] ⚠ Please note that also SSA indices of atoms can be taken into account for identifying common vocabulary, which is dual to the 'KILL' in a live variable analysis. We do not consider SSA indices in our implementation. Not considering the SSA indices can be seen as a form of sharing, that is, predicates apply to a broader set of problems.

5.6.3 Grinding

The process of filtering an abstraction precision aims at eliminating a set of elementary abstraction precisions from a given abstraction precision, before actually reusing the filtered result for an abstraction computation. We now discuss the granularity of elementary abstraction precisions and situations in that breaking them into smaller pieces can be relevant. We denote this process as precision grinding:

Definition 95: Abstraction Precision Grinding

Abstraction precision grinding is the process of splitting up elementary precisions into more general but again elementary precisions. The process of grinding is implemented by a *grinder*, which is an operator $\mathsf{grind} : \Pi_\square \to 2^{\Pi_\square}$, where $\Pi_\square$ is the set of elementary precisions for a particular precision lattice.

When dealing with abstraction procedures that use a predicate precision, as defined in Sect. 5.2, also the granularity of predicate formulas that are shared for reuse is relevant. The granularity of a predicate has influence on costs for computing abstractions, and the check for coverage. An *indicator* for the granularity of a predicate formula is the number of logical operators that can be found in it to connect its sub-formulas. Depending on the sharing and reuse scenario, choosing a different granularity of shared or reused predicates can lead to a considerably different performance of the model checker. In the context of a verification procedure that conducts a predicate analysis [122], computes predicate abstractions [28, 35], is driven by a CEGAR-loop [67], and that uses Craig interpolation [137, 190] to discover new predicates, we discuss three granularity levels of predicates:

- *Interpolant Atoms.* The standard configuration for using Craig interpolants in such a verification procedure is to extract all Boolean atoms from the interpolants and add them to the abstraction precision. The set of Boolean atoms of a formula ϑ is denoted by $\mathsf{atoms}(\vartheta) \subseteq \mathcal{F}$. That is, this approach adds the atoms of the Craig interpolant ϕ to the abstraction precision, that is $\pi' = \pi \cup \mathsf{atoms}(\phi)$.

  Sharing atoms of an interpolant for predicate abstraction can reduce the number of refinement iterations since the full interpolant might not apply to other abstraction tasks, but separate atoms might be so. On the other hand, the number of predicates to consider increases such that also the cost for computing predicate abstractions increases.

- *Full Interpolants.* We use Craig interpolation to identify predicates that are needed to rule out infeasible counterexamples. For this purpose, an infeasible counterexample is represented as formula ϑ, with $\vartheta \equiv \vartheta^- \wedge \vartheta^+$ *unsat*. While being an overapproximation of the reason for infeasibility—see Sect. 2.4.3 for details on Craig interpolation—a Craig interpolant $\phi \in \mathcal{F}$ can be an arbitrary complex formula—in this case, to overapproximate ϑ^-. One approach is to add the unmodified Craig interpolant to the abstraction precision, that is $\pi' = \pi \cup \{\phi\}$.

  Not splitting the interpolant formula maintains the relationship between its atoms, but reduces the chance to reuse this formula in other contexts—see our discussion on filtering abstraction precisions in the previous section. Splitting the interpolant can, on the other hand, increase costs for computing Boolean predicate abstractions needlessly. We have provided empirical evidence on this in Sect. 4.6.4 on page 105.

- *Abstraction Formulas.* Sharing and reusing full abstraction formulas as predicates is practical, for example, in the context of regression verification and the validation of correctness and error witnesses. This approach can reduce *both* the number of refinement iterations and the costs for computing predicate abstractions since no further combination of predicates is needed—which typically requires an AllSat call.

199 Future Work

We hypothesize that smaller predicates (with less logical connectors) are better for reuse within a verification run, while larger predicates are better for reuse among verification runs, on similar verification tasks—which can be found, for example, in the context of regression verification. We test this hypothesis in future work. Despite the discussed granularities, various mixed approaches to come up with predicates of different granularities are possible, especially for the reuse among verification runs. One approach is, for example, to make those predicates more fine-grained that are mapped to control locations of a program that are more likely to be impacted by changes.

5.7 EMPIRICAL STUDY

This chapter introduces precision transducers as generic means for systematically synthesizing and sharing abstraction precisions for reuse to create abstract models of programs. We now present an empirical study on the practical applicability of precision transducers for expressing different precision sharing strategies, and for precision synthesis. The replication package that is provided along with this work—see Appendix A—contains everything for reproducing the results.

5.7.1 Research Questions

We study if precision transducers are *useful and applicable* to real-world verification problems. To which extent can precision transducers aid a verifier and its refinement procedure to (1) reduce the number of abstraction refinement iterations, to (2) keep the abstract model as compact as possible, and to (3) increase the overall verification performance in terms of solved verification tasks and consumed CPU time? We study the reuse of abstraction precisions *within* verification runs; in earlier work [36], we have studied the reuse of abstraction precisions among runs.

Guided Sharing. Precision transducers can be derived from refinements and can then be used for scoping and sharing abstraction precisions. How is this reflected in the resulting abstract model and the verifier performance?

RQ 6 (Model Construction). Can *transducer-based* strategies for sharing abstraction precisions for reuse help to keep the *abstract model small* while keeping the number of *refinement iterations low*?

RQ 7 (Verifier Performance). Can *transducer-based* strategies for sharing abstraction precisions for reuse help to *increase the verification performance* in terms of solved tasks and consumed CPU time?

Guided Synthesis. Precision transducers can be used for the synthesis of abstraction precisions. How can this improve the verification procedure?

RQ 8 (Synthesis Effect). Can a *transducer-guided* approach for precision synthesis help to reduce the number of abstraction refinement iterations?

RQ 9 (Synthesis Performance). Can a *transducer-guided* synthesis of abstraction precisions help to increase the effectiveness and efficiency of a a verifier in terms of solved tasks and consumed CPU time?

5.7.2 Experiment Setup

Before we describe the actual verifier configurations for that we conduct experiments, we describe our case studies and the verifier settings that are the same for all configurations. The appendix contains a description of the benchmarking environment—see Sect. A.3 on page 169. We answer our research questions based on three case studies, two of them with high practical relevance. Each case study consists of a set of programs and a set of properties to verify, resulting in a total of 3 960 verification tasks:

Case Study BusyBox (B). BusyBox is a suite of Unix tools that is targeted at small or embedded systems. The suite covers command-line tools, but also tools that run as system services and provide different network services. Our set of BusyBox verification tasks covers 30 program modules. The specification is provided by a set of 7 Yarn transducers that represent safety properties that describe the correct usage of the POSIX API. More details of these properties can be found in Table 12 on page 170.

Case Study Community (C). This set covers 250 randomly chosen programs from the International Competition on Software Verification (SV-COMP'17) that have safety properties encoded. That is, we check if an instrumented ERROR statement, which signals an erroneous control location, is reachable.

Case Study Linux (L). This set covers 250 randomly chosen Linux kernel modules that had been extracted from the Linux kernel using the Linux Driver Verification toolkit [159]. Details on these tasks can be found in one of our papers [8]. The specification that we check consists of 14 safety properties specifying correct usage of the Linux kernel API—Table 13 on page 170 describes these properties.

Verifier Configuration. Our implementation is based on CPAchecker [32, 34]. We configure a program analysis based on predicate abstraction [122] with adjustable-block encoding [35]. The refinement is driven by spurious counterexamples (CEGAR) [67] from which we derive Craig interpolants [43, 84]; the set of predicates for a precision is derived by splitting interpolants—which are formulas in predicate logic—into their Boolean atoms. We compute summaries with Boolean predicate abstraction [177] on boundaries of function calls, and after the control-flow merged. The reachability graph is constructed from scratch, that is, beginning from the initial abstract state e_0, after each precision refinement.

Table 8: Performance of sharing and scoping strategies. The mean_A aggregates only those results that are solved by all (A) strategies.

	B	C	L	B	C	L	B	C	L
Strategy		$\lvert\pi\rvert_{max}$			#Refinements			$\lvert$reached$\rvert$	
Context	4	18	15	5	11	25	18 019	1 502	2 975
Function	5	19	21	6	11	79	17 278	2 039	3 412
Global	5	20	17	3	8	12	42 534	2 080	4 521
Location	5	22	18	16	34	126	17 284	2 481	3 370
	B	C	L	B	C	L	B	C	L
Strategy		#Solved			#Solved only by			CPU Time (s), median_A	
Context	31	160	128	6	4	7	46	11	76
Function	27	164	48	2			32	9	320
Global	18	165	137			15	160	8.5	32
Location	18	136	34	3			36	9	460

5.7.3 Experiments

We conduct two groups of experiments to answer our research questions:

Experiments on Guided Sharing. To study transducer-based precision sharing, we construct precision transducers from refinements—see Sect. 5.4. We benchmark the configurations $K_1 = SH \times CS$, with the set of sharing strategies $SH = \{\text{Global}, \text{Location}, \text{Function}, \text{Context}\}$, and the case studies $CS = \{\text{BusyBox}, \text{Community}, \text{Linux}\}$. We take the number of reached abstract states $\lvert\text{reached}\rvert$ and the maximal number of predicates $\lvert\pi\rvert_{max}$ that are shared for each abstraction computation as indicators for the *compactness* of the abstract model.

Experiments on Guided Synthesis. We implement precision synthesis as an extension of the Yarn transducer analysis, as described in Sect. 5.5. This approach is dual to creating a parameterized precision transducer from a given Yarn transducer. We benchmark the configurations $K_2 = SN \times SH \times CS$, with synthesis $SN = \{\text{SynthOn}, \text{SynthOff}\}$ on and off, the sharing and scoping strategies $SH = \{\text{Global}, \text{Location}, \text{Function}\}$, and the case studies $CS = \{\text{BusyBox}, \text{Linux}\}$. That is, we benchmark $\lvert K_2\rvert = 12$ configurations.

5.7.4 Results

We discuss the results of our study, separated by research question. Details on the presentation of our results can be found in Sect. A.1.

RQ 6: Model Construction. RQ 6 addresses the influence of different precision sharing strategies on the size of the abstract model and the number of refinement iterations. Small abstract models are useful, for example, to provide compact verification witnesses [149] for manual inspection—to retrace a correctness proof—of the abstract model. A small number of refinement iterations is essential in all those situations in that additional iterations add considerable costs [36].

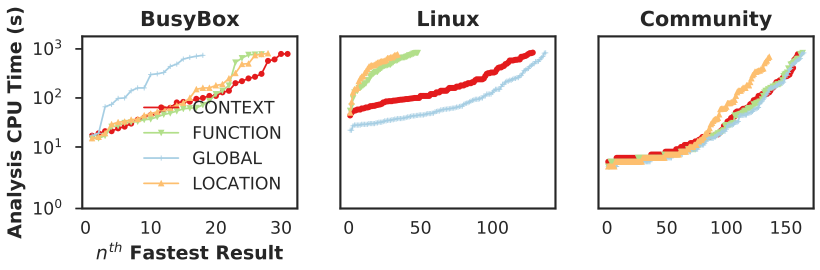


Figure 42: Quantile plots for different precision sharing strategies

Our novel precision sharing strategy Context provides the best results regarding the maximum number of predicates $|\pi|_{max}$ (precision elements) per abstract state. This indicates that a context-based sharing and scoping of precisions based on precision transducers is well suited for ending up with a compact abstract model. This observation is backed by the numbers on the size of the abstract models |reached|: Except for BusyBox, where Context is among the best three strategies, Context leads to (in mean) considerably smaller abstract models. As expected, the strategy Global leads to the lowest number of refinement iterations: All predicates are shared globally and do not have to be rediscovered for different parts of the abstract model repeatedly. Table 8 aggregates our results.

Summary (*RQ* 6) Novel precision sharing strategies that are expressible based on precision transducers can help to *keep the abstract model compact* while *keeping* the number of abstraction *refinement iterations low*.

RQ* 7: *Verifier Performance. Research question 7 asks if there are strategies for scoping and sharing of precisions that can solve tasks faster (CPU time), can solve more tasks, or other tasks.

The novel strategy Context provides the most results (31 solved tasks) for BusyBox, whereas Global can solve the most tasks for Community and Linux. Strategy Context can solve 17 tasks that none of the other strategies can solve, strategy Global solves 15 such tasks, strategy Location three such tasks, and strategy Function solves two of that kind. Only strategy Context can solve additional tasks *across all* case studies. The best performance improvement—a speedup of factor 14—was obtained by strategy Context for the program nested from the Community case study: The best other strategy has solved the task in 430 s, whereas Context needed 60 s. Overall, the strategy Global provides the best performance in terms of CPU time for the analysis, primarily due to the lowest number of refinement iterations needed. Table 8 and Fig. 42 illustrate the results in an aggregated fashion. Table 9 provides a comparison of the results for the strategies Context and Global on the level of verification tasks.

Summary (*RQ* 7) *Novel strategies*, which *are expressible* based on precision transducers, *can solve* tasks—among different program categories—for which established strategies failed. Moreover, new sharing strategies can be more *efficient* for specific tasks.

Table 9: Performance shifts for Community. Comparison of Global and Context. We show only tasks with a shift regarding the analysis time of at least 1 in either direction.

	Analysis Time (s)		Shift	|**reached**|		Shift	#Refinements	
Program	Global	Context		Global	Context		Global	Context
floppy_simpl4	44	140	—●┼—	2 827	2 489	—●—	23	65
floppy_simpl4	44	110	—●┼—	2 848	2 607	—●—	22	57
s3_clnt.blast.04	290	690	—●┼—	5 907	9 377	—●—	20	39
s3_srvr.blast.11	53	120	—●┼—	9 545	23 596	—●┼—	10	16
floppy_simpl3	32	67	—●┼—	2 074	1 867	—●—	18	45
s3_clnt.blast.04	43	90	—●┼—	5 725	6 695	—●—	9	23
s3_clnt.blast.03	45	94	—●┼—	6 624	8 346	—●—	9	23
s3_clnt.blast.01	58	120	—●┼—	7 174	9 093	—●—	11	27
s3_clnt.blast.02	45	90	—●┼—	7 046	8 024	—●—	9	23
s3_srvr_8	540	230	—┼●—	7 941	3 933	—┼●—	26	28
s3_srvr_6	600	250	—┼●—	8 239	4 034	—┼●—	26	28
test_locks_7	60	23	—┼●—	3 949	1 381	—┼●—	13	13
diamond1	160	60	—┼●—	794	783	—●—	50	50
test_locks_8	160	39	—┼●—	8 881	2 729	—┼●—	15	15
test_locks_9	480	79	—┼—●—	19 765	5 421	—┼●—	17	17
nested	400	29	—┼— ●	39 119	958	┼ ●	21	21

RQ 8: Synthesis Effect. We now analyze if transducer-guided precision synthesis can help to reduce the number of refinement iterations.

Our results show a massive difference in the performance gains between the case study BusyBox and Linux. While we can observe the desired decline in refinements for Linux, their number increases for almost all tasks of the case study BusyBox. For BusyBox and the sharing strategy Global, the median number of refinements does not change; for the sharing strategies Function and Location, the median number of refinements more than doubles when enabling precision synthesis. There is not a single case, where we can observe a considerable decline in refinements. The results are in contrast to those for Linux. For example, for the program net-l2tp-l2tp_ip6, precision synthesis can reduce the number of refinements from 110 to 4 (with the strategy Location). However, this is one of several extreme cases. In the median, precision synthesis reduces the number of refinements from 1 to 0, regardless of the scoping and sharing strategy. Table 10 and Table 11 provide the results for the case study Linux on the level of verification tasks: Enabling precision synthesis is beneficial for the majority of the tasks and reduces the number of refinements considerably.

Summary (*RQ 8*) Our results show that precision synthesis can reduce the number of refinements considerably for certain verification tasks.

Table 10: Performance shifts for Linux if precision synthesis is active and sharing strategy Function enabled. We show only tasks with a shift for the analysis time of at least 1 in either direction.

Program	Analysis Time (s) No Synthesis	Analysis Time (s) Synthesis	Shift	|reached| No Synthesis	|reached| Synthesis	Shift	#Refinements No Synthesis	#Refinements Synthesis
net-l2tp-l2tp_ip6	15	35		3 121	3 378		2	4
net-wan-sbni	43	21		4 922	4 922		0	0
oss-uart6850	16	6		1 692	1 221		8	0
tty-nozomi	33	13		11 394	8 882		3	0
net-wireless-ray_cs	57	16		13 429	13 375		5	0
usb-gadget-function-u_serial	180	48		116 692	119 060		2	0
char-mwave-mwave	170	17		4 726	7 082		24	0
snd-aloop	290	9		7 653	4 278		27	0

Table 11: Performance shifts for Linux if precision synthesis is active and sharing strategy Global enabled. We show only tasks with a shift for the analysis time of at least 1 in either direction.

Program	Analysis Time (s) No Synthesis	Analysis Time (s) Synthesis	Shift	|reached| No Synthesis	|reached| Synthesis	Shift	#Refinements No Synthesis	#Refinements Synthesis
net-ethernet-smsc-epic100	34	12		7 420	6 779		1	0
fs-coda-coda	40	14		5 874	3 092		7	1
memstick-core-mspro_block	48	12		8 384	7 876		1	0
block-paride-pg	440	79		5 437	5 466		6	1
snd-aloop	79	9		4 804	4 278		2	0
char-mwave-mwave	330	17		5 112	7 082		17	0

RQ 9: Synthesis Performance. The last question focuses on the influence of precision synthesis on the *performance* of a verifier: Does transducer-guided precision synthesis result in faster verification runs or more solved tasks?

For the case study BusyBox, our results would suggest that precision synthesis does not work at all. Not only do we lose about half of the results for each sharing and scoping strategy, but also do the remaining tasks need more time to complete. Positive speedup is only achieved for single tasks. The least efficient, but most effective strategy here is Location, with a median speedup of 0.5. For Linux, we obtain a different picture: Each of the scoping and sharing strategies yields *more results* if precision synthesis is enabled (Global: 82, Function: 57 and Location: 78 additional results). Also, the *speedup* gained by enabling transducer-guided precision synthesis is positive: With a peak speedup of 64 for the strategy Function, where precision synthesis reduces the analysis CPU time of the program sound-drivers-snd-aloop from 640 s to 10 s. In the median, however, only a moderate speedup can be achieved (Global: 1.2, Function: 1.2, Location: 1.2).

Summary (*RQ 9*) The performance of synthesis *depends on the case study*. While the performance of precision synthesis based on precision transducers suffers for BusyBox, both efficiency and effectiveness are *increased considerably* for Linux.

5.7.5 Discussion

Our results demonstrate that precision transducers are a useful and practical concept for software verification. Novel sharing strategies that are made expressible based on precision transducers can solve tasks for which established strategies fail. The performance of the sharing and scoping strategy Global was surprising because we would have expected that the costs for predicate abstraction would explode in many cases. At least for our verifier configuration and the set of benchmark programs, results suggest that reducing the number of refinement iterations seems to be one of the best ways to gain performance in terms of time for a verification run. Nevertheless, the promising results of our novel strategy Context witness that there are strategies that become expressible with the introduction of our precision transducers and that make verifiers more effective in terms of solved tasks. Expanding the scope of precisions based on statistics on refinements could be one way to go [147] in the future. In general, we observe that the chosen strategy for scoping and sharing has a massive influence on the performance of a verification approach. That is, the chosen strategy should be at least a controlled variable in empirical studies on abstraction-based verification techniques! For precision synthesis, the results heavily depend on the case study: Linux supports our hypothesis that precision synthesis *can* make verification more effective and efficient by reducing the number of refinements; this conclusion is not possible based on the results for BusyBox. While we were able to come up with exciting results, we consider them to be only the start of a series of studies on transducer-guided synthesis and sharing of abstraction precisions. The concept of precision transducers provides a perspective that brings problems of sharing, scoping, and synthesizing of precisions into the focus, and thus, the problem of balancing the number of refinement iterations and the size of the abstract model.

5.7.6 Threats to Validity

Several threats to validity have to be considered. The complexity of verification tasks—pairs of programs and specification—can have, as our results indicate, a substantial influence on the efficiency and effectiveness of a verifier configuration. We would need a sampling strategy that could derive a set of tasks that are representative of the whole population. The verification of LINUX modules has a high practical relevance, thus studying approaches on them is important. Our set of BUSYBOX modules is quite small, but the properties we check are considerably different from those of the LINUX kernel modules. Furthermore, not all verification tasks allow the same degree of abstraction, whereas the LINUX and BUSYBOX tasks allow to abstract away many details; smaller tasks—as found in the set of SV-COMP tasks—tend to have a higher percentage of relevant statements or memory locations. A major factor that influences the performance of scoping and sharing strategies is the size of blocks that should get summarized by abstraction computations: For larger blocks, more of the shared predicates might apply; smaller blocks tend to cause more refinement iterations. Possible configurations [35] of block sizes range from computing an abstraction after each program operation to computing abstractions only on loop heads or the n-th unrolling of a loop—or even larger blocks [28, 35]. A crucial factor for refinement-based techniques is the procedure for discovering facts to track. An infeasible counterexample can be ruled out by different sets of predicates, whereas these predicates can be distributed differently along the infeasible program path. Different strategies for scoping and sharing might be preferable depending on the distribution of facts to track in the control flow. Our refinement procedure depends on Craig Interpolation and the heuristics that are implemented in the solver to derive interpolants. We have conducted our experiments with a predicate analysis; other abstract domains may provide different results.

5.8 RELATED WORK

We now discuss work that is related to the contributions of this chapter. One central contribution is the concept of *precision transducers*, a concept based on abstract transducers for synthesizing, scoping, and sharing precisions.

Abstraction Precision. We provide a formalization of abstraction precision and different notions that build on this formalization: a precision lattice, precision equality, candidate and actual precision, and the notion of an elementary precision. Furthermore, we define the notion of precision scope and introduce the notion of abstract precision to allow for concern-specific abstraction precisions. Nevertheless, the term precision is well-established in the literature to describe the information content [89] of an abstract model. The term precision is applied in the program analysis and verification community, mostly in an informal fashion [32, 67, 119, 136]. The formalization that is closest to our notion of abstraction precision was provided by Nayak and Levy [202]: They define a model level specification that specifies *how* a set of abstract interpretations is derived from another set of interpretations.

Automata-Aided Abstraction. A restricted form of precision transducers had been presented in terms of interpolant automata [133]. States of interpolant automata are annotated with inductive invariants—for example, Craig interpolants [84]—thus their language is the set of correct program traces. Instead, precision transducers aid abstraction-based techniques, such as predicate abstraction [122, 177], by providing predicates—which are not necessarily inductive invariants—as precision. Besides emitting precisions for reuse, precision transducers do not stipulate any restrictions of the accepted language, and they are independent of the abstract domain of the program analysis. So, every interpolation automaton is a precision transducer, but not vice versa! Interpolant automata and the related concept of trace abstraction are closely related to the IMPACT [191] approach, whereas precision transducers can be seen as their counterpart for predicate abstraction.

Precision transducers provide a perspective that brings the problems of sharing, scoping, and synthesizing of abstraction precisions into the focus, and thus, the problem of keeping the number of refinement iterations small, and the resulting abstraction as minimal as possible. We instantiate the concept of precision transducers in terms of abstract transducers.

Discovery and Synthesis. Abstraction-based model checking techniques rely on procedures that identify facts about a program to track—i.e., the precision—such that the correctness regarding a specification can be proved. Relevant techniques range from using an user-provided precision [20, 122], (informed) guessing [105], reuse from previous verification runs [36], to more systematic approaches, such as using Craig interpolation [137, 191], UNSAT cores [165], or computing weakest preconditions [147]. Another technique to infer predicates are templates [238]; parameterized precision transducers can be considered to be annotated with such templates. Precision transducers can provide context-dependent templates, that is, they provide different templates depending on the state of program analysis. Precision transducers can be derived directly from a formal specification—which might be already given as a set of automata—to reduce the number of specification-related refinement iterations. The possibility of mining predicates (or precision elements) on-the-fly was already discussed earlier [32].

Scoping and Sharing. Precision transducers are a means to implement different strategies for sharing abstraction precisions for reuse in different scopes. This concept is independent of a specific abstract domain and separates precision sharing clearly into a cohesive building block of a model checker. It is a step towards a generic abstraction refinement procedure.

Precision transducers consummate the idea of lazy abstraction [136] to share a precision depending on the context, that is, the same program fragment might be represented in different parts of the abstract state space, but can be modeled with different levels of precision. The state-space exploration algorithm that was proposed along with lazy abstraction uses one data structure (an ARG) to represent both the state space and the abstraction precisions in the different parts. This interwinding of different types of data adds additional complexity to the algorithm: The state-space graph of each preceding iteration must be maintained in memory to be able to not lose the mapping of abstraction precisions into the different parts of the state space. Precision transducers overcome this limitation by providing a data structure for abstraction precisions that is independent of those of the state space.

Literature discusses a number of strategies for scoping and sharing predicates for predicate abstraction: Early approaches use all predicates globally [122]. Local sharing has been proposed [137] to reduce the costs for predicate abstraction—Craig interpolation is used to localize predicates. Function-wide sharing [20, 191] has been applied to reduce the number of refinement iterations. To find the balance between reducing the number of refinement iterations and costs for abstraction computations, a heuristic [147] for automatically escalating the scope of predicates has been proposed. Precision transducers are an expressive concept to describe and control context-dependent sharing and scoping of abstraction precisions. The fact that precision transducers specify languages of refinements makes our approach related to techniques that take several counterexamples into account [242] to come up with a precision refinement.

Task Artifact Reuse. With precision transducers, we provide a means for systematically sharing abstraction precisions within and among verification runs, for example, for incremental verification or regression verification. In our previous work on precision reuse [36] we evaluated precision reuse among verification runs for regression verification of Linux device drivers. In another work [29], we proposed to reuse error witnesses among verification runs, for example, to validate them.

There is a large body of work on reusing task artifacts among verification runs. These approaches reuse state-space graphs [33, 135, 182], constraint solving results [248, 253], function summaries [233], or counterexample traces [44]. The reuse of correctness proofs and error witnesses has been proposed [63] for formal regression verification of hardware. An interesting approach to identify reusable artifacts for a given reasoning problem is to use hash values [129]. A more general fashion of reuse is to store and reuse canonicalized constraint-solver queries and the corresponding results [248].

5.9 SUMMARY

200 *Formal Foundation*

201 *Precision Transducers*

202 *Guided Precision Sharing*

203 *Guided Precision Synthesis*

204 *Empirical Evidence*

We started by providing a formal definition of *abstraction precisions*. Based on this definition, we provide concepts such as an abstraction precision lattice, an elementary precision, the precision scope, and the notion of an abstract precision. Building on this foundation, we presented *precision transducers*, a class of abstract transducers that emit abstraction precisions as output. Precision transducers have *several applications* in the context of automatic program analysis, but also other fields: *Scoping and sharing* of abstraction precisions, *synthesis* of program invariants candidates, and program *comprehension*. We study two use cases, for which we present a set of novel techniques: Guided precision scoping and sharing, and guided precision synthesis. Our results highlight the importance of a well-chosen strategy for sharing and scoping abstraction precisions for the performance of a verifier, and the research that is conducted based on them. This work provides a new framework for formalizing different strategies for the reuse of abstraction precisions. The framework supports both sharing and reuse within a verification run and among different runs. We provided evidence on the usefulness of techniques built on the concept of precision transducers: Precision transducers, helped us to solve verification tasks, for that traditional verification techniques failed. Predicate discovery, sharing, and reuse is an active area of research [93], and many new important questions arise with each work.

> “*The sublime in art is the attempt to express the infinite without finding in the realm of phenomena any object which proves itself fitting for this representation.*”
>
> — Georg Wilhelm Hegel

6 CONCLUSION

This work was motivated by the problem of repeated, similar, computational, and manual efforts in the creation of syntactic and semantic task models.

205 Syntactic Task Models

Parts. The control-flow structure of an analysis task is represented by its syntactic task model. We describe the creation of a *syntactic* task model as the composition from several syntactic task artifacts. A syntactic task artifact is, for example, a control-flow relation that represents a full program, a program component, an aspect, or a function thereof. Chapter 4 presented Yarn transducers as a means for representing and sharing syntactic task artifacts and the Loom as means for reusing them to compose the syntactic model of an analysis task.

206 Semantic Task Models

207 Abstraction

208 Abstraction Precision

A *semantic* model is constructed to reason about the actual behavior and states of a program (which is represented by its syntactic model). Since a program can exhibit infinite states and behaviors, a finite abstraction thereof must be constructed to make sound statements, for example, to prove that a program adheres to a specification. Only that fraction of information about the state space of a program is maintained in the abstraction, which is relevant for the reasoning task at hand. The set of facts about the program (its states, and behaviors) to model is called the *abstraction precision*. Chapter 5 presented *precision transducers* as a means for sharing abstraction precisions, to define which information from the different parts of a state-space should be maintained to conduct an analysis task.

209 Words and Languages

Both, a syntactic task model and a semantic task model can be described as a set of words over corresponding alphabets. A letter of the alphabet of the syntactic task model is a program operation, for example, a variable assignment, or a guard (assume) [94]. A letter from the alphabet of a semantic task model can be a concrete state the program can have [189]. This perspective motivated us to present an *automata-theoretic* approach for sharing task artifacts along with adequate mechanisms for reusing those. Chapter 3 presented *abstract transducers* as the common conceptual foundation for systematically sharing abstract entities for reuse. Abstract transducers are a new type of machine with unique semantics that required the design of new algorithms to provide the operations known for automata.

210 Common Formalism
Methodology. While the work covers topics from different sub-disciplines of computer science (formal methods, theory of computation, programming languages, algorithms and data structures, software engineering), we stick to a *common formalism* for all of our contributions, among all chapters—or discuss the relationships between the different perspectives or dual notions. Where necessary, the work introduces new theoretical foundations and formalisms or extends existing ones. Differences to concepts that can be found in the literature are discussed where appropriate and corresponding references are given. To present and implement our program analysis techniques, 211 CPA we build on the CPA framework [31, 32] and the corresponding tool CPAchecker [34], which strictly implements the CPA formalism. CPA builds on abstract interpretation [80] and adds configurable operators to express different program analysis approaches.

212 Empirical Evidence
To support our claims and to show the applicability of our concepts and techniques, we provided *empirical evidence* based on various experiments. The experiments were conducted on thousands of verification tasks. The tasks include real-world problems (Linux), community-agreed benchmarks (SV-COMP), and handcrafted scenarios. The *scenarios* have a well-known control-flow structure and data dependencies and help to arrive at fundamental insights into the performance characteristics of a model checker.

213 Replication Package
Along with this work, we provide a *replication package* with all tool configurations, the tool implementations, verification tasks, and the raw results.

Practicality. The projects and papers that were conducted along with this 214 Linux Verification thesis had a considerable practical impact. Especially our work in collaboration with the Linux Verification Center[1] (LDV), which resulted in several papers, among two [8, 36] with the author of this work. The performance gain that is obtained by precision reuse helped to identify bugs [36] in the Linux kernel and resulted in corresponding patches. Our lightweight specification and weaving approach [8] was well-received by the LDV team and reduced the upfront instrumentation effort. One aim of the empirical studies that were conducted for this work was to illustrate the practical applicability of the novel techniques that are presented.

1 http://linuxtesting.org

A REPLICATION PACKAGE

To make our results convenient to reproduce, we provide a replication package that includes the verification tasks of all case studies, the tool configurations, the raw results, and the full source code of our implementation. Fractions of this replication package have already been published along with our previous papers [8, 36]—where these artifacts *have been evaluated positively* by the corresponding conferences and their *artifact evaluation committees*. The replication package can be found on the supplementary Web page of the thesis: `https://andreas.stahlbauer.net/thesis/`.

A.1 PERFORMANCE AND ITS PRESENTATION

Our empirical studies have a strong focus on the performance of different analysis configurations. We characterize the *performance* of an analysis configuration by the efficiency and effectiveness that it provides for a given set of analysis tasks. In the majority of the cases, an analysis configuration is the configuration of a verifier and we use the term verifier configuration.

215 Efficiency and Effectiveness

We measure the *efficiency* of a verifier configuration by the (CPU) time spent to solve a given verification task—or a set of thereof. We measure the *effectiveness* of a verifier configuration in terms of solved verification tasks, that is, the number of verification tasks for which the verifier (in the particular configuration) was able to provide a result (verification verdict) within the given resource bounds.

216 Analysis CPU Time

The *CPU time for analysis* excludes the time for parsing the program, constructing the control-flow automaton, and other initialization steps. In some cases we aggregate only results of tasks that were solved by all approaches; we indicate this with the index A on the measure. We report all time measures with *two significant figures*, for example, 723.8296 s is reported as 720 s, and 1.42941 s is reported as 1.4 s.

217 Significant Figures

A.1.1 Performance Shift

We introduce the notion of *performance shift*. Given a performance measure, and two values $v_1, v_2 \in \mathbb{R}$ for a particular performance measure for two analysis configurations c_1 and c_2. The *performance shift* is *in favour of* the left configuration c_1 if $v_1 < v_2$, and it is in favour of the right configuration c_2 if $v_1 > v_2$. We define and use the function $\mathsf{shift} : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$ as follows:

$$\mathsf{shift}(v_1, v_2) = \begin{cases} (v_1/v_2) - 1 & \text{if } v_1 > v_2 \\ (v_2/v_1) + 1 & \text{if } v_1 < v_2 \\ 0 & \text{otherwise} \end{cases} .$$

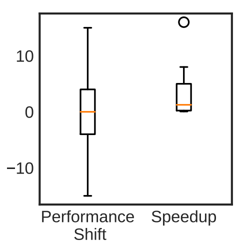


Figure 43: Visual difference in plotting the pairs of performance measures $\{(1,2),(2,1),(4,1),(1,4),(8,1),(1,8),(1,16),(16,1)\}$ as box plots of (a) the performance shifts and (b) the speedups.

The performance shift can be plotted with standard plots, such as box plots or violin plots, while ensuring that *visual distances* match to performance differences. An established approach to compare the performance between two performance measures is to compute the *speedup* with the function $\mathsf{speedup}(v_1, v_2) = v_1/v_2$. Speedup differences are, compared to performance shifts, harder to assess based on standard plotting techniques; we illustrate this in Fig. 43. It is valid to compute the *arithmetic mean* of performance shifts, which is not the case for speedups.

A.1.2 Sensitivity Plot

In this work, we evaluate various analysis configurations regarding their performance on a set of analysis tasks with different characteristics. We present sensitivity plots as a systematic way to study the impact of different configuration parameters. Before we describe this type of plot in detail, we introduce formalisms that are relevant in this context.

An *analysis configuration* is a tuple $\overline{k} = (v_1, \ldots, v_n)$ of configuration arguments (parameter values, or configuration values) for a corresponding list $\langle k_1, \ldots, k_n \rangle$ of *configuration parameters*. An analysis configuration is called *partial* if some of the parameters are not bound to a specific value. We call an analysis configuration *sensitive* to a value for a specific configuration parameter (the variable under sensitivity analysis) if its alternation leads to a considerable difference in the observable performance.

A *sensitivity plot* illustrates the sensitivity of a set of analysis configurations to the change of one particular parameter. Given a list of analysis configurations $K_p = \langle \overline{k}_1, \ldots, \overline{k}_n \rangle$, with $\overline{k}_i = (v_1, \ldots, v_m)$, that should be studied based on a sensitivity plot. The plot illustrates the performance shift regarding a particular performance measure if the studied parameter k_i is changed from its base value v_b to a specified parameter value v_s. Figure 44 shows such a sensitivity plot with annotations that explain the different parts of the plot.

A.2 IMPLEMENTATION

Along with this replication package, we provide the implementation of our techniques as open source, under Apache 2.0 licence. Our tool builds on the CPA framework [31, 32] and uses CPAchecker [34] as its foundation.

A threat to validity of our experiments is the possible presence of *bugs* in the tool implementation, solvers, or libraries. At some point, we had to work on a separate branch of the code to ensure stable results and to allow efficient modifications of the framework itself. Our code base does not reflect the latest *bug fixes* and optimizations that can have an influence on the performance and the soundness of the different analysis components. Nevertheless, we have tried to ensure the correctness and soundness of the implementation as much as possible, especially to guarantee the correctness of the results for handcrafted scenarios and examples.

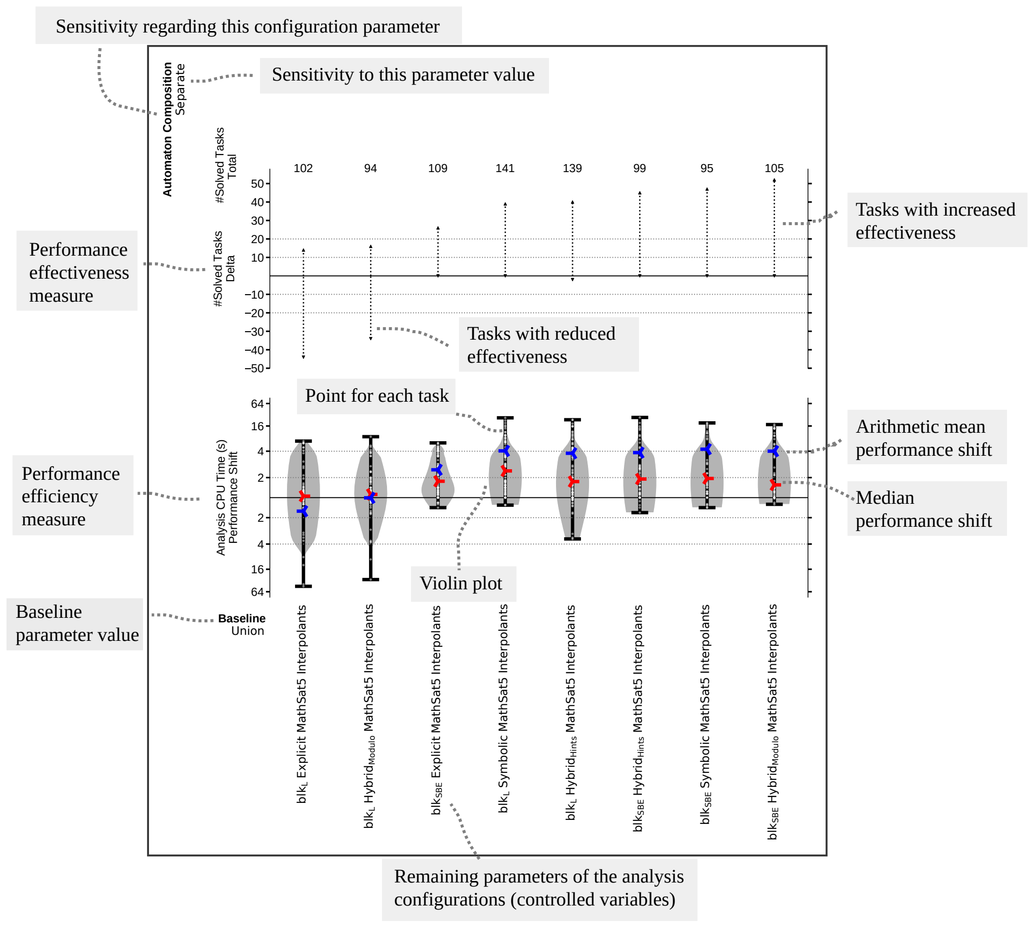


Figure 44: Sensitivity plot and its components

The *formalisms* that we use to describe our concepts and techniques evolved and generalized while writing this thesis, and are not yet fully reflected in the tool versions that were used to conduct the experiments.

A.3 BENCHMARKING ENVIRONMENT

We carefully control variables of our experiment setup. The machines, on which we conduct our experiments, are based on a NUMA shared memory architecture; we therefore bind the CPU cores to their local memory banks to ensure reliable performance measures. If not stated otherwise, we limit the process CPU time for each verification run to 900 s and its memory to 15 GB. Since our verifier is written in Java, we configure the JVM such that the effect of the Just-In-Time (JIT) compiler is reduced to a minimum by forcing the JVM to compile the full bytecode during its startup; the initial heap size is at the maximum (10 GB). We use MATHSAT5 [65] as the *default* SMT solver and JAVABDD for representing abstraction formulas (summaries that are the result of abstraction computations) as BDDs—which reduces costs for coverage checks [177].

Each experiment was conducted on a homogeneous set of machines. We use two different sets of homogeneous machines: One set with Intel Xeon E5-

Table 12: The list of checked POSIX safety properties. The full POSIX documentation can be found at https://pubs.opengroup.org/onlinepubs/9699919799/.

Property	Description
File State	Only *valid file handles* must be accessed by stdio functions.
File Limit	A process must have open a limited (5) *number of file handles*.
Dir State	Only *valid directory handles* must be accessed.
Dir Limit	A process must have open a limited (5) *number of directory handles*.
Division	No calculation must result in a *division by zero*.
Socket	Only valid *socket descriptors* must be passed as arguments.
AddrInfo	Only valid *address descriptors* must be passed as arguments.

2650 v2 CPUs and 128 GB of RAM, and another set with Intel Xeon E5-2620 v4 CPUs and 256 GB of RAM. The performance characteristics can vary dramatically on other machines, with faster or slower RAM, larger or smaller caches, or a different CPU architecture. We reduced the effects of other processes or users on the machine as much as possible by using dedicated benchmarking machines that were used exclusively by the author of this work.

Table 13: List of LDV safety properties. More details on the API functions can be found in the Linux kernel documentation: `https://www.kernel.org/doc/htmldocs/kernel-api/`. Table taken from [8].

Property	Description
08_1a	Each module that was referenced with `module_get` must be released by `module_put`.
10_1a	Each memory allocation that gets performed in the context of an interrupt must use the flag `GFP_ATOMIC`.
32_1a	The same mutex must not be acquired or released twice in the same process.
43_1a	Each memory allocation must use the flag `GFP_ATOMIC` if a spinlock is held.
68_1a	All resources that were allocated with `usb_alloc_urb` must be released with `usb_free_urb`.
68_1b	Each DMA-consistent buffer that was allocated with `usb_alloc_coherent` must be released by calling `usb_free_coherent`.
77_1a	Each memory allocation in a code region with an active mutex must be peformed with the flag `GFP_NOIO`.
101_1a	All structs that were obtained with `blk_make_request` must be released by calling `blk_put_request` afterwards.
106_1a	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.
118_1a	Reader-writer spinlocks must be used in the correct order.
129_1a	An offset argument of a `find_bit` function must not be greater than the size of the corresponding array.
132_1a	Each device that was allocated by `usb_get_dev` must be released with `usb_put_dev`.
134_1a	The `probe` functions must return a non-zero value in case of a failed call to `register_netdev` or `usb_register`.
147_1a	RCU pointer/list update operations must not be used inside RCU read-side critical sections.

BIBLIOGRAPHY

[1] *ATIS Telecom Glossary 2011*. Standard. 2011.

[2] M. Abadi and G. D. Plotkin. "A Logical View of Composition." In: *Theor. Comput. Sci.* 114.1 (1993), pp. 3–30.

[3] P. A. Abdulla, Y. Chen, L. Holík, R. Mayr, and T. Vojnar. "When Simulation Meets Antichains." In: *Proc. TACAS*. Vol. 6015. LNCS. Springer, 2010, pp. 158–174.

[4] S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum. *Handbook of Logic in Computer Science, Semantic Structures*. Vol. 3. Clarendon Press, 1994.

[5] C. Allan, P. Avgustinov, A. S. Christensen, L. J. Hendren, S. Kuzins, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. "Adding trace matching with free variables to AspectJ." In: *Proc. OOPSLA*. ACM, 2005, pp. 345–364.

[6] S. Apel, D. S. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013.

[7] S. Apel, D. Beyer, K. Friedberger, F. Raimondi, and A. v. Rhein. "Domain Types: Abstract-Domain Selection Based on Variable Usage." In: *Proc. HVC*. Vol. 8244. LNCS. Springer, 2013, pp. 262–278.

[8] S. Apel, D. Beyer, V. O. Mordan, V. S. Mutilin, and A. Stahlbauer. "On-the-fly Decomposition of Specifications in Software Model Checking." In: *Proc. FSE*. ACM, 2016, pp. 349–361.

[9] S. Apel, A. von Rhein, P. Wendler, A. Größlinger, and D. Beyer. "Strategies for product-line verification: case studies and experiments." In: *Proc. ICSE*. IEEE, 2013, pp. 482–491.

[10] S. Apel, C. Lengauer, B. Möller, and C. Kästner. "An algebraic foundation for automatic feature-based program synthesis." In: *Sci. Comput. Program.* 75.11 (2010), pp. 1022–1047.

[11] U. Aßmann and A. Ludwig. "Aspect Weaving with Graph Rewriting." In: *Proc. GCSE*. Vol. 1799. LNCS. Springer, 1999, pp. 24–36.

[12] P. Avgustinov, J. Tibble, and O. de Moor. "Making trace monitors feasible." In: *Proc. OOPSLA*. ACM, 2007, pp. 589–608.

[13] P. Avgustinov et al. "Aspects for Trace Monitoring." In: *Proc. FATES/RV*. Vol. 4262. LNCS. Springer, 2006, pp. 20–39.

[14] G. Avni and O. Kupferman. "When does abstraction help?" In: *Inf. Process. Lett.* 113.22-24 (2013), pp. 901–905.

[15] D. Babic and A. J. Hu. "Structural Abstraction of Software Verification Conditions." In: *Proc. CAV*. LNCS 4590. Springer, 2007, pp. 366–378.

[16] R. Back and J. von Wright. "Duality in Specification Languages: A Lattice-Theoretical Approach." In: *Acta Inf.* 27.7 (1990), pp. 583–625.

[17] A. D. Baddeley and G. Hitch. "The recency effect: Implicit learning with explicit retrieval?" In: *Memory & Cognition* 21.2 (1993), pp. 146–155.

[18] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner. "Thorough static analysis of device drivers." In: *Proc. EuroSys*. ACM, 2006, pp. 73–85.

[19] T. Ball, V. Levin, and S. K. Rajamani. "A Decade of Software Model Checking with SLAM." In: *Commun. ACM* 54.7 (2011), pp. 68–76.

[20] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani. "Automatic Predicate Abstraction of C Programs." In: *Proc. PLDI*. ACM, 2001, pp. 203–213.

[21] T. Ball, A. Podelski, and S. K. Rajamani. "Boolean and Cartesian Abstraction for Model Checking C Programs." In: *Proc. TACAS*. Springer, 2001, pp. 268–283.

[22] T. Ball and S. K. Rajamani. *SLIC: A Specification Language for Interface Checking (of C)*. Tech. rep. MSR-TR-2001-21. Microsoft Research, 2002.

[23] T. Ball and S. K. Rajamani. "The SLAM project: Debugging system software via static analysis." In: *Proc. POPL*. ACM, 2002, pp. 1–3.

[24] T. Ball and S. K. Rajamani. "The SLAM Toolkit." In: *Proc. CAV*. Vol. 2102. LNCS. Springer, 2001, pp. 260–264.

[25] T. Ball and S. K. Rajamani. *Generating abstract explanations of spurious counterexamples in C programs*. Tech. rep. Technical Report MSR-TR-2002-09, Microsoft Research, 2002.

[26] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, V. Prevosto, and I. Saclay Île-de france. "ACSL: ANSI/ISO C Specification Language." In: (2008).

[27] S. Bensalem, S. Graf, and Y. Lakhnech. "Abstraction as the Key for Invariant Verification." In: *Verification: Theory and Practice*. Vol. 2772. LNCS. Springer, 2003, pp. 67–99.

[28] D. Beyer, A. Cimatti, A. Griggio, M. E. Keremoglu, and R. Sebastiani. "Software model checking via large-block encoding." In: *Proc. FMCAD*. IEEE, 2009, pp. 25–32.

[29] D. Beyer, M. Dangl, D. Dietsch, M. Heizmann, and A. Stahlbauer. "Witness Validation and Stepwise Testification across Software Verifiers." In: *Proc. ESEC/FSE*. ACM, 2015, pp. 721–733.

[30] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler. "Conditional model checking: A technique to pass information between verifiers." In: *Proc. FSE*. ACM, 2012, p. 57.

[31] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis." In: *Proc. CAV*. Springer, 2007, pp. 504–518.

[32] D. Beyer, T. A. Henzinger, and G. Théoduloz. "Program Analysis with Dynamic Precision Adjustment." In: *Proc. ASE*. IEEE, 2008, pp. 29–38.

[33] D. Beyer, A. Holzer, M. Tautschnig, and H. Veith. “Information Reuse for Multi-goal Reachability Analyses.” In: *Proc. ESOP*. Vol. 7792. LNCS. Springer, 2013, pp. 472–491.

[34] D. Beyer and M. E. Keremoglu. “CPAchecker: A Tool for Configurable Software Verification.” In: *Proc. CAV*. Springer, 2011, pp. 184–190.

[35] D. Beyer, M. E. Keremoglu, and P. Wendler. “Predicate abstraction with adjustable-block encoding.” In: *Proc. FMCAD*. IEEE, 2010, pp. 189–197.

[36] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler. “Precision Reuse for Efficient Regression Verification.” In: *Proc. ESEC/FSE*. ACM, 2013, pp. 389–399.

[37] D. Beyer and A. Stahlbauer. “BDD-based Software Verification - Applications to Event-Condition-Action Systems.” In: *STTT* 16.5 (2014), pp. 507–518.

[38] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. “Generating Tests from Counterexamples.” In: *Proc. ICSE*. 2004, pp. 326–335.

[39] D. Beyer, A. Chlipala, T. A. Henzinger, R. Jhala, and R. Majumdar. “The Blast Query Language for Software Verification.” In: *Proc. SAS*. Vol. 3148. LNCS. Springer, 2004, pp. 2–18.

[40] D. Beyer, M. Dangl, D. Dietsch, and M. Heizmann. “Correctness witnesses: exchanging verification results between verifiers.” In: *Proc. FSE*. ACM, 2016, pp. 326–337.

[41] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar. “The software model checker Blast.” In: *STTT* 9.5-6 (2007), pp. 505–525.

[42] D. Beyer and A. K. Petrenko. “Linux Driver Verification - (Position Paper).” In: *Proc. ISoLA (2)*. Vol. 7610. LNCS. Springer, 2012, pp. 1–6.

[43] D. Beyer and P. Wendler. “Algorithms for software model checking: Predicate abstraction vs. Impact.” In: *Proc. FMCAD*. IEEE, 2012, pp. 106–113.

[44] D. Beyer and P. Wendler. “Reuse of Verification Results: Conditional Model Checking, Precision Reuse, and Verification Witnesses.” In: *Proc. SPIN*. LNCS 7976. Springer, 2013, pp. 1–17.

[45] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. “Symbolic Model Checking without BDDs.” In: *Proc. TACAS*. LNCS 1579. Springer, 1999, pp. 193–207.

[46] G. Birkhoff. “On the structure of abstract algebras.” In: *Mathematical proceedings of the Cambridge philosophical society*. Vol. 31. 4. Cambridge University Press. 1935, pp. 433–454.

[47] H. Björklund and W. Martens. “The tractability frontier for NFA minimization.” In: *J. Comput. Syst. Sci.* 78.1 (2012), pp. 198–210.

[48] P. Boonstoppel, C. Cadar, and D. R. Engler. “RWset: Attacking Path Explosion in Constraint-Based Test Generation.” In: *Proc. TACAS*. Vol. 4963. LNCS. Springer, 2008, pp. 351–366.

[49] V. Botbol, E. Chailloux, and T. L. Gall. "Static Analysis of Communicating Processes Using Symbolic Transducers." In: *Proc. VMCAI*. Vol. 10145. LNCS. Springer, 2017, pp. 73–90.

[50] J. P. Bowen. "The Ethics of Safety-Critical Systems." In: *Commun. ACM* 43.4 (2000), pp. 91–97.

[51] A. R. Bradley. "SAT-based model checking without unrolling." In: *Proc. VMCAI*. LNCS 6538. Springer, 2011, pp. 70–87.

[52] M. J. J. Branco and J. Pin. "Equations Defining the Polynomial Closure of a Lattice of Regular Languages." In: *Proc. ICALP (2)*. Vol. 5556. LNCS. Springer, 2009, pp. 115–126.

[53] S. D. Brookes. "On the Relationship of CCS and CSP." In: *Proc. ICALP*. Vol. 154. LNCS. Springer, 1983, pp. 83–96.

[54] M. C. Browne. "An Improved Algorithm for the Automatic Verification of Finite State Systems Using Temporal Logic." In: *Proc. LICS*. IEEE, 1986, pp. 260–266.

[55] G. Bruns and P. Godefroid. "Model Checking Partial State Spaces with 3-Valued Temporal Logics." In: *Proc. CAV*. Vol. 1633. LNCS. Springer, 1999, pp. 274–287.

[56] R. E. Bryant. "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams." In: *ACM Comput. Surv.* 24.3 (1992), pp. 293–318.

[57] J. A. Brzozowski. "Derivatives of Regular Expressions." In: *J. ACM* 11.4 (1964), pp. 481–494.

[58] T. Bultan, F. Yu, M. Alkhalaf, and A. Aydin. *String Analysis for Software Verification and Security*. Springer, 2017.

[59] S. Chaki, E. M. Clarke, A. Groce, and O. Strichman. "Predicate Abstraction with Minimum Predicates." In: *Proc. CHARME*. LNCS 2860. Springer, 2003, pp. 19–34.

[60] S. Chaki, A. Groce, and O. Strichman. "Explaining abstract counterexamples." In: *Proc. FSE*. ACM, 2004, pp. 73–82.

[61] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. "State/Event-Based Software Model Checking." In: *Proc. IFM*. Vol. 2999. LNCS. Springer, 2004, pp. 128–147.

[62] G. Chen, H. Jin, D. Zou, Z. Liang, B. B. Zhou, and H. Wang. "A Framework for Practical Dynamic Software Updating." In: *IEEE Trans. Parallel Distrib. Syst.* 27.4 (2016), pp. 941–950.

[63] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. "Incremental formal verification of hardware." In: *Proc. FMCAD*. 2011, pp. 135–143.

[64] A. Cimatti, A. Griggio, A. Micheli, I. Narasamdya, and M. Roveri. " Kratos: A Software Model Checker for SystemC." In: *Proc. CAV*. LNCS 6806. Springer, 2011, pp. 310–316.

[65] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. "The MathSAT5 SMT Solver." In: *Proc. TACAS*. LNCS 7795. Springer, 2013, pp. 93–107.

[66] A. Cimatti and A. Griggio. "Software Model Checking via IC3." In: *Proc. CAV*. Vol. 7358. LNCS. Springer, 2012, pp. 277–293.

[67] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. "Counterexample-Guided Abstraction Refinement." In: *Proc. CAV*. LNCS 1855. Springer, 2000, pp. 154–169.

[68] E. M. Clarke, O. Grumberg, M. Talupur, and D. Wang. "High Level Verification of Control Intensive Systems Using Predicate Abstraction." In: *Proc. MEMOCODE*. IEEE, 2003, pp. 55–64.

[69] E. M. Clarke, E. A. Emerson, and A. P. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications." In: *ACM Trans. Program. Lang. Syst.* 8.2 (1986), pp. 244–263.

[70] E. M. Clarke, O. Grumberg, and D. E. Long. "Model Checking and Abstraction." In: *Proc. POPL*. ACM Press, 1992, pp. 342–354.

[71] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2001.

[72] E. M. Clarke, O. Grumberg, M. Talupur, and D. Wang. "Making Predicate Abstraction Efficient: How to Eliminate Redundant Predicates." In: *Proc. CAV*. Vol. 2725. LNCS. Springer, 2003, pp. 126–140.

[73] E. M. Clarke, T. A. Henzinger, H. Veith, and R. Bloem, eds. *Handbook of Model Checking*. Springer, 2018.

[74] M. R. Clarkson and F. B. Schneider. "Hyperproperties." In: *Journal of Computer Security* 18.6 (2010), pp. 1157–1210.

[75] T. R. Colburn and G. M. Shute. "Abstraction in Computer Science." In: *Minds and Machines* 17.2 (2007), pp. 169–184.

[76] I. E. Commission et al. "Functional safety of electrical/electronic/programmable electronic safety-related systems-Part 5, Examples of methods for the determination of safety integrity levels." In: *IEC 61508-5* (1998).

[77] B. Cook, A. Podelski, and A. Rybalchenko. "Termination proofs for systems code." In: *Proc. PLDI*. ACM, 2006, pp. 415–426.

[78] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[79] A. Cortesi, G. Costantini, and P. Ferrara. "A Survey on Product Operators in Abstract Interpretation." In: *Festschrift for Dave Schmidt*. Vol. 129. EPTCS. 2013, pp. 325–336.

[80] P. Cousot and R. Cousot. "Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints." In: *Proc. POPL*. ACM, 1977, pp. 238–252.

[81] P. Cousot and R. Cousot. "Systematic design of program-analysis frameworks." In: *Proc. POPL*. ACM, 1979, pp. 269–282.

[82] P. Cousot. "Abstract interpretation based formal methods and future challenges." In: *Informatics*. Springer. 2001, pp. 138–156.

[83] P. Cousot and R. Cousot. "Modular Static Program Analysis." In: *Proc. CC*. Vol. 2304. LNCS. Springer, 2002, pp. 159–178.

[84] W. Craig. "Three Uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory." In: *J. Symb. Log.* 22.3 (1957), pp. 269–285.

[85] L. D'Antoni and M. Veanes. "Minimization of symbolic automata." In: *Proc. POPL*. ACM, 2014, pp. 541–554.

[86] L. D'Antoni and M. Veanes. "Forward Bisimulations for Nondeterministic Symbolic Finite Automata." In: *Proc. TACAS*. LNCS 10205. 2017, pp. 518–534.

[87] L. D'Antoni and M. Veanes. "The Power of Symbolic Automata and Transducers." In: *Proc. CAV*. LNCS 10426. Springer, 2017, pp. 47–67.

[88] L. D'Antoni and M. Veanes. "Extended symbolic finite automata and transducers." In: *Formal Methods in System Design* 47.1 (2015), pp. 93–119.

[89] D. R. Dams, R. Gerth, and O. Grumberg. "Abstract Interpretation of Reactive Systems." In: *ACM Transactions on Programming Languages and Systems* 19.2 (1997), pp. 253–291.

[90] M. Delahaye, N. Kosmatov, and J. Signoles. "Common specification language for static and dynamic analysis of C programs." In: *Proc. SAC*. ACM, 2013, pp. 1230–1235.

[91] Y. Demyanova, P. Rümmer, and F. Zuleger. "Systematic Predicate Abstraction Using Variable Roles." In: *Proc. NFM*. Springer. 2017, pp. 265–281.

[92] N. Dershowitz. "Orderings for Term-Rewriting Systems." In: *Proc. FOCS*. IEEE, 1979, pp. 123–131.

[93] D. Dietsch, M. Heizmann, B. Musa, A. Nutz, and A. Podelski. "Craig vs. Newton in software model checking." In: *Proc. ESEC/FSE*. ACM, 2017, pp. 487–497.

[94] E. W. Dijkstra. "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." In: *Commun. ACM* 18.8 (1975), pp. 453–457.

[95] E. W. Dijkstra and C. S. Scholten. *Predicate Calculus and Program Semantics*. Texts and Monographs in Computer Science. Springer, 1990.

[96] E. W. Dijkstra. *Notes on Structured Programming*. 1970.

[97] I. Dillig, T. Dillig, B. Li, and K. L. McMillan. "Inductive Invariant Generation via Abductive Inference." In: *Proc. OOPSLA*. ACM, 2013, pp. 443–456.

[98] R. Douence, P. Fradet, M. Südholt, et al. "Trace-based aspects." In: *Aspect-Oriented Software Development* (2004), pp. 201–217.

[99] D. Duffus, B. Jónsson, and I. Rival. "Structure Results for Function Lattices." In: *Canadian Journal of Mathematics* 30.2 (1978), 392–400.

[100] J. Engelfriet. "Top-down Tree Transducers with Regular Look-ahead." In: *Mathematical Systems Theory* 10 (1977), pp. 289–303.

[101] A. Engels, L. M. G. Feijs, and S. Mauw. "Test Generation for Intelligent Networks Using Model Checking." In: *Proc. TACAS*. Vol. 1217. LNCS. Springer, 1997, pp. 384–398.

[102] D. R. Engler and M. Musuvathi. "Static Analysis versus Software Model Checking for Bug Finding." In: *Proc. VMCAI*. LNCS 2937. Springer, 2004, pp. 191–210.

[103] A. Ferguson and J. Hughes. "An Iterative Powerdomain Construction." In: *Functional Programming*. Workshops in Computing. Springer, 1989, pp. 41–55.

[104] G. Filé, R. Giacobazzi, and F. Ranzato. "A Unifying View of Abstract Domain Design." In: *ACM Comput. Surv.* 28.2 (1996), pp. 333–336.

[105] C. Flanagan and S. Qadeer. "Predicate abstraction for software verification." In: *Proc. POPL*. ACM, 2002, pp. 191–202.

[106] R. W. Floyd. "Assigning meanings to programs." In: *Mathematical Aspects of Computer Science*. AMS, 1967, pp. 19–32.

[107] P. Fradet and M. Südholt. "Towards a Generic Framework for AOP." In: *Proc. ECOOP*. Springer, 1998, pp. 394–397.

[108] P. Fradet and M. Südholt. "An Aspect Language for Robust Programming." In: *Proc. ECOOP Workshops*. Vol. 1743. LNCS. Springer, 1999, pp. 291–292.

[109] G. Fraser and F. Wotawa. "Using Model-Checkers for Mutation-Based Test-Case Generation, Coverage Analysis and Specification Analysis." In: *Proc. ICSEA*. IEEE, 2006, p. 16.

[110] D. D. Freydenberger. "Extended Regular Expressions: Succinctness and Decidability." In: *Theory Comput. Syst.* 53.2 (2013), pp. 159–193.

[111] C. A. Furia, B. Meyer, and S. Velder. "Loop invariants: Analysis, classification, and examples." In: *ACM Comput. Surv.* 46.3 (2014), 34:1–34:51.

[112] T. L. Gall and B. Jeannet. "Lattice Automata: A Representation for Languages on Infinite Alphabets, and Some Applications to Verification." In: *Proc. SAS*. Vol. 4634. LNCS. Springer, 2007, pp. 52–68.

[113] V. K. Garg. *Introduction to lattice theory with computer science applications*. Wiley, 2015.

[114] A. Gargantini and C. L. Heitmeyer. "Using Model Checking to Generate Tests from Requirements Specifications." In: *Proc. ESEC/FSE*. 1999, pp. 146–162.

[115] P. Gastin and D. Oddoux. "Fast LTL to Büchi Automata Translation." In: *Proc. CAV*. Vol. 2102. LNCS. Springer, 2001, pp. 53–65.

[116] M. Gehrke, S. Grigorieff, and J. Pin. "Duality and Equational Theory of Regular Languages." In: *Proc. ICALP (2)*. Vol. 5126. LNCS. Springer, 2008, pp. 246–257.

[117] D. Giannakopoulou, K. S. Namjoshi, and C. S. Pasareanu. "Compositional Reasoning." In: *Handbook of Model Checking*. Springer, 2018, pp. 345–383.

[118] A. M. Glenberg, M. Meyer, and K. Lindem. "Mental Models Contribute to Foregrounding during Text Comprehension." In: *Journal of Memory and Language* 26.1 (1987). Last updated - 2013-02-23, p. 69.

[119] P. Godefroid and R. Jagadeesan. "Automatic Abstraction Using Generalized Model Checking." In: *Proc. CAV*. Vol. 2404. LNCS. Springer, 2002, pp. 137–150.

[120] P. Godefroid and N. Piterman. "LTL generalized model checking revisited." In: *STTT* 13.6 (2011), pp. 571–584.

[121] K. Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I." In: *Monatshefte für Mathematik und Physik* 38.1 (1931), pp. 173–198.

[122] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS." In: *Proc. CAV*. Springer, 1997, pp. 72–83.

[123] G. Grätzer. *Lattice Theory: Foundation*. Birkhäuser, 2011.

[124] A. Gupta and A. Rybalchenko. "InvGen: An Efficient Invariant Generator." In: *Proc. CAV*. Springer, 2009, pp. 634–640.

[125] M. Hamana. "Term rewriting with variable binding: an initial algebra approach." In: *Proc. PPDP*. ACM, 2003, pp. 148–159.

[126] R. Hamlet. "Test reliability and software maintenance." In: *Proc. COMPSAC*. IEEE, 1978, pp. 315–320.

[127] G. Hamon, L. M. de Moura, and J. M. Rushby. "Generating Efficient Test Sets with a Model Checker." In: *Proc. SEFM*. 2004, pp. 261–270.

[128] T. Hansen, P. Schachte, and H. Søndergaard. "State Joining and Splitting for the Symbolic Execution of Binaries." In: *Proc. RV*. Vol. 5779. LNCS. Springer, 2009, pp. 76–92.

[129] R. H. Hardin, R. P. Kurshan, K. L. McMillan, J. A. Reeds, and N. J. A. Sloane. "Efficient Regression Verification." In: *Proc. WODES*. 1996, pp. 147–150.

[130] J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[131] J. B. Hart and C. Tsinakis. "A concrete realization of the Hoare powerdomain." In: *Soft Comput.* 11.11 (2007), pp. 1059–1063.

[132] M. Heizmann, J. Christ, D. Dietsch, J. Hoenicke, M. Lindenmann, B. Musa, C. Schilling, S. Wissert, and A. Podelski. "Ultimate Automizer with Unsatisfiable Cores (Competition Contribution)." In: *Proc. TACAS*. LNCS 8413. Springer, 2014, pp. 418–420.

[133] M. Heizmann, J. Hoenicke, and A. Podelski. "Refinement of Trace Abstraction." In: *Proc. SAS*. Vol. 5673. LNCS. Springer, 2009, pp. 69–85.

[134] M. Heizmann, J. Hoenicke, and A. Podelski. "Software Model Checking for People Who Love Automata." In: *Proc. CAV*. Vol. 8044. LNCS. Springer, 2013, pp. 36–52.

[135] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. A. Sanvido. "Extreme Model Checking." In: *Verification: Theory and Practice*. 2003, pp. 332–358.

[136] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. "Lazy abstraction." In: *Proc. POPL*. ACM, 2002, pp. 58–70.

[137] T. A. Henzinger, R. Jhala, R. Majumdar, and K. L. McMillan. "Abstractions from proofs." In: *Proc. POPL*. ACM, 2004, pp. 232–244.

[138] C. A. R. Hoare. "An Axiomatic Basis for Computer Programming." In: *Commun. ACM* 12.10 (1969), pp. 576–580.

[139] C. A. R. Hoare. "The verifying compiler: A grand challenge for computing research." In: *J. ACM* 50.1 (2003), pp. 63–69.

[140] T. Hoare and J. Misra. *Verified software: theories, tools, experiments; Vision of a Grand Challenge project*. 2005.

[141] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. "FShell: Systematic Test Case Generation for Dynamic Analysis and Measurement." In: *Proc. CAV*. Vol. 5123. LNCS. Springer, 2008, pp. 209–213.

[142] A. Holzer, C. Schallhart, M. Tautschnig, and H. Veith. "Query-Driven Program Testing." In: *Proc. VMCAI*. Vol. 5403. LNCS. Springer, 2009, pp. 151–166.

[143] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation - international edition (2. ed)*. Addison-Wesley, 2003.

[144] S. Horwitz, J. Prins, and T. W. Reps. "Integrating Non-Interfering Versions of Programs." In: *Proc. POPL*. ACM Press, 1988, pp. 133–145.

[145] P. Hui and J. Riely. "Temporal aspects as security automata." In: *Foundations of Aspect-Oriented Languages (FOAL 2006), Workshop at AOSD 2006*. 2006, pp. 06–01.

[146] E. V. Huntington. "Sets of Independent Postulates for the Algebra of Logic." In: *Transactions of the American Mathematical Society* 5.3 (1904), pp. 288–309. ISSN: 00029947.

[147] H. Jain, F. Ivancic, A. Gupta, and M. K. Ganai. "Localization and Register Sharing for Predicate Abstraction." In: *Proc. TACAS*. LNCS 3440. Springer, 2005, pp. 397–412.

[148] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. "Using Statically Computed Invariants Inside the Predicate Abstraction and Refinement Loop." In: *Proc. CAV*. LNCS 4144. Springer, 2006, pp. 137–151.

[149] M. Jakobs and H. Wehrheim. "Compact Proof Witnesses." In: *NFM*. Vol. 10227. LNCS. 2017, pp. 389–403.

[150] R. Jhala and R. Majumdar. "Path Slicing." In: *Proc. PLDI*. ACM, 2005, pp. 38–47.

[151] R. Jhala and R. Majumdar. "Software Model Checking." In: *ACM Computing Surveys* 41.4 (2009).

[152] R. Jhala and K. L. McMillan. "A Practical and Complete Approach to Predicate Refinement." In: *Proc. TACAS*. LNCS 3920. Springer, 2006, pp. 459–473.

[153] R. Jhala and K. L. McMillan. "Array Abstractions from Proofs." In: *Proc. CAV*. Vol. 4590. LNCS. Springer, 2007, pp. 193–206.

[154] T. Jiang and B. Ravikumar. "Minimal NFA Problems are Hard." In: *SIAM J. Comput.* 22.6 (1993), pp. 1117–1141.

[155] J. B. Kam and J. D. Ullman. "Monotone Data Flow Analysis Frameworks." In: *Acta Inf.* 7 (1977), pp. 305–317.

[156] D. Kapur. "Automatically Generating Loop Invariants Using Quantifier Elimination." In: *Proc. Deduction and Applications*. Vol. 05431. IBFI Schloss Dagstuhl, 2006.

[157] B. Katz, M. Krug, A. Lochbihler, I. Rutter, G. Snelting, and D. Wagner. "Gateway Decompositions for Constrained Reachability Problems." In: *Proc. SEA*. Vol. 6049. LNCS. Springer, 2010, pp. 449–461.

[158] S. Katz. "Aspect Categories and Classes of Temporal Properties." In: LNCS 3880. Springer, 2006, pp. 106–134.

[159] A. V. Khoroshilov, V. Mutilin, A. Petrenko, and V. Zakharov. "Establishing Linux Driver Verification Process." In: *Proc. Ershov Memorial Conference*. LNCS 5947. Springer, 2009, pp. 165–176.

[160] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. "Aspect-Oriented Programming." In: *Proc. ECOOP*. LNCS 1241. Springer, 1997, pp. 220–242.

[161] J. C. King. "Symbolic execution and program testing." In: *Commun. ACM* 19.7 (1976), pp. 385–394.

[162] G. Klein, J. Andronick, M. Fernandez, I. Kuz, T. C. Murray, and G. Heiser. "Formally verified software in the real world." In: *Commun. ACM* 61.10 (2018), pp. 68–77.

[163] G. Kniesel. "Detection and Resolution of Weaving Interactions." In: *LNCS Trans. Aspect Oriented Softw. Dev.* 5 (2009), pp. 135–186.

[164] S. Ko and Y. Han. "Left is Better than Right for Reducing Nondeterminism of NFAs." In: *Proc. CIAA*. Vol. 8587. LNCS. Springer, 2014, pp. 238–251.

[165] A. Komuravelli, A. Gurfinkel, S. Chaki, and E. M. Clarke. "Automatic Abstraction in SMT-Based Unbounded Software Model Checking." In: *Proc. CAV*. Vol. 8044. LNCS. Springer, 2013, pp. 846–862.

[166] S. Kong, Y. Jung, C. David, B. Wang, and K. Yi. "Automatically Inferring Quantified Loop Invariants by Algorithmic Learning from Simple Templates." In: *Proc. APLAS*. LNCS 6461. Springer, 2010, pp. 328–343.

[167] L. Kovács and A. Voronkov. "Finding Loop Invariants for Programs over Arrays Using a Theorem Prover." In: *Proc. FASE*. Springer, 2009, pp. 470–485.

[168] J. Kramer. "Is abstraction the key to computing?" In: *Commun. ACM* 50.4 (2007), pp. 36–42.

[169] O. Kupferman and Y. Lustig. "Lattice Automata." In: *Proc. VMCAI*. Vol. 4349. LNCS. Springer, 2007, pp. 199–213.

[170] O. Kupferman and M. Y. Vardi. "Model Checking of Safety Properties." In: *Proc. CAV*. Vol. 1633. LNCS. Springer, 1999, pp. 172–183.

[171] O. Kupferman and M. Y. Vardi. “Model Checking of Safety Properties.” In: *Formal Methods in System Design* 19.3 (2001), pp. 291–314.

[172] O. Kupferman and M. Y. Vardi. “Vacuity detection in temporal model checking.” In: *STTT* 4.2 (2003), pp. 224–233.

[173] O. Kupferman, M. Y. Vardi, and P. Wolper. “An automata-theoretic approach to branching-time model checking.” In: *J. ACM* 47.2 (2000), pp. 312–360.

[174] I. Kupka. “Unique Fixpoints in Complete Lattices with Applications to Formal Languages and Semantics.” In: *Foundations of Computer Science: Potential - Theory - Cognition*. Vol. 1337. LNCS. Springer, 1997, pp. 107–115.

[175] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. “Efficient state merging in symbolic execution.” In: *Proc. PLDI*. ACM, 2012, pp. 193–204.

[176] D. Lacey and O. de Moor. “Imperative Program Transformation by Rewriting.” In: *Proc. CC*. Vol. 2027. LNCS. Springer, 2001, pp. 52–68.

[177] S. K. Lahiri, R. Nieuwenhuis, and A. Oliveras. “SMT Techniques for Fast Predicate Abstraction.” In: *Proc. CAV*. LNCS 4144. Springer, 2006, pp. 424–437.

[178] L. Lamport. “Proving the Correctness of Multiprocess Programs.” In: *IEEE Trans. Software Eng.* 3.2 (1977), pp. 125–143.

[179] L. Lamport. “What Good is Temporal Logic?” In: *IFIP Congress*. 1983, pp. 657–668.

[180] L. Lamport. “Composition: A Way to Make Proofs Harder.” In: *Proc. COMPOS*. LNCS 1536. Springer, 1997, pp. 402–423.

[181] G. A. Lanzarone and M. Ornaghi. “Program Construction by Refinements Preserving Correctness.” In: *Comput. J.* 18.1 (1975), pp. 55–62.

[182] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan. “Incremental state-space exploration for programs with dynamically allocated data.” In: *Proc. ICSE*. ACM, 2008, pp. 291–300.

[183] M. Leucker, G. Markin, and M. R. Neuhäußer. “A New Refinement Strategy for CEGAR-Based Industrial Model Checking.” In: *Proc. HVC*. Vol. 9434. LNCS. Springer, 2015, pp. 155–170.

[184] J. Liebig, A. v. Rhein, C. Kästner, S. Apel, J. Dörre, and C. Lengauer. “Scalable analysis of variable software.” In: *Proc. ESEC/FSE*. ACM, 2013, pp. 81–91.

[185] C. Löding and A. Tollkötter. “Transformation Between Regular Expressions and omega-Automata.” In: *MFCS*. Vol. 58. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016, 88:1–88:13.

[186] Y. Lustig and M. Y. Vardi. “Synthesis from component libraries.” In: *STTT* 15.5-6 (2013), pp. 603–618.

[187] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems - specification*. Springer, 1992.

[188] I. Marshall. “The Production of Acoustic Impulses in Air.” In: *Measurement Science and Technology* 1.5 (1990), p. 413.

[189] A. W. Mazurkiewicz. “Trace Theory.” In: *Advances in Petri Nets*. Vol. 255. LNCS. Springer, 1986, pp. 279–324.

[190] K. L. McMillan. “Interpolation and SAT-Based Model Checking.” In: *Proc. CAV*. LNCS 2725. Springer, 2003, pp. 1–13.

[191] K. L. McMillan. “Lazy Abstraction with Interpolants.” In: *Proc. CAV*. LNCS 4144. Springer, 2006, pp. 123–136.

[192] K. L. McMillan. “Interpolation and Model Checking.” In: *Handbook of Model Checking*. Springer, 2018, pp. 421–446.

[193] G. H. Mealy. “A method for synthesizing sequential circuits.” In: *The Bell System Technical Journal* 34.5 (1955), pp. 1045–1079.

[194] F. Merz, S. Falke, and C. Sinz. “LLBMC: Bounded Model Checking of C and C++ Programs Using a Compiler IR.” In: *Proc. VSTTE*. LNCS 7152. Springer, 2012, pp. 146–161.

[195] J. Midtgaard, F. Nielson, and H. R. Nielson. “A Parametric Abstract Domain for Lattice-Valued Regular Expressions.” In: *Proc. SAS*. Vol. 9837. LNCS. Springer, 2016, pp. 338–360.

[196] E. F. Moore. “Gedanken-experiments on sequential machines.” In: *Automata studies* 34 (1956), pp. 129–153.

[197] N. Moray. “A lattice theory approach to the structure of mental models.” In: *Philosophical Transactions of the Royal Society of London B: Biological Sciences* 327.1241 (1990), pp. 577–583. ISSN: 0080-4622.

[198] S. Muggleton. “Predicate invention and utilization.” In: *J. Exp. Theor. Artif. Intell.* 6.1 (1994), pp. 121–130.

[199] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. “ Cmc: A Pragmatic Approach to Model Checking Real Code.” In: *Proc. OSDI*. USENIX, 2002.

[200] G. J. Myers. *The Art of Software Testing*. Wiley, 1979.

[201] P. Naur and B. Randell. “Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th-11th October 1968.” In: (1969).

[202] P. P. Nayak and A. Y. Levy. “A Semantic Theory of Abstractions.” In: *IJCAI*. Morgan Kaufmann, 1995, pp. 196–203.

[203] I. Neamtiu, M. W. Hicks, G. P. Stoyle, and M. Oriol. “Practical dynamic software updating for C.” In: *Proc. PLDI*. ACM, 2006, pp. 72–83.

[204] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[205] T. Nipkow and C. Prehofer. “Higher-order rewriting and equational reasoning.” In: *Automated Deduction—A Basis for Applications* 1 (1998), pp. 399–430.

[206] G. van Noord and D. Gerdemann. “Finite State Transducers with Predicates and Identities.” In: *Grammars* 4.3 (2001), pp. 263–286.

[207] T. Oonishi, P. R. Dixon, K. Iwano, and S. Furui. “Generalization of specialized on-the-fly composition.” In: *Proc. ICASSP*. IEEE, 2009, pp. 4317–4320.

[208] S. Ott. “Implementing a Termination Analysis using Configurable Program Analysis.” 2016.

[209] *Oxford Dictionary of English*. Oxford University Press, 2010.

[210] O. Padon, N. Immerman, S. Shoham, A. Karbyshev, and M. Sagiv. “Decidability of inferring inductive invariants.” In: *Proc. POPL*. ACM, 2016, pp. 217–231.

[211] D. Peled. “Partial-Order Reduction.” In: *Handbook of Model Checking*. Springer, 2018, pp. 173–190.

[212] S. Pinchinat and H. Marchand. “Symbolic abstractions of automata.” In: *Discrete Event Systems*. Springer, 2000, pp. 39–48.

[213] N. Pippenger. “Regular Languages and Stone Duality.” In: *Theory Comput. Syst.* 30.2 (1997), pp. 121–134.

[214] N. Piterman and A. Pnueli. “Temporal Logic and Fair Discrete Systems.” In: *Handbook of Model Checking*. Springer, 2018, pp. 27–73.

[215] G. D. Plotkin. “A structural approach to operational semantics.” In: *J. Log. Algebr. Program.* 60-61 (2004), pp. 17–139.

[216] A. Pnueli and R. Rosner. “On the Synthesis of a Reactive Module.” In: *Proc. POPL*. 1989, pp. 179–190.

[217] M. D. Preda, R. Giacobazzi, A. Lakhotia, and I. Mastroeni. “Abstract Symbolic Automata: Mixed syntactic/semantic similarity analysis of executables.” In: *Proc. POPL*. ACM, 2015, pp. 329–341.

[218] M. D. Preda, R. Giacobazzi, and I. Mastroeni. “Completeness in Approximate Transduction.” In: *Proc. SAS*. Vol. 9837. LNCS. Springer, 2016, pp. 126–146.

[219] C. Prehofer. “Feature-Oriented Programming: A Fresh Look at Objects.” In: *Proc. ECOOP*. Vol. 1241. LNCS. Springer, 1997, pp. 419–443.

[220] G. Ramalingam. “On loops, dominators, and dominance frontiers.” In: *ACM Trans. Program. Lang. Syst.* 24.5 (2002), pp. 455–490.

[221] S. Rayadurgam and M. P. E. Heimdahl. “Coverage Based Test-Case Generation Using Model Checkers.” In: *ECBS*. IEEE, 2001, p. 83.

[222] A. von Rhein, T. Thüm, I. Schaefer, J. Liebig, and S. Apel. “Variability encoding: From compile-time to load-time variability.” In: *J. Log. Algebr. Meth. Program.* 85.1 (2016), pp. 125–145.

[223] X. Rival and L. Mauborgne. “The trace partitioning abstract domain.” In: *ACM Trans. Program. Lang. Syst.* 29.5 (2007), p. 26.

[224] H. Saïdi. “Model Checking Guided Abstraction and Analysis.” In: *Proc. SAS*. Vol. 1824. LNCS. Springer, 2000, pp. 377–396.

[225] C. Sammut and G. I. Webb, eds. *Encyclopedia of Machine Learning and Data Mining*. Springer, 2017.

[226] I. Schaefer, L. Bettini, V. Bono, F. Damiani, and N. Tanzarella. "Delta-Oriented Programming of Software Product Lines." In: *Proc. SPLC*. Vol. 6287. LNCS. Springer, 2010, pp. 77–91.

[227] D. A. Schmidt and B. Steffen. "Program Analysis as Model Checking of Abstract Interpretations." In: *Proc. SAS*. LNCS 1503. Springer, 1998, pp. 351–380.

[228] F. B. Schneider. "Enforceable security policies." In: *ACM Trans. Inf. Syst. Secur.* 3.1 (2000), pp. 30–50.

[229] V. Schuppan and A. Biere. "Liveness Checking as Safety Checking for Infinite State Spaces." In: *Electr. Notes Theor. Comput. Sci.* 149.1 (2006), pp. 79–96.

[230] M. N. Seghir and D. Kroening. "Counterexample-Guided Precondition Inference." In: *Proc. ESOP*. Vol. 7792. LNCS. Springer, 2013, pp. 451–471.

[231] M. N. Seghir and P. Schrammel. "Necessary and Sufficient Preconditions via Eager Abstraction." In: *Proc. APLAS*. Springer, 2014, pp. 236–254.

[232] O. Sery. "Enhanced Property Specification and Verification in Blast." In: *Proc. FASE*. LNCS 5503. 2009, pp. 456–469.

[233] O. Sery, G. Fedyukovich, and N. Sharygina. "Incremental Upgrade Checking by Means of Interpolation-based Function Summaries." In: *Proc. FMCAD*. FMCAD, 2012, pp. 114–121.

[234] R. Sharma, A. V. Nori, and A.Aiken. "Bias-variance tradeoffs in program analysis." In: *Proc. POPL*. ACM, 2014, pp. 127–138.

[235] N. Sharygina, J. C. Browne, F. Xie, R. P. Kurshan, and V. Levin. "Lessons Learned from Model Checking a NASA Robot Controller." In: *Formal Methods in System Design* 25.2-3 (2004), pp. 241–270.

[236] G. Singh, M. Püschel, and M. T. Vechev. "Fast polyhedra abstract domain." In: *Proc. POPL*. ACM, 2017, pp. 46–59.

[237] M. Sipser. *Introduction to the Theory of Computation*. PWS Publishing Company, 1997.

[238] S. Srivastava and S. Gulwani. "Program verification using templates over predicate abstraction." In: *Proc. PLDI*. ACM, 2009, pp. 223–234.

[239] D. Statter and M. Armoni. "Learning Abstraction in Computer Science: A Gender Perspective." In: *Proc. WiPSCE*. ACM, 2017, pp. 5–14.

[240] M. H. Stone. "The Theory of Representation for Boolean Algebras." In: *Transactions of the American Mathematical Society* 40.1 (1936), pp. 37–111. ISSN: 00029947.

[241] M. Tautschnig. "Query-Driven Program Testing." PhD thesis. Universität Wien, 2011.

[242] N. Timm, H. Wehrheim, and M. Czech. "Heuristic-Guided Abstraction Refinement for Concurrent Systems." In: *Proc. ICFEM*. Vol. 7635. LNCS. Springer, 2012, pp. 348–363.

[243] M. Y. Vardi. "An Automata-Theoretic Approach to Linear Temporal Logic." In: *Logics for Concurrency - Structure versus Automata (Proc. Banff'95)*. LNCS 1043. Springer, 1996, pp. 238–266.

[244] M. Veanes. "Applications of Symbolic Finite Automata." In: *Proc. CIAA*. LNCS 7982. Springer, 2013, pp. 16–23.

[245] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjørner. "Symbolic finite state transducers: algorithms and applications." In: *Proc. POPL*. ACM, 2012, pp. 137–150.

[246] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer." In: *Formal Methods and Testing*. Vol. 4949. LNCS. Springer, 2008, pp. 39–76.

[247] M. Veanes, C. Campbell, and W. Schulte. "Composition of Model Programs." In: *Proc. FORTE*. Vol. 4574. LNCS. Springer, 2007, pp. 128–142.

[248] W. Visser, J. Geldenhuys, and M. B. Dwyer. "Green: Reducing, reusing and recycling constraints in program analysis." In: *Proc. FSE*. ACM, 2012.

[249] H. Wehrheim. "Bounded Model Checking for Partial Kripke Structures." In: *Proc. ICTAC*. Vol. 5160. LNCS. Springer, 2008, pp. 380–394.

[250] P. M. Whitman. "Lattices, equivalence relations, and subgroups." In: *Bulletin of the American Mathematical Society* 52.6 (1946), pp. 507–522.

[251] J. Winkowski and A. Maggiolo-Schettini. "An Algebra of Processes." In: *J. Comput. Syst. Sci.* 35.2 (1987), pp. 206–228.

[252] Y. Xie and A. Aiken. "Context- and path-sensitive memory leak detection." In: *Proc. ESEC/FSE*. ACM, 2005, pp. 115–125.

[253] G. Yang, C. S. Păsăreanu, and S. Khurshid. "Memoized symbolic execution." In: *Proc. ISSTA*. ACM, 2012, pp. 144–154.

[254] J. Yang, P. Twohey, D. R. Engler, and M. Musuvathi. "Using model checking to find serious file system errors." In: *ACM Trans. Comput. Syst.* 24.4 (2006), pp. 393–423.

[255] K. Yi and W. L. H. III. "Automatic Generation and Management of Interprocedural Program Analyses." In: *Proc. POPL*. ACM Press, 1993, pp. 246–259.

[256] F. Yu, T. Bultan, and O. H. Ibarra. "Relational String Verification Using Multi-Track Automata." In: *Int. J. Found. Comput. Sci.* 22.8 (2011), pp. 1909–1924.

COLOPHON

Version as of September 11, 2020.