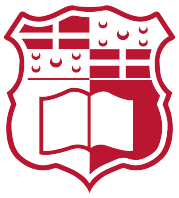


Residual-Based Combination of Static and Runtime Verification

SHAUN AZZOPARDI

Supervised by Gordon J. Pace

Department of Computer Science
Faculty of ICT
University of Malta

June, 2019

A dissertation submitted in partial fulfilment of the requirements for the degree of PhD in Computer Science.

Statement of Originality

I, the undersigned, declare that this is my own work unless where otherwise acknowledged and referenced.

Candidate Shaun Azzopardi

Signed _____

Date June 14, 2019

To my family

Acknowledgements

I would first like to thank Gordon Pace and Christian Colombo, without whose support and insight the work presented here would not be possible.

This work was initially inspired by Eric Bodden's work in the combination of static and runtime verification for finite-state properties. Insights gained from exploring this work aided me greatly in the contributions presented here.

Moreover, the prototypes implementing some contributions here depend on the LARVA and the CONTRACTLARVA tool and their specification languages. Gratitude goes to contributors to their development, especially Gordon Pace and Christian Colombo. The Soot framework, developed by the Sable Research Group and other contributors, was also essential to implement a prototype implementation of our analysis.

I am grateful for my board of examiners including Kevin Vella, Mark Micallef, Adrian Francalanza, and Borzoo Bonakdarpour for their suggestions and advice, and Stephanie Abood for her help on the administrative side. I am indebted to the Department of Computer Science's faculty who provided feedback and advice during presentation of early versions of this work at the department's workshops. I would also like to thank JP Ebejer for the template[1] this document is written in.

I am most of all indebted to my family for their support throughout these past years, especially my mum Grace and dad Frank, my siblings Stephanie, Charles, Brandon and Alex, my aunts Margaret and Michelina, and finally my nanna Carm) .

[1] `https://github.com/jp-um/university_of_malta_LaTeX_dissertation_template`

Abstract

While verification techniques attempt to solve the problem of checking that a program satisfies a property they can fail and give an indeterminate verdict. In a survey of literature we identify approaches that improve this by allowing a verification technique to produce a new proof obligation by either transforming a property and/or transforming a program. Here we choose to focus on techniques that transform a property. To reason about these we present an abstract formal framework that characterises properties in terms of the programs that satisfy them. In turn we are able to characterise a *residual property* that is equivalent to the original property for the program under verification. We instantiate this approach for both state- and event-based properties, showing how the conjunction of properties can be wielded to produce effective residual properties to reduce the verification effort for subsequent techniques.

We validate further this approach for state-based properties in the context of industrial project involving a payments ecosystem and untrusted client applications, where a enforced model of behaviour allows us to verify or partially evaluate at pre-deployment certain regulations specified as universally quantified propositions.

Our main contribution is an approach for the combination of static and runtime verification for automata-based properties. Existing such approaches focus on the reduction of instrumentation of a program, while we present an approach to the reduction of structural elements of a property. We focus on *dynamic event automata* (DEA) as event-based properties that monitor both the program control-flow and the program data-state. We give an intraprocedural approach to analysing a property against a program. This allows us to collect knowledge about the behaviour of each procedure individually (ignoring possible outside behaviour) to produce a residual of the whole-program. The algorithm can easily be adapted to the interprocedural case. Moreover, by adding an assertion propagation algorithm and the use of an SMT solver we show how this can be optimised by pruning the possible behaviour and evaluating property guards. We validated this approach on both Java programs and Solidity smart contracts, showing moderate overhead improvements but significant progress in reducing the property.

Contents

List of Figures

List of Tables

Introduction

0.1 Motivation

Verification methods attempt to check whether a program satisfies a specification. However, one is not assured that in general such a method will always succeed in giving a verdict. There are two main reasons for this:

(i) theoretical limits on what the chosen method can prove; and

(ii) practical limits on the amount of resources (including time and memory) that are available for verification.

Combining different verification techniques with different power and at different stages of deployment can push further back these limits. To illustrate this we give a background of the limitations of different verification techniques at a high-level, after which we give a summary of existing work in combining different techniques.

0.1.1 Background

Ideally that a program is proven compliant with a specification is confirmed *pre-deployment* to ensure the program's well-behaviour *post-deployment*. However pre-deployment verification can sometimes fail, requiring post-deployment verification.

L-Università ta' Malta

University of Malta Library – Electronic Thesis & Dissertations (ETD) Repository

The copyright of this thesis/dissertation belongs to the author. The author's rights in respect of this work are as defined by the Copyright Act (Chapter 415) of the Laws of Malta or as modified by any successive legislation.

Users may access this full-text thesis/dissertation and can make use of the information contained in accordance with the Copyright Act provided that the author must be properly acknowledged. Further distribution or reproduction in any format is prohibited without the prior permission of the copyright holder.

0.1.1.1 Pre-deployment Verification

Two central approaches to pre-deployment verification are *model checking* and *static analysis* of programs.

Model checking attempts to verify that a finite-state representation of a program is a refinement of a specification [Andersen, 1992; Vardi, 2007], which can lead to state explosion problems for large and precise representations of programs. Although techniques have been employed to safely reduce the state space that has to be explored (e.g. using partial order reductions [Peled, 1996]), this remains a problem associated with techniques that attempt both soundness and completeness.

On the other hand static analysis approaches tend to use abstraction techniques [Cousot and Cousot, 1977] to relax either or both soundness, completeness, leading to an analysis that is generally less intensive. Relaxing soundness (but not completeness) allows one to possibly identify true violations, but cannot allow us to verify the program is fully compliant. Relaxing completeness (but not soundness) allows one to possibly verify the program is fully compliant but any identified violations are not assured to be part of the program's behaviour at runtime.

Then, while model checking verification can fail because of resource exhaustion, static analysis techniques can also fail because they lack enough precision for the problem at hand. In some scenarios the program code may not even be available, thus excluding the possibility of proving the code safe pre-deployment leaving only the option of post-deployment verification.

0.1.1.2 Post-deployment Verification

Runtime verification is the main approach to post-deployment verification.

Runtime verification (RV) involves monitors that observe a program's execution at runtime and attempt to give a verdict about its adherence to a specification. This however comes with the limitation that there are interesting properties that are not monitorable (e.g. liveness properties [Bauer et al., 2011; Falcone et al., 2012b; Francalanza et al., 2017a]). Pure RV only observes the current execution prefix at runtime, unlike model checking that can analyse all possible executions.

RV techniques can be classified into two kinds, *asynchronous* and *synchronous*. Asynchronous techniques maintain a queue of program events with the monitor processing these possibly out-of-sync with the program. On the other hand synchronous techniques block the program while the monitor processes a program event. Working synchronously can affect program performance by adding time and memory overheads, unlike the asynchronous case. However synchronous monitoring allows one to detect violations as they occur, which can be useful

for the purposes of enforcing a property [Azzopardi et al., 2018b; Falcone et al., 2012b]. Tweaking and optimising monitor strategy implementations has proven successful in reducing these overheads (e.g. [Purandare et al., 2012]), however overheads remain significant (as evidenced by a recent competition for RV tools [Reger et al., 2016]) and unavoidable to an extent in pure RV.

0.1.1.3 Combining Verification Techniques

Upon a verification method failure then another method can be attempted, e.g. if static analysis fails to prove a program satisfies a certain monitorable property then RV can be used. However, instead of simply failing, verification attempts can be modified to produce artifacts that can be used by subsequent techniques, avoiding duplicate work. We briefly consider some approaches of this kind.

For model checking we find Beyer [2016]'s *conditional model checking*. Upon failure to prove the whole program satisfies the property, Beyer [2016] proposes modifying a model checker to: (i) return a representation of the program state space that remains to be proven safe; and (ii) to require as input such a constraint on the state space. Beyer et al. [2018b] go one step further and slice the program to produce a reduced program that can be used with off-the-shelf model checkers.

On the other-hand we also find limited combinations of static analysis and runtime verification. Bodden et al. [2010] analyse an instrumented Java program and a safety property as a finite-state automaton to identify event instrumentation sequences that can be removed without having an effect on violation detection. Another approach by Dwyer and Purandare [2007] instead summarises instrumentation sequences that always have the same effect module the specification. Both these approaches succeed in exploiting static analysis to make the RV problem more efficient, while also having the potential to prove certain parts of the program safe. Both of these however have their limitations: they are only validated with finite-state automaton with events corresponding to method calls, while the utility of the approaches with events that carry program data is unclear.

Another approach in literature attempts to directly prove parts of the property. Chimento et al. [2015] consider properties as automata with variable state, and states tagged with Hoare triples (pre- and post-conditions for method calls). Automated theorem proving is used to prove as many of possible of these method contracts, while the remainder are checked for at runtime with monitoring. This approach is data-oriented, unlike that of Bodden et al. [2010] and Dwyer and Purandare [2007], but however it only attempts to prove the Hoare triples. The automata used here in fact use transitions that can be guarded by the local and the program variable state, which are ignored by the analysis.

0.2 Thesis Contributions

The general context we are interested in this thesis is then *partial verification techniques* that extend verification techniques to return a verification *residue* or *residual* as proposed by Dwyer and Purandare [2008].

We identify several open questions, that we seek to answer:

- How do partial verification techniques report their progress in literature?
- What is a general theoretical basis for the notion of a residual in the context of partial verification?
- In the context of static analysis and runtime verification, what are appropriate residuals of automata-based specification that have both event- and state-based aspects?

To answer the first question we surveyed literature and identified a trend in the verification community towards combining verification techniques in a black-box manner, where a verification technique is not simply expected to fail when it cannot prove a property, but it is expected to report any progress it made. This allows one to transform the input problem into an easier one. We survey literature on such *partial verification techniques*, focusing on the kind of residue these techniques output. We classified these approaches based on different dimensions, including whether they reduce the program or property or whether they focus on event- or state-based verification techniques.

In response to the second question we give a formal basis for the combination of different verification techniques by defining precisely the notion of a *residual property*. We consider a meta-theory of programs and properties, where we remain agnostic of any program or property formalisation. Instead we simply assume a satisfaction relation between programs and properties, and use this to interpret a property as the set of programs that satisfy it. This allows us to define a partial order of residuals that captures the notion of when a residual property is equivalent to the original property modulo some knowledge of the program. A simple example is that when we want to prove that program P satisfies property $A \wedge B$, and we only manage to show that P satisfies A then B is an appropriate residual property. We instantiate this for both state-based and event-based verification.

We validate this approach by illustrating our contributions to an industrial project where the use of residuals added value to the business process in the form of static guarantees. This case study involved a payments ecosystem, where in the desire to remain technology-agnostic payment applications can be developed

in any language and run from external servers. Analysing the application code is a non-starter here then, since we cannot trust that the code on untrusted servers will not be modified maliciously at runtime. Instead a model-based approach is taken, where the developer provides a model of the assured behaviour of the application at runtime, that will also be enforced by the ecosystem. Our contribution to this picture involved the verification of regulations represented in a *controlled natural language* against these models, leaving any residuals to be enforced at runtime. This approach allowed us to give static guarantees of the application pre-deployment, which was integral for the ecosystem to make a determination on whether an application would be appropriate for deployment.

To answer the third question we fix our properties as *Dynamic Event Automata* (DEAs), and our program representation as the respective *control-flow automaton* (CFA). DEAs are extensions of finite-state automata with dynamic variable state, and in fact are a variant of the automata used by Chimento et al. [2015]. DEAs here are representations of monitors that wait for an event to occur and then act, while CFAs represent instrumented programs that trigger events. We apply static analysis in this context in an attempt to prune the required CFAs events, and the required DEA transitions. The analysis we present can be applied both at the *interprocedural* (as done by Dwyer and Purandare [2008]) and the *intraprocedural* level (as done by Bodden et al. [2010]).

In our presentation we focus on the latter approach, which allows us to perform incremental verification by verifying each program method separately (and possibly in parallel). By collecting the results for each method we can produce a result for the whole-program. Our approach involves modeling concretely all the possible ways a method may exercise the property, soundly abstracting all the program behaviour occurring outside of that method. This will allow us to prune DEAs and CFAs, removing any transitions and instrumentation that are determined to be irrelevant to violation-detection. This analysis for DEAs is implemented for both Java and the Solidity smart contract language. We evaluate the approach using several case studies in both these languages.

0.3 Outline

In our presentation of this work we divide this document in two parts. In Part I we survey partial verification techniques in literature, and formalise an approach to this in terms of property residuals, while we present an industrial case study utilising the proposed approach. In Part II we present an implementation of a residual for properties as dynamic event automata and programs as control-flow automata, specifying static analyses to reduce both event instrumentation and

produce property residuals, leaving a lesser burden on the program at runtime in terms of overheads.

Part I

Combining Verification Methods

1

Introduction

A significant bulk of research in formal verification has been dedicated to techniques that increase the reach and automatibility of model checking and theorem proving. These powerful techniques are used to attempt to verify exactly the properties required of a program. However this problem can be hard, and in fact many such techniques can simply fail after having exhausted the memory available, or fail to give a result within a reasonable time frame. Meanwhile, there are further theoretical and business limits on what can be proven by one method.

Model checking of infinite-state systems is undecidable [Uribe, 2000]. While monitoring is limited to monitorable properties — a property of the kind *eventually the file must be closed* cannot be proven satisfied by monitoring the prefixes of an infinite execution, since after observing an execution prefix the monitor cannot determine anything about its continuation [Bauer et al., 2011; Falcone et al., 2012b; Francalanza et al., 2017a]). On the other hand business requirements can further limit the kind of analysis possible. Some requirements may be subject to change at runtime (e.g. the limit on a transaction, in an online financial system), requiring dynamic techniques that can be adapted in real-time to the new specification. Other requirements may necessarily be required by the client to be guaranteed pre-deployment, regardless of the difficulty of verification.

Theoretical limitations of a single method have been handled by combining different verification or analysis techniques in one hybrid algorithm. For example, Leucker [2013] shows how model checking can be applied at runtime to explore the possible continuations of a prefix, leading to the possibility of determining satisfaction of liveness properties for the current execution. Bonakdarpour et al. [2018] and Stucki et al. [2019] also propose to use such a method to be able monitor for properties about sets of executions, while Russo and Sabelfeld [2010]

parametrise monitoring by a control-flow graph, allowing a monitor to detect when knowledge about high-level variables can escape to low-level locations. Aceto et al. [2018a] propose a theoretical framework for this kind of parametrised monitoring. Dually, runtime information can be used pre-deployment, e.g. Grech et al. [2017] and Grech et al. [2018] collect information about the heap at runtime to reduce the unsoundness of certain static analyses. These kinds of approaches use analysis techniques to provide more information to a verification step.

Other approaches combine techniques that are both engaged in the activity of verification, as opposed to analysis. These techniques are our focus here. This kind of approach requires self-contained verification techniques that attempt to prove a program satisfies a specification, and report their progress, pushing proof obligations onto other verification steps. This ensures that any verification step has the potential to contribute to the verification effort, even upon failure, and that the computational effort does not go to waste. Here we survey such techniques in literature, identifying work that fits in this view from typing, model checking, static analysis, and runtime verification. We classify these mainly based on the artifact or residue used to report the work done. At a high-level we classify these as transformations of the program state space and/or transformation of the property.

To analyse the notion of when a reduced property is a valid residual, in that it is equivalent to the original property with respect to the program, we consider an abstract formal framework for residual analysis, where given an abstraction of a program we use the notion of *quotienting* to reduce the property by what is known about the program. The use of quotienting in the context of program verification is not new, in fact Andersen [1995] use quotienting to perform incremental verification of a program, by reducing a property incrementally by each component of the system, reducing the problem to showing that residual property holds of the remaining components. However, here we are applying quotienting in the context of abstractions of a whole program, that are not necessarily complete, and such that the verification problem cannot necessarily be reduced to a structural sub-part of the program. We instantiate this theory for both predicate-based (or state-based) and event-based theories for verification.

As a case study, we consider an application of residuals in the context of an industrial project with untrusted code that however comes with a model of promised runtime behaviour. This model is enforced at runtime, ensuring it is an abstraction of the program's runtime behaviour. We show how other required properties of the system can be checked on this model pre-deployment, possibly leaving some residual properties to be proven at runtime.

In Chapter 2 we explore how specific instances of these techniques have been exploited to allow *partial verification*. In this work we can tie the output of a partial

verification technique with the notion of a *residual*, capturing the remaining proof obligations. In Chapter 3 we present a foundation for composing verification methods using property residuals giving instantiations of this framework for both state- and event-based properties, while in Chapter 4 we present an industrial case study utilising the notion of residuals for partial verification.

2

Partial Verification in Literature

Verification techniques that simply return a verdict are not very useful when they fail, as they must do in some cases since the problem of verification is in general undecidable. Failing can also occur for other reasons, among these because the method employed has some constraints on precision, or a user set timeout was reached before a verdict could be given. Then, not reaching a verdict does not mean the technique did not manage to make some progress towards proving a property of a program, but only that it did not manage to prove it for the whole program. Without a verdict then, in the traditional verification context, one is forced to use other verification methods that hopefully do not suffer from the same fate. However one can do better.

Techniques exist that allow for *partial or gradual verification*, where different methods chip away at a verification problem, exploiting each other's strengths in solving a problem that each individual method could not solve. This requires an extension of the notion of a verifier, where instead of a fail verdict the verifier returns some representation of the progress that it managed to make. This artifact would then be usable by other verifiers that can continue refining the problem in this way, until hopefully the problem is proven or disproven. In effect such a verifier is transforming the verification problem, reducing it to an easier one (e.g. one that requires less state space to explore). We call such verifiers *partial verifiers*, since they at the very least attempt to solve part of the problem. To our knowledge there is no published review of such methods.

In this chapter we review partial verifiers that appear in literature. We seek to answer the following questions:

- What are existing approaches that allow for partial reasoning in verification?

- How do partial verifiers communicate their progress in practice?

We start with some preliminaries in Section 2.1, including a description of the criteria we use for the classification of partial verifiers, while in Section 2.2 we consider the approaches found in literature. We consider related work in Section 2.3, and conclude in Section 2.4.

2.1 Preliminaries

The focus here is on partial verification methods and their combination with other (possibly non-partial) methods. We maintain a distance between the notion of *analysis* and that of *verification*, where analysis is the computation of properties of a program and verification is the problem of deciding whether a program satisfies a certain property. Although partial verifiers may make use of analysis techniques (e.g. program abstraction techniques) to attempt to reach a verdict, the focus here is on the partial verifier as a black box that takes a program and a property and returns either a verdict or a residual verification problem. The focus then was on identifying verifiers of this form, and their use with other verifiers, ignoring tighter combinations of analysis methods.

2.1.1 Classification

To compare and contrast approaches at a high-level we classify them on several dimensions:

(i) the kind of verification methods they employ;

(ii) the input program and property languages;

(iii) the way they transform the verification problem; and

(iv) the kind verification methods with which they are paired.

Figure 2.1 illustrates the taxonomy we specified to classify the approaches in this way, of which we now give an overview.

We do not identify verification methods by the specific algorithm used, but keep to the general kind of an algorithm, including: model checking (MC), static analysis (SA), runtime verification (RV), testing, and manual code review. This allows us one to compare the approaches at a high-level based on the limitations associated with each approach.

We take note of the programming language (concrete or abstract) that each approach can process. In some reviewed papers the work may be presented in

terms of some abstract representation of a program, but evaluated in terms of a certain concrete programming language. In these cases here we list this concrete language, and list the abstract language only when there is no implementation in terms of a concrete language.

The specific property languages used vary widely, with little intersection between the approaches. Then, instead of noting the property language used by an approach, we consider high-level aspects of the language, on two dimensions. We consider the kinds of properties specifiable by the language, including assertions (predicates that hold at a certain point in a program), Hoare triples (pre- and post-conditions around a part of the system), automaton-based properties, reachability properties (i.e. whether a certain state in the program is reachable or not), and types. We also include as a kind of properties *specialised properties*, since some approaches may have some pre-defined properties they wish to verify, instead of having a more general property language. We also classify the property language by the program context the property is talking about, which includes statement-level (e.g. *this point in the program should be reachable*, or *this assertion should hold true at this point in the program*), module-level (e.g. pre- and post-conditions over a function), program-level (e.g. *the program should never exhibit this behaviour*), and typestate (e.g. *objects of a certain type must exhibit this behaviour*).

The main point of comparison between the different methods is the artifacts they produce. We divide these into two major interrelated kinds: (i) property transformations; and (ii) program state space transformations.

By property transformations we mean that the original property is in some way modified. This can include a partial evaluation of the property where proven parts of the property are pruned away, or the creation of sufficient properties, termed assumptions, whose satisfaction implies that of the original property. These two notions are similar, however the latter is a powerful and general approach. We distinguish between them to distinguish between approaches that simply structurally prune a property and others that create generate new proof obligations that are at least semantically sufficient.

Another approach is to consider the program state space and make progress on proving it safe by incrementally covering more of the space. The approaches we reviewed approach this by either outputting a representation of the states proven compliant by a method or dually those states that can still potentially fail. These are dual approaches, since the compliant set of states can be identified by the set difference of the set of all states and the potentially failing ones. Other approaches go further than this by projecting the verification problem onto smaller parts of the program. These parts can either be structural components or more a general a slice of the program that represents the yet unproven part of

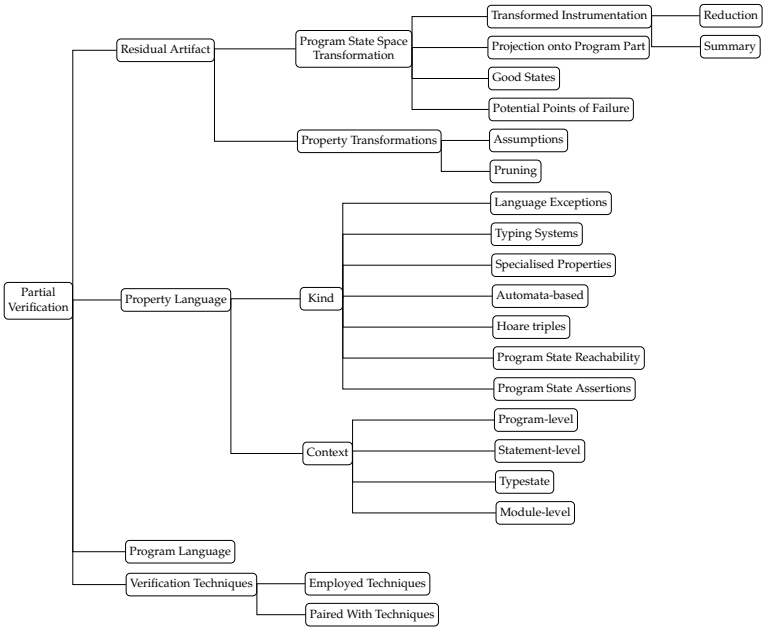


Figure 2.1: A taxonomy for combinations of verification methods in literature.

the program state space. These kinds of approaches are particularly re-usable, since the *residual program* can be used with other techniques off-the-self. Other approaches consider properties about events that a program may trigger. In this case, the state space to be explored is represented by the possible traces of events triggered at runtime. We identified some approaches that attempt to prove such event-instrumented programs safe, while reporting any progress by removing or summarising sequences of event instrumentation.

The described taxonomy is illustrated in Figure 2.1. In the next section we consider and classify pertinent literature along these lines.

2.2 Description of Approaches

In this section we describe and discuss published literature about partial verifiers along the lines of the taxonomy discussed in the previous section. The approaches discussed are classified according to this taxonomy in Table 2.2 and Table 2.3. Another distinction is at the level of implementation, which we do not consider here. For example, some methods are more generally applicable, like those by Beyer et al. [2018b]; Chebaro et al. [2012]; Czech et al. [2015] and Chimento et al. [2015], in that they produce immediately a reduced program or property that can be analysed by subsequent unrelated checkers. While others act as frameworks

Table 2.2: Artifact class of each partial verification approach included in the review.

Approach	Residual Artifact						
	Property Transformations		Program State Space Transformations				
	Assumptions	Pruning	Good States	Potential Points of Failure	Projection onto Program Part	Transforming Instrumentation	
						Reduction	Summary
[Correnson and Signoles, 2012]	✓	-	-	-	-	-	-
[Christakis et al., 2012]	✓	-	-	-	-	-	-
[Kanig et al., 2014]	✓	-	-	-	-	-	-
[Christakis and Wüstholz, 2016]	✓	-	-	-	-	-	-
CLARA [Bodden et al., 2010]	-	✓	-	-	-	✓	-
STARVOORS [Chimento et al., 2015]	✓	✓	-	-	-	-	-
[Beyer et al., 2012], [Beyer et al., 2018b]	-	-	✓	-	✓	-	-
[Fink et al., 2008]	-	-	-	✓	-	-	-
[Chebaro et al., 2012]	-	-	-	-	✓	-	-
[Czech et al., 2015]	-	-	✓	-	✓	-	-
[Lal et al., 2007]	-	-	-	✓	✓	-	-
[Choi et al., 2002]	-	-	-	-	-	✓	-
[Dwyer and Purandare, 2007]	-	-	-	-	-	-	✓
[Cartwright and Fagan, 1991]	-	✓	-	-	-	-	-
[Knowles and Flanagan, 2010]	-	✓	-	-	-	-	-
[Siek, 2006]	-	✓	-	-	-	-	-
[Thatte, 1990]	-	✓	-	-	-	-	-
[Andersen, 1995]	✓	-	-	-	✓	-	-

for algorithms using some internal representation of the transformed property or state space, for example work by Bodden et al. [2010]; Fink et al. [2008] and Correnson and Signoles [2012].

2.2.1 Property Transformation

A class of approaches we identified act solely on the property, in that they transform the property that is to be proven by either generating sufficient conditions for it, or by reducing a property by pruning away parts of it. Subsequent techniques can then focus on the reduced or transformed properties. We describe several examples of these approaches below.

2.2.1.1 Moving Goalposts with Assumptions

A chosen analysis technique may not always be powerful enough to prove a property, succeeding for some cases but failing for others. Several techniques frame these failed cases in terms of assumptions or proof obligations that must be fulfilled before full compliance is assured. These assumptions then act as the new transformed property whose verification is attempted by subsequent steps. Here we consider several applications of this approach to the combination of verification attempts, describing in the process the environment within which they are used for a better view of their use.

Table 2.3: Verification aspects of the partial verification approaches reviewed.

Approach	Program State Space Language	Property Language		Verification Techniques	
		Kind	Context	Employed	Paired With
[Correnson and Signoles, 2012]	C	Exceptions, Assertions, Hoare Triples	Statement-level	SA, Theorem Proving	Testing
[Christakis et al., 2012]	.NET	Assertions	Statement-level	SA	SA, Testing
[Kanig et al., 2014]	SPARK	Specialised, Hoare Triples	Module-level	SA, Unit Testing, Code Review	SA, Unit Testing, Code Review
[Christakis and Wüstholz, 2016]	.NET	Assertions	Statement-level	SA	SA, T
CLARA [Bodden et al., 2010]	Java	Automata-based	Program-level, Typestate	SA	SA, RV
STARVOORS [Chimento et al., 2015]	Java	Automata-based	Program-level, Typestate	SA	RV
[Beyer et al., 2012], [Beyer et al., 2018b]	C	Automata-based, Reachability	Program-level	MC, SA	MC, SA
[Fink et al., 2008]	Java	Automata-based	Program-level, Typestate	SA	SA
[Chebaro et al., 2012]	C	Specialised (Division by 0, Out of bounds array access)	Statement-level	SA	DA
[Czech et al., 2015]	C	Automata-based, Reachability	Program-level	MC	Concolic Testing
[Lal et al., 2007]	Weighted Pushdown Systems	Reachability	Program-level	SA, MC	SA, MC
[Choi et al., 2002]	Java	Specialised (Data Races)	Program-level, Typestate	SA	RV
[Dwyer and Purandare, 2007]	Java	Automata-based	Program-level, Typestate	SA	RV
[Cartwright and Fagan, 1991]	*A Functional Language*	Typing System	Value-level	SA	RV
[Knowles and Flanagan, 2010]	*A Lambda Calculus*	Typing System	Value-level	SA	RV
[Siek, 2006]	*A Lambda Calculus*	Typing System	Value-level	SA	RV
[Thatte, 1990]	Lambda Calculus	Typing System	Value-level	SA	RV
[Andersen, 1995]	μ-calculus	Logic-based	Program-level	MC	MC

Kirchner et al. [2015] present Frama C, a tool that combines different analysis techniques all aimed at the verification of C programs. Properties here include both out-of-the-box common C runtime errors, and custom specifications. the latter are specified using a first-order logical language [Baudin et al., 2011] for Hoare-style functional contracts that is also capable of encoding LTL and finite-state automata [Kirchner et al., 2015]. The specifications are inlined as code assertions, thus properties here are considered to be pairs of predicates and program points. Correnson and Signoles [2012] describe how Frama C combines the results of multiple static analysis techniques in a correct manner allowing for partial results from one technique to be made complete by another. This is done by allowing the different techniques to produce results based on certain assumptions (e.g. when one technique cannot prove that a property holds of a certain pointer, it assumes it holds so that it can continue, outputting this outstanding proof obligation to be handled by another technique). These assumptions can also

be discharged at runtime using runtime assertion checking. The techniques combined here are an abstract interpretation based data-flow analysis of the possible value of memory locations, and a weakest precondition analysis.

A similar approach proposed by Christakis et al. [2012] takes a C# program instrumented with assertions, and a static checker which marks an assertion as satisfied, partially satisfied under some assumption, or unknown. Other static checkers are then proposed try to refine these results through further annotation. Testing is then directed towards assumptions and unverified assertions by generating appropriate inputs through symbolic execution. This is implemented in CodeContracts [Fähndrich et al., 2010]. Assumptions here are also at points in a function, rather than whole-program conditions.

Kanig et al. [2014] propose that instead of a verification method outputting simply the verdict, that the results also include the *assumptions* that this verdict depends on, and generalise a verdict to a general notion of a *claim* expressed as a Horn clause. The assumptions here include both any assumptions that are taken by the tool to simplify the verification context, and also assumptions manually included by the developer about components of the system (a difference from the other approaches we found). The authors use this approach in an industrial project using SPARK (a subset of Ada). Implementation-wise, they use parametrised labels to identify properties about the program, while allowing for assumptions to be discharged either by static analysis, unit testing, or simple code review.

Lack of adequate precision is not the only reason an analysis may not fully verify a property, but an analysis may also fail because it exhausts all the available resources. One approach that deals with this gracefully takes Clousot [Fähndrich and Logozzo, 2011], an abstract interpretation engine for CodeContracts, and allows its analysis to be bound (e.g. by time, or memory usage). On the resources being exhausted partial results are recorded through assertions in the program [Christakis and Wüstholz, 2016]. This approach then allows for an unsound limitation of an abstract interpreter (since bounding resource usage results in an analysis that does not necessarily finish) to be made sound by subsequent verification steps.

2.2.1.1.1 Gradual Quotienting

Partial model checking was introduced by Andersen [1995] in the context of a variant of the μ-calculus and finite-state model checking. The problem context is that of multiple components whose parallel composition must satisfy some property. Parallel composition easily leads to state-explosions, given the multitude of possible interleavings. Instead of producing the composition explicitly,

Andersen [1995] proposes a *quotient operator* to reduce a property by a component, producing a residual specification of the rest of the program. Doing this in turn for every component but one, transforms the problem into the simpler checking scenario of needing to check that one sequential component satisfies a property.

2.2.1.2 Pruning Parts of a Property

The techniques considered so far generate assumptions that are sufficient for compliance with the original property assertions. Instead of proceeding in this manner, another more limited approach is to prove parts of the specification, leaving the parts left to prove. This is related to the previous approaches, in that the reduced property is the assumption that must be proven by subsequent attempts. However the artifact here is limited to structural reductions of the previous property with respect to the program, while generated assumptions may be specialised to different program locations and different cases.

A simple approach by Bodden et al. [2010], included in the CLARA tool (described in more detail in the program transformation section), is that of identifying transitions in a finite-state automaton that are tagged by events that never occur in a given program, leaving a reduced automaton. A more sophisticated but technique is that of Chimento et al. [2015], where they present an approach for Java that combines control- and data-oriented analysis in the STARVOORS tool. This approach combines deductive verification with runtime verification, for properties as symbolic automata with states possibly labelled with Hoare triples (pre- and post-conditions for function calls). This approach uses the `KeY` theorem prover [Ahrendt et al., 2016] for Java to prove some of these triples safe. The residual property here then is equivalent to the original property without the statically proven Hoare triples, and is left for runtime to be monitored for by the LARVA runtime verification tool [Colombo et al., 2009]. Reducing a property in this way was shown to significantly reduce overheads in the presented use cases. STARVOORS is also able to specialise Hoare triples to the program, adding conditions that prevent runtime checks for already proven cases.

We find similar approaches for static and dynamic typing, where properties consist of types for certain values or variables at certain points in a program. Static type systems accept programs that cannot have an execution that respects typing, providing static guarantees at load-time. On the other hand dynamic type systems allow all correctly typed executions of a program. Dynamic type systems thus allow more behaviour, at the expense of pushing analysis to runtime, which is not ideal. However dynamicity is essential in some cases, e.g. when a program interacts with another statically unknown program. Meijer and Drayton [2004] argue for the use of both of these in conjunction, using static typing wherever

possible and where it is impossible using dynamic typing. In fact a number of approaches have been developed to this end.

Cartwright and Fagan [1991] introduce a *soft typing* system to use static typing to prove most of the program safe, and instrumenting the parts of the program that cannot be proven safe or violating statically (since static type checkers are usually just sound not complete) with dynamic checks (and possibly exception handlers). Knowles and Flanagan [2010] introduce the notion of *hybrid type checking*, which allows one to define a type as a set of values that obey a certain predicate. Due to the expressiveness of these types a static type checker may fail in proving type consistency in certain parts of a program. Possibly unsafe operations can then be instrumented with type casts, to perform type checking at runtime. Another notable approach is that of *gradual typing* as introduced by Siek [2006], which allows functions to be given an implicit dynamic or unknown type. Then static typing confirms consistency of the statically typed part of the program, leaving the untyped (or dynamically typed) constructs to be verified at runtime. The program then is instrumented with appropriate type casts for terms with a dynamic type. This approach differs from hybrid type checking in that the latter is fully typed with concrete types, while in gradual typing we have a dynamic type. A similar approach is *quasi-static type checking* [Thatte, 1990], however it compares negatively in the power of static type checking.

2.2.2 Program State Space Transformation

A dual approach to transforming properties is to transform the program, by producing either: (i) artifacts that identify the parts of the program that have been determined to satisfy the property; or (ii) transformed programs that exercise a property in an equivalent manner to the original. These are related, in that the first kind of artifacts can be used to generate a smaller program exhibiting only the possibly violating behaviour. However there are approaches that simply use these artifacts on-the-fly without producing a concrete transformed program. Both approaches have the effect of reducing the state space that is considered by subsequent techniques. We describe examples of these below.

2.2.2.1 Identifying Satisfying States

By identifying program states that respect a property, subsequent attempts can focus on those programs states that have not been determined to respect the property. A dual way to think of this is to identify the states that can potentially violate the property and have analyses to prune this set. Here we describe instances of this approach.

2.2.2.1.1 Conditional Model Checking

Beyer et al. [2012] identify a problem with traditional model checking, namely that they either return a satisfaction verdict, a violation verdict, or they fail (i.e. an unknown verdict). However, model checking is time- and memory-intensive and an unknown verdict results in a waste of these resources. Thus, Beyer et al. [2012] illustrate how instead model checking techniques can be developed that instead of an unknown verdict they return a condition ϕ that represents the states of the model that satisfy the property. Then other model checkers can be restricted to reduce the search for compliance with a property to states that do not respect the condition ϕ. This approach to model checking is termed *conditional model checking*.

2.2.2.1.2 Potential Points of Failure

A similar approach is employed for the static verification of typestate properties (i.e. properties that objects of a certain type should always satisfy) by [Fink et al., 2008]. Here multiple static analysis techniques are used successively, with each technique using a more precise abstract semantics of the programming language. For example, one analysis relates object pointers together with a *may* relation, representing that the pointers may point to the same object; while a more strong subsequent analysis considers also that some pointers may be related with a *must* relation, representing that the pointers necessarily point to the same object. These different analyses each take as input a *verification context*, representing a state in the system and the object that may violate at the state. These are also known as *potential points of failure* (PPFs) Each step refines this context, such that if it is empty then the property has been verified.

2.2.2.2 Program Transformation

Outputting an artifact that represents the satisfied (or potentially violating) states of a program needs subsequent analysis tools that are tailored to take such an artifact as an input. However the majority of available established tools do not allow for such a parameter. To be able to re-use other techniques, while simplifying the problem, slicing and chopping techniques (see [Tip, 1995] for a survey of these) can be used to extract a pruned version of the program that exhibits the possibly violating behaviour.

Other approaches start with an instrumented version of the program, e.g. for the purposes of runtime verification, and attempt to identify instrumentation points that can either be removed soundly and completely, or identify a sequence of these points that can be summarised. Removing all such points means a

program satisfies the property, while summaries helps subsequent techniques avoid re-performing the same work.

2.2.2.2.1 Reducer-Based Conditional Model Checking

Beyer et al. [2018b] extend conditional model checking by showing how even model checkers not equipped to handle conditional model checking can be used off-the-shelf. This is done through a pre-processing step that slices the program, resulting in a *residual program* that can be inputted to any traditional model checker. Programs are represented as a variant of control-flow graphs, specifically as automata with program operations on transitions. Conditions are also represented as automata that represent covers of certain program paths. An algorithm is then defined to reduce the program automaton by the already covered paths, and is implemented using parallel composition and pruning of the edges of this composition.

Other earlier approaches by Chebaro et al. [2012] and Czech et al. [2015] apply slicing in a similar manner to extract slices of the program that can possibly be violated and exercise them using testing to attempt to find counter-examples to the property in these slices. This reduces the state-space that testing has available, increasing the probability that a counter-example is found (if present).

2.2.2.2.2 Abstract Error Projection

An interesting related notion that allows different model checking techniques to be combined is that of an *abstract error projection* [Lal et al., 2007]. Essentially, given a property, and a program abstraction then model checking is employed to partition the abstraction into the set of paths in the abstraction that are compliant with the property, and the set of paths that are not. The part of the abstraction for which there may be an error is then characterized by the set of states in the abstraction that are part of a possibly violating path, i.e. the error projection. Then, further verification attempts can be focused on this part instead of the whole program. Lal et al. [2007] use abstraction-refinement techniques to make the abstraction more precise, and optimise the analysis iteratively by focusing these refinement techniques on the error projection of the previous abstraction.

2.2.2.2.3 Transforming Instrumentation

Some approaches to verification are event-based, where they used automata tagged by program events to represent the desired behaviour, and instrumented the program with these events. The problem is then to show that the event traces triggered by the instrumented program during each run are traces marked as

compliant by the property automaton. This is a common way to implement runtime verification, for example. Here we discuss approaches in literature that prune this instrumentation in the process of proving that certain program behaviour does not affect compliance of the property.

Bodden et. al. present a framework, CLARA [Bodden et al., 2010], that analyses Java programs at different levels of precision to determine which parts of a program cannot violate a finite-state machine equivalent specification. By removing instrumentation points or sequences of such points that do not affect the property monitor state, the points left for subsequent analyses (including monitoring) is reduced.

A similar approach for data race detection is presented by Choi et al. [2002], where the instrumentation required for data race detection is reduced by using points-to analysis and escape analysis to identify soundly: (i) when two objects may access the same memory location; (ii) when these two objects may be in different threads; and (ii) when they do not have a common lock. Instrumentation events for objects are then discarded when it is determined statically that they cannot participate in data races.

A complementary approach instead of simply removing instrumentation is to summarise parts of the program that always behave in the same manner. [Dwyer and Purandare, 2007], given a property as a finite-state automaton with method calls as events, use static analysis over the control-flow graph of a program to attempt to prove compliance with the property. In the case that not all of the program is proven compliant, safe regions of the program are identified. Instrumentation is then changed to summarise the already known effects of these regions, allowing subsequent analysis to step over these already verified regions. This involves simulating runtime monitoring statically by running the control-flow graph and property in parallel, and identifying sequences of statements that are both never in a bad state of the property and whose single entry and exit points are always in the same respective property states (q_{entry}, q_{exit}). In preparation for runtime monitoring, the instrumentation within this region is removed, and its effect is simply summarised by instrumenting it to trigger a new unique event (e). The property is then augmented with a transition from q_{entry} to q_{exit} with the e. This kind of analysis is termed a *residual analysis*, and the new property produced the *residual* property.

2.3 Related Work

We have focused on loosely coupled verification methods that produce partial results, leaving further proof obligations for other verification methods. Here we

briefly consider tangentially related methods that do not fit into our definition of partial verification but are related in motivation or techniques.

Tighter combinations of analysis and verification methods are also useful. A notable example of an approach that allows tight combinations of multiple techniques is that of Beyer et al. [2007] in CPACHECKER. In this approach to *configurable program analysis* an analysis over the program is a configurable tuple of: (i) an abstract domain (the concrete states, a semi-lattice of abstract states, and a concretization function); (ii) a transfer relation (how abstract states evolve given a program step); (iii) a merge operator (how two abstract states are merged into one); and (iv) a termination check (when to stop the analysis). Multiple analyses can then be composed simply by having multiple abstract domains, and defining an appropriate transfer relation, merging operator, and termination over the extended abstract domain. This allows for the combination of analyses with different abstract domains, including also model checking, as illustrated in Beyer et al. [2018a]. Tighter combinations of formal methods can create an analysis that is more powerful than its constituent parts, however it can still benefit from partial reasoning, given exhaustion of resources or too much abstraction. In fact conditional model checking [Beyer et al., 2012] has been implemented around CPACHECKER.

Here we have also not considered approaches that reduce the verification problem for purposes of optimisation or simplification, since we are interested in approaches that deal gracefully with failure to prove a property. These kinds of approaches include: partial order reduction [Peled, 1996] that reduces model checking to checking only the representatives of an equivalence class, verification condition generators [Frade and Pinto, 2011] that use program analysis to produce a logical proposition that is true if and only if the program satisfies the property, thus combining both program analysis and theorem proving. A notable approach is that of decomposition or reduction of properties onto components [Lamport and Abadi, 1994], usually for *assume-guarantee reasoning* where an abstraction of a component (perhaps computed automatically, e.g. [Gheorghiu Bobaru et al., 2008]) is used instead of its implementation in the checking of a whole system.

In the context of time-triggered runtime verification Navabpour et al. [2013] present RiTHM, an approach for time-triggered monitoring that depends on a static analysis for correct results at runtime. In this context instrumentation adds code that keeps a history or record of the program state at some point. This is necessary since in the time-triggered context a monitor is not activated by program events but instead polls the program intermittently. Navabpour et al. [2013] uses static analysis to identify the minimum amount of instrumentation points required given a certain polling interval to ensure no relevant behaviour is missed at runtime. Navabpour et al. [2012]; Wu [2013] define further some

heuristics for optimising this kind of monitoring, essentially involving static analysis of which variables' values need to be instrumented and buffered for a near-optimal solution to the problem of reducing overheads with time-triggered monitoring.

In the context of runtime enforcement, Colcombet and Fradet [2000] exploit static analysis to reduce the program transformations required to produce a program that satisfies a separately defined property. This is similar to partial verification methods, however it differs in motivation – enforcement is the main aim here as opposed to verification.

Other approaches use static analysis to control overheads. Stoller et al. [2012] consider systems that may turn off monitoring for some events at runtime to avoid overheads, but which may affect the verdict given on the execution. Instead they use statistical methods to be able to give a probability of the verdict given being correct. Bartocci et al. [2013] optimise this approach with an algorithm that can calculate the possibly required runtime calculations in this context ahead-of-time.

2.4 Conclusions

In this chapter we have surveyed the literature and identified several approaches utilising some kind of partial verification techniques. These approaches either give a verdict for the verification problem, or encode their progress in verifying a problem by transforming the verification problem. We have classified them according to the artifacts they produce, both the programs and kinds of properties they accept, and the verification methods they both employ and are paired with.

The presented work represents a comprehensive overview of the current work along the lines of partial verification methods. However, although much of this work uses or mentions the notion of a *residual* (of a program or of a property), no clear general characterisation is given. In the next chapter we discuss this notion, focusing on the residual of a property.

3

A Foundation for Residual Analysis

In the previous chapters we have motivated the combination of verification methods to make up for the possible failure of a single verification method in computing a verdict. We have also reviewed existing methods that use a verification method to prove part of a property and leaving the remaining part for other methods. This work lacks a general framework of how and when this approach works. In this section we tackle this problem, and present a foundational formal theory for properties and verification which we use to prove the general correctness of such approaches. This work is based on work published in [Azzopardi et al., 2016a].

Summary A wide variety of property specification languages exist, corresponding to some form of logic or automata. Programs similarly can be specified using a multitude of languages, where they are to be interpreted through some semantics. By defining a satisfaction condition we can attempt to check whether a program satisfies a property.

A common approach is to specify properties in some form of temporal logic, e.g. LTL which can be given semantics in terms of traces or automata [Bauer et al., 2010], or to specify properties directly as some form of automata. Programs can similarly be intepreted as a transition system using some operational semantics. A common satisfaction condition is then simply that the program is a refinement (or a model) of the property [Vardi, 2007]. Verification methods (such as model checking, deductive verification, and runtime verification) are an attempt to show such a satisfaction condition.

In static analysis this is done by abstracting a concrete program, for instance by using an abstract operational semantics [Cousot and Cousot, 1977] we can

construct a labelled transition system (explicit or implicit) representation (e.g. a control-flow graph) of the program and compare this against the property. Model checking on the other hand tends to assume a more exact program representation [Visser et al., 2003], and explores its state space for possible violations of the property. In runtime verification (RV) we do not reason about the semantics of the whole program, but only of the current execution trace. These all have different power, and attempt to approximate the satisfaction condition at different levels — RV cannot be used to prove liveness properties in general, while static analysis is naturally limited by the abstraction it uses and more precise model checking by the resources (memory and time) available.

To deal with the possible failure to return a satisfaction verdict of any verification method in this chapter we layout a foundation for the combination of different verification methods using the notion of property *quotients*. In this chapter we remain language-agnostic in that we do not assume any specification language for properties or programs, nor do we assume any concrete semantics for these. We choose to do this to show how the notion of quotients arises naturally and easily in the context of verification.

We simply assume a class of programs $\mathcal{P}$, a class of properties Π, and a satisfaction operator $\vdash$: $\mathcal{P} \times \Pi$ where $P \vdash \pi$ means that program P satisfies property π. This relation does not represent the application of a verification technique, but the general satisfaction condition that verification techniques attempt to verify.

We then interpret properties as *program acceptors*, using the satisfaction operator, i.e. $[\pi] \subseteq \mathcal{P}$ such that $P \in [\pi]$ iff π accepts P iff $P \vdash \pi$. Over this we can define a lattice corresponding to $\mathcal{P}$'s subset lattice by defining conjunction, disjunction, and negation operators for properties corresponding to intersection, union, and complement in the program space. Properties are related in this lattice by a refinement operator $\sqsubseteq$ that corresponds to the subset operator in the program space. In the context of this simple theory we will consider that verification methods can fail to return a result but that the analysis used for verification may still prove something of the program. We can then characterize the notion of a property quotient as a representation of what remains to be proven.

In Section 3.1 we describe the background theory, while in Section 3.2 we present the class of quotient properties, with respect to a couple of properties. We present instantiations of this for state- and event-based theories in Section 3.3, while we conclude in Section 3.6.

3.1 A Formal Theory of Verification

Formal verification is the attempt at showing that a certain program satisfies a certain property. Within this definition we have three unclear notions: (i) what are programs? (ii) what are properties? and (iii) what does it mean for a program to satisfy a property? Here, instead of defining these using a certain formalisation, we simply assume we have some interpretation of these and use them as the basic building blocks of our framework.

Definition 3.1.1. *A verification context is a tuple* $\langle \mathcal{P}, \Pi, \vdash \rangle$*, where* $\mathcal{P}$ *is a non-empty set of programs,* Π *is a non-empty set of properties, and* $\vdash$*:* $\mathcal{P} \times \Pi$ *is a relation that relates a program with a property if the program satisfies that property. We write* $P \vdash \pi$ *for* $(P, \pi) \in \vdash$*, and for* $(P, \pi) \notin \vdash$ *we write* $P \nvdash \pi$*.*

Algorithms do not exist that can compute $\vdash$ fully for Turing-complete programs and for all properties (e.g. consider that the Halting problem is undecidable), however methods exist that can under-approximate it. We thus define verification methods as any relation that can do this.

Definition 3.1.2. *A* verification method $\vdash_V \subseteq \mathcal{P} \times \Pi$*, is an algorithm that under-approximates the verification problem, i.e.* $\vdash_V \subseteq \vdash$*.*

Note that here $P \not\vdash_V \pi$ does not mean that $P \nvdash \pi$, but simply that we have not been able to prove satisfaction (if we have property negation then $P \vdash_V \neg\pi$ means that $P \nvdash \pi$).

3.1.1 Properties

We consider a general semantics for properties in terms of the verification relation. In verification properties are used to be checked against a program. The actual formal semantics of properties used (e.g. trace semantics) is intended to capture certain desirable qualities of a program, and then they can be given a semantics in terms of the programs they accept.

Definition 3.1.3 (Property Verification Semantics)**.** *A property* π *has an associated set of programs that it accepts, denoted by* $[\pi] \stackrel{\text{def}}{=} \{P \mid P \vdash \pi\}$*. We call this the* verification semantics *of properties.*

Generally, specification languages contain some form of conjunction (e.g. logical conjunction for logic-based approaches or language intersection for automata), disjunction (e.g. logical disjunction or language union), and negation (e.g. logical negation or language complement). We can characterise these operators in terms of the given semantics.

Definition 3.1.4 (Property Operators)**.** *The semantics of the basic property operators over* Π *are defined as follows:*

- *The conjunction of two properties,* $\barwedge$*, creates a property that accepts only the programs accepted by the two properties conjuncts:* $[\pi \barwedge \pi'] \stackrel{\text{def}}{=} [\pi] \cap [\pi']$.
- *The disjunction of two properties,* $\barvee$*, creates a property that accepts all the programs accepted by the two properties disjuncts:* $[\pi \barvee \pi'] \stackrel{\text{def}}{=} [\pi] \cup [\pi']$.
- *The negation of a property,* $\neg$*, accepts any program not accepted by the negated property:* $[\neg\pi] \stackrel{\text{def}}{=} \mathcal{P} \setminus [\pi]$.

We can then characterise refinement between properties in this semantics, based on whether satisfaction of a property implies that of another, or whether the programs accepted by a property are a subset of the other.

Definition 3.1.5 (Property Refinement)**.** *A property* π' *is said to be a refinement of a property* π *if every program* π *accepts is also accepted by* π'*:* $\pi \sqsubseteq \pi' \stackrel{\text{def}}{=} [\pi] \subseteq [\pi']$. *We use* $\pi \equiv \pi'$ *to denote* $\pi \sqsubseteq \pi' \barwedge \pi' \sqsubseteq \pi$.

Proposition 3.1.1. *The following propositions hold true of refinement:*

1. $\pi \barwedge \pi' \sqsubseteq \pi$
2. $\pi \sqsubseteq \pi \barvee \pi'$

Proof This easily follows from Definition. 3.1.4, since $\barwedge$ and $\barvee$ are given semantics in terms of intersection and union respectively.

From the definition of the operators we can immediately deduce some useful properties, following easily from their encoding as the basic set operators.

Proposition 3.1.2. *Conjunction, disjunction, and negation of properties have the following properties:*

1. *Conjunction and disjunction are idempotent:* $\otimes \in \{\barwedge, \barvee\} \cdot \pi \otimes \pi = \pi$
2. *Conjunction and disjunction are commutative:* $\otimes \in \{\barwedge, \barvee\} \cdot \pi \otimes \pi' = \pi' \otimes \pi$
3. *Conjunction and disjunction are associative:* $\otimes \in \{\barwedge, \barvee\} \cdot (\pi \otimes \pi') \otimes \pi'' = \pi \otimes (\pi' \otimes \pi'')$
4. *Conjunction and disjunction are distributive over each other:* $\otimes, \ominus \in \{\barwedge, \barvee\} \cdot \otimes \neq \ominus \Rightarrow (\pi \otimes \pi') \ominus \pi'' = (\pi \ominus \pi'') \otimes (\pi' \ominus \pi'')$

5. *Double negation is equivalent to no negation:* $\neg(\neg\pi) = \pi$

Proof These properties follow easily from the definition of the operators in the basic set-theoretic operators.

We assume that the property space Π is closed under these operators, where if π and π' are both in Π then $\pi \barwedge \pi'$ and $\pi \barvee \pi'$ are also both in Π, along with there negations.

Assumption 3.1.1. Π *is closed under* $\barwedge$, $\barvee$, *and* $\neg$.

Moreover we can easily show how refinement means that the larger property accepts all programs of the smaller property.

Proposition 3.1.3. $P \vdash \pi \wedge (\pi \sqsubseteq \pi') \Rightarrow P \vdash \pi'$

Proof $P \vdash \pi \Rightarrow P \in [\pi]$ by definition of $[\pi']$. But $\pi \sqsubseteq \pi' \Rightarrow [\pi] \subseteq [\pi']$, by definition of $\sqsubseteq$, thus $P \in [\pi']$.

While, given a program satisfies a certain property, then the same program violates any other property whose conjunction with the previous property is empty.

Proposition 3.1.4. $P \vdash \pi \wedge ([\pi \barwedge \pi'] = \varnothing) \Rightarrow P \nvdash \pi'$

Proof Note how $[\pi \barwedge \pi'] = \varnothing$ implies that $P \notin \pi'$, since by $P \not\vdash$ we can conlude that $P \in \pi$.

Using the definitions of the property operators in terms of their set-theoretic counterparts in the program space we can easily show the properties form a lattice induced by the refinement relation.

Theorem 3.1.1. $\langle \Pi, \barwedge, \barvee \rangle$ *forms a lattice.*

Proof Note how each two elements of Π have their conjunction ($\barwedge$) as their infimum and their disjunction ($\barvee$) as their supremum.

Property specification languages can allow the specification of the smallest and largest properties, i.e. the property that accepts no program (e.g. *false*) or all programs (*true*). These correspond to the least and greatest element in the lattice.

Proposition 3.1.5. *A property that accepts no programs is a refinement of every program, while every property is a refinement of the property that accepts every program:*

1. $[\pi] = \varnothing \Rightarrow \forall\, \pi' \cdot \pi \sqsubseteq \pi'$
2. $[\pi] = \mathcal{P} \Rightarrow \forall\, \pi' \cdot \pi' \sqsubseteq \pi$

Proof This follows from the definition of refinement using the subset relation on the property semantics.

Moreover, the presence of these properties in the class of properties implies that it does not just form a lattice but a boolean algebra.

Theorem 3.1.2. *Given a property $\pi_\bot$ that accepts no programs, and a property $\pi_\top$ that accepts all programs then $\langle \Pi, \barwedge, \barvee, \neg, \pi_\bot, \pi_\top \rangle$ forms a boolean algebra.*

In the next section we give a foundation for residual analysis in terms of this general semantics.

3.2 Residual Analysis

The problem we want to tackle in this work, as previously described, is to allow a property to be proven using a combination of multiple verification methods. We choose to do this by allowing a property to be proven in parts. We will be using inference rules throughout to illustrate our approach.

In brief, we want some method to check whether a program P satisfies a certain property π (here we use ? to represent an unknown proof obligation):

$$\frac{?}{P \vdash \pi}$$

We can do this by using some appropriate verification method V (e.g. some form of static analysis):

$$\frac{P \vdash_V \pi}{P \vdash \pi}$$

However, this may fail to prove a property π (given Definition. 3.1.2), but may instead only prove that some other property π_1 holds:

$$\frac{\dfrac{P \vdash_V \pi_1}{P \vdash \pi_1} \qquad ?}{P \vdash \pi}$$

Given we have another verification method at our disposal (e.g. RV), a choice is to then similarly use it to attempt to prove π. However, although we may be able to fully prove π using this other method, there may be some overheads associated (e.g. with runtime verification we can prove all safety properties that we could not prove using static analysis, but it induces undesirable overheads on the program at runtime).

Thus, to decrease the load on the second verification method we can decrease the proof obligation, i.e. instead of requiring that π is proven we can instead find a property π_2, such that when both π_1 and π_2 hold of a program then π also holds of the program, i.e. the conjunction of π_1 and π_2 is a refinement of π: $\pi_1 \mathbin{\mathbb{\wedge}} \pi_2 \sqsubseteq \pi$ (e.g. π itself is a candidate for π_2, but choosing π would not encode any progress). If we manage to prove that $P \vdash \pi_2$ then we can conclude that $P \vdash \pi_1 \mathbin{\mathbb{\wedge}} \pi_2$ and thus by using Proposition. 3.1.3 we can conclude that $P \vdash \pi$:

$$\frac{\dfrac{P \vdash_V \pi_1}{P \vdash \pi_1} \qquad \pi_1 \mathbin{\mathbb{\wedge}} \pi_2 \sqsubseteq \pi \qquad \dfrac{P \vdash_{V'} \pi_2}{P \vdash \pi_2}}{P \vdash \pi}$$

However note that in this case we may choose a π_2 such that $P \vdash \pi_2$ is not true, while $P \vdash \pi$ is. An example of a valid π_2 is the empty property $\pi_\perp$, however $P \vdash \pi_\perp$ clearly cannot hold. Then here π_2 is only sufficient to prove π, but we are not assured that if π holds of the program then π_2 will also hold. Thus when we attempt to prove π_2 we may get a violating verdict but we are not able to carry this conclusion onto π. To prevent this we strengthen the condition on π_2 to ensure that π_2 accepts all programs accepted by π, and since we have the knowledge that the program we are interested in satisfies π_1 we can limit π_2 also to programs of π_1.

$$\frac{\dfrac{P \vdash_V \pi_1}{P \vdash \pi_1} \qquad \pi_1 \mathbin{\mathbb{\wedge}} \pi_2 \sqsubseteq \pi \qquad \pi_1 \mathbin{\mathbb{\wedge}} \pi \sqsubseteq \pi_2 \qquad \dfrac{P \vdash_{V'} \pi_2}{P \vdash \pi_2}}{P \vdash \pi}$$

We use these conditions to characterise the notion of a *property quotient*.

Definition 3.2.1 (Residual/Quotient)**.** *Given properties π, π_1 and π_2, property π_2 is said to be a residual or quotient of π with respect to π_1, or $\pi_2 \in \pi \div \pi_1$, if:*

1. *the conjunction of π_1 and π_2 accepts only programs accepted by π: $\pi_1 \mathbin{\mathbb{\wedge}} \pi_2 \sqsubseteq \pi$; and*

2. *the conjunction of π with π_1 accepts only programs accepted by π_2: $\pi_1 \mathbin{\mathbb{\wedge}} \pi \sqsubseteq \pi_2$.*

Note we have define a quotient class, where there may be multiple properties that can function as an appropriate quotient, as opposed to a function. This was done for two reason. Firstly, note how at the level of the language there may be different representations for the same property, and thus by defining a class of quotients we are including all such equivalent properties. Secondly, as we shall see, there are multiple appropriate quotient properties that accept different sets of programs but are equivalent modulo the program being verified. For example, if we want to show that $\pi = A \Rightarrow B$ then showing $\pi_1 = \neg A \vee C$ then both $\pi_2 = \neg A \vee (C \Rightarrow B)$ and $\pi_2' = (C \wedge A) \Rightarrow B$ are appropriate quotients/residuals. With this definition then we are being as general and opinionated as possible, rather than choosing just one quotient operator that may not be appropriate in all cases.

Then the following proof rules are enough for the combination of two verification methods (here static analysis and runtime verification), respectively to prove satisfaction and violation:

$$\text{QUOTSAT}\ \frac{\dfrac{P \vdash_V \pi_1}{P \vdash \pi_1} \qquad \pi_2 \in \pi \div \pi_1 \qquad \dfrac{P \vdash_{V'} \pi_2}{P \vdash \pi_2}}{P \vdash \pi}$$

$$\text{QUOTVIOL}\ \frac{\dfrac{P \vdash_V \pi_1}{P \vdash \pi_1} \qquad \pi_2 \in \pi \div \pi_1 \qquad \dfrac{P \vdash_{V'} \neg\pi_2}{P \nvdash \pi_2}}{P \nvdash \pi}$$

The correctness of these inference rule is ensured by the following theorem.

Theorem 3.2.1 (Correctness)**.** *Given knowledge that program P satisfies property π_1, then checking for a quotient π_2 of π with respect to π_1 is enough to verify π:* $\pi_2 \in \pi \div \pi_1 \quad \Leftrightarrow \quad \forall P : \mathcal{P} \cdot P \vdash \pi_1 \Rightarrow (P \vdash \pi_2 \Leftrightarrow P \vdash \pi)$

Proof Starting from $\pi_2 \in \pi \div \pi_1$, then from $\pi_1 \mathbb{\wedge} \pi \sqsubseteq \pi_2$ and assuming $P \vdash \pi_1$ and $P \vdash \pi$ we can easily conclude π_2, similarly from $\pi_1 \mathbb{\wedge} \pi_2 \sqsubseteq \pi$ and assuming $P \vdash \pi_1$ and $P \vdash \pi_2$ we can easily conclude $P \vdash \pi$. From the left-hand side the proof follows easily in a similar manner.

One such quotient is simply the conjunction of π and π_1, which follows almost immediately from the definition of quotients, giving a general approach to computing a quotient that can be used to combine verification methods.

Proposition 3.2.1. $\pi_1 \mathbb{\wedge} \pi \in \pi \div \pi_1$

Proof The required quotient conditions easily follow: (i) $\pi_1 \mathbb{\wedge} (\pi_1 \mathbb{\wedge} \pi) \sqsubseteq \pi$ follows from associativity and idempotency of $\mathbb{\wedge}$, and since $\mathbb{\wedge}$ computes a property that accepts the intersection of the two properties; and (ii) $\pi_1 \mathbb{\wedge} \pi \sqsubseteq (\pi_1 \mathbb{\wedge} \pi)$ follows immediately.

We call this the *smallest quotient* of π with respect to π_1 since every other quotient contains it, by the second condition of being a quotient.

Proposition 3.2.2. $\pi_2 \in \pi \div \pi_1 \Rightarrow \pi \mathbb{\wedge} \pi_1 \sqsubseteq \pi_2$

We can identify another interesting quotient namely the union of π with the negation of π_1.

Proposition 3.2.3. $\pi \mathbb{\vee} \neg \pi_1 \in \pi \div \pi_1$

Proof The required quotient conditions easily follow: (i) note how $\pi_1 \mathbb{\wedge} (\pi \mathbb{\vee} \neg \pi_1)$ can be resolved to $(\pi_1 \mathbb{\wedge} \pi) \mathbb{\vee} (\pi_1 \mathbb{\wedge} \neg \pi_1)$ using distributivity of $\mathbb{\wedge}$ over $\mathbb{\vee}$, which can further be resolved to $(\pi_1 \mathbb{\wedge} \pi)$ (since the left disjunct corresponds to the empty property), which is a refinement of π, allowing us to conclude that $\pi_1 \mathbb{\wedge} (\pi \mathbb{\vee} \neg \pi_1) \sqsubseteq \pi$ as required; and (ii) $\pi_1 \mathbb{\wedge} \pi \sqsubseteq (\pi \mathbb{\vee} \neg \pi_1)$ follows from the fact that $\pi_1 \mathbb{\wedge} \pi$ is a refinement of π, and since π is a refinement of $\pi \mathbb{\vee} \neg \pi_1$.

This can infact be shown to be the *largest quotient*.

Proposition 3.2.4. $\pi_2 \in \pi \div \pi_1 \Rightarrow \pi_2 \sqsubseteq (\pi \mathbb{\vee} \neg \pi_1)$

Proof Note how from Theorem. 3.2.1 we can conclude that $P \vdash \pi_1 \Rightarrow (P \vdash \pi_2 \Leftrightarrow P \vdash \pi)$. Then if we assume that $P \vdash \pi_2$, there are two cases: (i) $P \vdash \pi_1$, which allows to conclude that $P \vdash \pi$ and then we can introduce a disjunct $P \vdash \pi \vee P \vdash \neg \pi_1$ from which we can conclude that $P \vdash \pi \mathbb{\vee} \neg \pi_1$ as required; and (ii) in the case that $P \vdash \neg \pi_1$ we can immediately introduce a disjunct, $P \vdash \pi \vee P \vdash \neg \pi_1$, from which we can conclude that $P \vdash \pi \mathbb{\vee} \neg \pi_1$.

With these largest and smallest element, the quotient set itself forms a boolean algebra.

Theorem 3.2.2. $\langle \pi \div \pi_1, \mathbb{\wedge}, \mathbb{\vee}, \neg, \pi \mathbb{\wedge} \pi_1, \pi \mathbb{\vee} \neg \pi_1 \rangle$ *is a boolean algebra.*

We have defined a notion of quotients, and identified two kinds of quotients that can be acquired in a property language with conjunction, disjunction, and negation. In the next section we instantiate this abstract theory and consider concrete instances of these quotients.

3.3 Instantiations

In this section we present two example instantiations of this framework, respectfully with properties as state invariants of the program and with properties as sets of event traces. We also give examples of quotient operators for each instantiation.

3.3.1 State-based Analysis

Consider properties specified in a propositional logic, allowing us to specify program invariants. We can then ground programs, properties, and the verification problem over sentences in this logic.

Definition 3.3.1. *Given a set of basic atoms Atoms and a language over them $L(Atoms)$ with conjunction, disjunction, and negation, then:*

- *The class of programs is given semantics as traces of sentences in $L(Atoms)$: $\mathcal{P} = 2^{(L(Atoms))^*}$.*
- *The class of properties is given semantics as a proposition: $\Pi = L(Atoms)$.*
- *The verification operator is then defined as truth of the property at each program state: $P \vdash \pi \stackrel{\text{def}}{=} \forall\, t \in P \cdot \forall\, 0 \leq i < length(t) \cdot t_i \Rightarrow \pi$.*

We can then show that the usual boolean conjunction operators for propositions corresponds to conjunction in the property's verification semantics:

Proposition 3.3.1. $P \vdash \pi \wedge \pi' \Leftrightarrow P \vdash \pi \mathbin{\mathbb{\wedge}} \pi'$

Proof Recall that $[\pi \mathbin{\mathbb{\wedge}} \pi'] = [\pi] \cap [\pi']$, and thus $P \vdash \pi \mathbin{\mathbb{\wedge}} \pi' \Leftrightarrow P \vdash \pi \wedge \vdash \pi'$.

Note how if $P \vdash \pi \wedge \pi'$ then given any proposition trace t in P, any member of this trace t_i satisfies: $t_i \Rightarrow \pi \wedge \pi'$. By the usual definition of boolean conjunction then $t_i \Rightarrow \pi$ and $t_i \Rightarrow \pi'$. Therefore $P \vdash \pi \wedge P \vdash \pi$, and then $P \vdash \pi \mathbin{\mathbb{\wedge}} \pi$. The same argument can be made backwards.

We have similar results for disjunction and negation.

Proposition 3.3.2. $P \vdash \pi \vee \pi' \Leftrightarrow P \vdash \pi \mathbin{\mathbb{\vee}} \pi'$

Proposition 3.3.3. $P \vdash \neg\pi \Leftrightarrow P \vdash \neg\!\!\!\!\sim \pi$

While refinement easily corresponds to implication.

Proposition 3.3.4. $\pi \Rightarrow \pi' \;\Leftrightarrow\; \pi \sqsubseteq \pi'$

In this case then, given knowledge that $P \vdash \pi'$, and the proof obligation that $P \vdash \pi$ we can construct the strict quotient $\pi \wedge \pi'$ and attempt to prove that instead of π directly. The question here is when this is more efficient to check on P rather than just checking for π. This depends if the expression $\pi \wedge \pi'$ can be reduced to a structurally smaller expression.

Example 3.3.1. *Consider that $\pi = A \vee B$, while $\pi' = \neg A$. Then $\pi \wedge \pi_1 = (A \vee B) \wedge \neg A$, which can be reduced to B. If B holds then π holds, clearly.* □

Example 3.3.2. *With $\pi = A \wedge B$, and $\pi' = C$, then it is not clear that attempting to verify $A \wedge B \wedge C$ would be easier that simply attempting π with another method.* □

To attempt to create a structurally smaller quotient proposition, we can instead attempt to simplify π directly: when we already know and have verified π_1, we can analyse π in conjunctive normal form and remove any conjunct that is implied by π'.

Assume π is in conjunctive normal form: $\pi = \bigwedge_{A \in \mathcal{A}} A$, for some set of proposition $\mathcal{A} \subseteq L(Atoms)$. Then we can define the set of basic expressions that are not implied by π', and use this create a quotient of of π with respect to π'.

Definition 3.3.2. *The set of proposition $\mathcal{A}$ reduced by a property π is the set of proposition in $\mathcal{A}$ not implied by π: $reduced(\mathcal{A}, \pi) = \{A \in \mathcal{A} \mid \pi \not\Rightarrow A\}$.*

We can show that using this to reduce our specification leaves an equivalent verification problem.

Theorem 3.3.1. *Given a proposition expression π, in conjunctive normal form, $\pi = \bigwedge_{A \in \mathcal{A}} A$, and given a property π', then the conjunction of the reduced expression set $\mathcal{A}$ with respect to π' is a quotient of π with respect to π' : $\bigwedge_{A \in reduced(\mathcal{A}, \pi')} A \in \pi \div \pi'$.*

Proof If we assume that $P \vdash \pi'$, then we need to prove that $P \vdash \bigwedge_{A \in reduced(\mathcal{A}, \pi')} A \Leftrightarrow P \vdash \bigwedge_{A \in \mathcal{A}} A$, then we prove this bi-implication from both directions: (i) assuming $P \vdash \bigwedge_{A \in reduced(\mathcal{A}, \pi')} A$, then we can add any $A \in \mathcal{A}$ back since if they are not present in $reduced(\mathcal{A}, \pi')$ they are implied by π', and this is sound since we know $P \vdash \pi'$; and (ii) for the other direction we can simply remove $A \in \mathcal{A}$ already implied by π', since we know P satisfies it.

Although it may be difficult to compute this set precisely in general, we can over-approximate it efficiently, e.g. by simply checking for syntactic equality.

Example 3.3.3. *Consider that $\pi = A \wedge B$, while $\pi' = A$. Then $\pi \wedge \pi' = \pi$, which does not reduce the expression. However using the previous theorem (and assuming we have shown π' to be true) we can easily reduce π to B and monitor just for B.* □

In other cases such a useful transformation may not be clear, e.g. $\pi = A \wedge B$ and $\pi' = A \Rightarrow B$. Note that we can define a similar reduction when π is in disjunctive normal form, by removing any disjunct that contradicts π', as illustrated by Example 3.3.1. We consider further partial evaluation of such expressions in Chapter 4, in the context of an industrial case study.

3.3.2 Event-based Analysis

For runtime monitoring, variations on an event trace semantics are standard. In this section we discuss the verification problem with programs, and properties grounded over traces of events.

Definition 3.3.3. *Given an alphabet of events Σ:*

- *The class of programs here is the power set of event traces: $\mathcal{P} = 2^{\Sigma^*}$.*
- *The class of properties is given semantics in terms of event traces: $\Pi = 2^{\Sigma^*}$.*
- *The verification operator is then defined as the subset operation between the program and the property: $P \vdash \pi \stackrel{\text{def}}{=} P \subseteq \pi$.*

Properties here can be, for instance, automata, regular expressions, or temporal logic expressions, i.e. any formalism that can be interpreted in terms of traces.

We can define a conjunction operator between properties whose trace semantics is the intersection of the properties' interpretations.

Definition 3.3.4. *An operator $\|_t : \Pi \times \Pi \mapsto \Pi$ is a* trace intersection operator *if the produced property accepts only the traces accepted by both properties: $\pi \|_t \pi' = \pi \cap \pi'$.*

If the properties are automata then trace intersection here corresponds to standard parallel composition, while for temporal logics we can use their standard logical conjunction.

This trace intersection operator corresponds to the conjunction operator defined over the property's verification semantics.

Proposition 3.3.5. $P \in \pi \mathbb{\wedge} \pi' \Leftrightarrow P \vdash \pi \|_t \pi'$

Similarly, we can define a union operator that corresponds to the disjunction operator in the verification semantics.

Definition 3.3.5. *An operator* $\bigsqcup_t : \Pi \times \Pi \mapsto \Pi$ *is a* trace union operator *if the produced property accepts a trace only if it is accepted by at least one of the properties:* $[\![\pi \bigsqcup_t \pi']\!] = [\![\pi]\!] \cup [\![\pi']\!]$.

Proposition 3.3.6. $P \in \pi \mathbb{W} \pi' \Leftrightarrow P \vdash \pi \bigsqcup_t \pi'$

Negation corresponds to trace language complement.

Definition 3.3.6. *An operator* $\neg_t : \Pi \mapsto \Pi$ *is a* trace negation operator *if the produced property accepts a trace if it is not accepted by the input property:* $\neg_t \pi = \Sigma^* \setminus \pi$.

Proposition 3.3.7. $P \in \neg \pi \Leftrightarrow P \vdash \neg_t \pi$

Moreover, property refinement can be characterized by the subset relation.

Proposition 3.3.8. $\pi \sqsubseteq \pi' \Leftrightarrow \pi \subseteq \pi'$

Then we can easily reproduce the previous results here. In fact given knowledge that a program P satisfies π_1 then we can compute the strict quotient of π with respect to π_1 with trace composition (or intersection): $\pi \|_t \pi_1$. This property thus either accepts the same amount of traces as π (if π_1 includes all of π, or possibly when π accepts an infinite amount of traces) or less.

Working with explicit sets of traces however is not very useful, especially since we may want to specify an infinite set of allowed program traces. Automata-theoretic methods to verification are a common way to express these event trace based properties ([Colombo et al., 2008; Vardi, 2007]), while other logics (e.g. LTL, or regular expressions) can also be expressed as automata. Here then we consider briefly properties as deterministic finite-state automaton (with *bad states* that should not be reached instead of accepting states), and illustrate how their composition can be wielded to produce a quotient.

Definition 3.3.7. *A deterministic finite-state automaton property is a tuple* $\langle \Sigma, Q, q_0, B, \rightarrow \rangle$ *where* Σ *is a set of events,* Q *is a set of states,* q_0 *is the initial state,* $B \subseteq Q$ *is a set of bad states, and* $\rightarrow \subseteq Q \times \Sigma \times Q$ *is the total deterministic transition function, that treats bad states as sink states. We write* $q \xrightarrow{e} q'$ *for* $(q, e, q') \in \rightarrow$. *We use* $\Rightarrow \subseteq Q \times \Sigma^* \times Q$ *as the transitive closure of* $\rightarrow$.

We can then give the semantics of a finite-state automaton property π *in terms of the set of all traces that do not pass through a bad state, i.e. the set of good traces:* $G(\pi) = \{t \in \Sigma^* \mid \nexists t' \in prefixes(t) \cdot q_0 \stackrel{t'}{\Rightarrow} q \wedge q \in B_\pi\}$, *and then a program satisfies* π *if it is a subset of this set:* $P \vdash \pi \stackrel{\text{def}}{=} P \subseteq G(\pi)$.

Conjunction of two properties here is then standard synchronous composition.

Definition 3.3.8. *The synchronous composition of two deterministic finite-state automaton properties π_1 and π_2 is the property $\pi_1 \| \pi_2$ generated with the following rule:*

$$\frac{q_1 \xrightarrow{e} q_1' \qquad q_2 \xrightarrow{e} q_2'}{(q_1, q_2) \xrightarrow{e} (q_1', q_2')}$$

The bad states of $\pi_1 \| \pi_2$ are those tuples of states that are made up of at least one bad state: $B(\pi_1 \| \pi_2) \stackrel{\text{def}}{=} (B_{\pi_1} \times Q_{\pi_2}) \cup (Q_{\pi_1} \times B_{\pi_2})$.

We also have the standard result that synchronous composition results in the language intersection of the two automata.

Proposition 3.3.9. $G(\pi_1 \| \pi_2) = G(\pi_1) \cap G(\pi_2)$

Proof Note how a trace is in $G(\pi_1 \| \pi_2)$ if and only if it does not reach a bad state either in $G(\pi_1)$ or $G(\pi_2)$.

Then we can show that a program is accepted by this composition only if it is accepted by the verification semantics composition.

Proposition 3.3.10. $P \in \pi \between \pi' \Leftrightarrow P \vdash \pi \| \pi'$

Proof This follows from Proposition. 3.3.9 and the definition of $\vdash$ in terms of the property's good traces.

$\|$ then corresponds to the conjunction operator of the program verification semantics, and thus we can use it to compute the strict quotient of two properties.

Theorem 3.3.2. $\pi \| \pi' \in \pi \div \pi'$

Proof In Proposition. 3.3.10 we related $\pi \| \pi'$ to the property intersection, and such property intersections are quotients, by Proposition. 3.2.2.

Example 3.3.4. *Consider that we want to prove that the program never performs event b after event a, i.e. π as in Figure. 3.1(a). If instead we only manage to prove that c never occurs, i.e. a property π' as in Figure. 3.1(b), then we can compute the parallel composition of π and π', and create a property that is equivalent to monitor for π, i.e. Figure. 3.1(c).* □

As in the case with state-based residual verification the residual computed with conjunction may be further simplified. Consider that if we prove $P \vdash \pi_1$ then we are assured that a trace of P will never reach a bad state of π_1, and thus we can remove the states in $\pi \| \pi_1$ that are bad in π_1. We define such a reduced version of the synchronous composition.

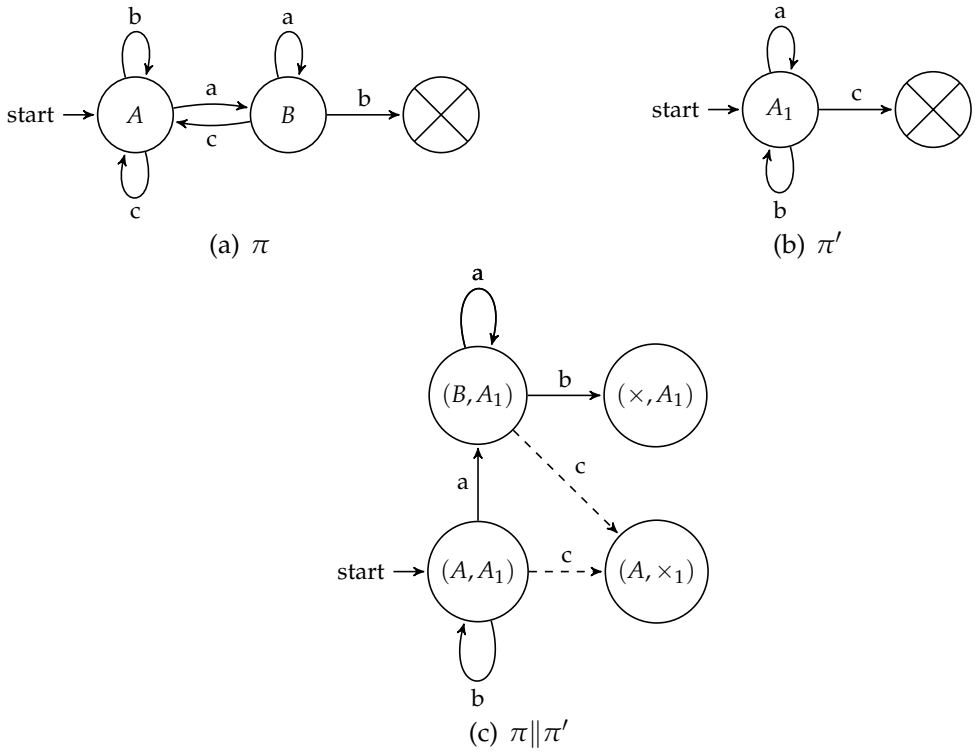


Figure 3.1: Two properties and their composition (and the residual composition without the dashed transitions).

Definition 3.3.9. *The* reduced synchronous composition *of two deterministic finite-state automaton property π and π_1 is the property $\pi \rceil \| \pi_1$ generated with the following rule:*

$$\frac{q \xrightarrow{e} q' \qquad q_1 \xrightarrow{e} q_1'}{(q, q_1) \xrightarrow{e} (q', q_1')} \, q_1' \notin B_{\pi_1}$$

The bad states of $\pi \| \pi_1$ are those tuples of states that containing a bad state of π_1: $B(\pi \rceil \| \pi_1) \stackrel{\text{def}}{=} (B_\pi \times Q_{\pi_1})$.

The reduced synchronous composition of π and π_1 can be shown to be a quotient of π with respect to π_1. This follows from the fact that the synchronous composition is also such a quotient, and that the removed states and transitions cannot be reached or taken by programs that satisfy π_1.

Corollary 3.3.1. $\pi \rceil \| \pi_1 \in \pi \div \pi_1$

Proof Consider that if $P \vdash \pi_1$ then no trace of P reaches a bad state of π_1, then any transitions in $\pi \| \pi_1$ to a bad state of π_1 do not represent any step in a

trace of P, and can be removed soundly. $\pi \lceil \| \pi_1$ is then equivalent to $\pi \| \pi_1$, with respect to the chosen P, and since the latter is a quotient of π and π_1 then so is its reduction.

For example Figure. 3.1(c) illustrates the reduced synchronous composition as the automaton without the dashed transitions, when we know π_1.

In the worst case however synchronous composition may introduce an exponential blowup of states, since it depends on the powerset construction of states. Then it may create a much larger property, which is not ideal if one intends to use it for monitoring. However, since the number states in the property automaton is usually limited this limits the blowup.

3.4 Related Work

The notion of property quotients have been used elsewhere in literature, here we discuss these uses in relation to their use in this work.

One use of quotients is in *compositional reasoning*. In such work programs are assumed to be made up of different parallelly composed components, i.e. there is a program composition operator $\|_{\mathcal{P}} : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$, and the problem is verifying that a program constructed from multiple components satisfies a property: $C_1 \|_{\mathcal{P}} C_2 \vdash \pi$. This can be attempted by creating the composed program $C_1 \|_{\mathcal{P}} C_2$, but it can result in a state-explosion. Instead, Andersen [1995] defines a quotient operator $/\!\!/ : \Pi \times \mathcal{P} \to \Pi$ such that the verification problem $C_1 \|_{\mathcal{P}} C_2 \vdash \pi$ can be transformed into the reduced verification problem $C_2 \vdash \pi /\!\!/ C_1$. Our use of quotients here can be considered to be more general than this, since we are not limited to what is known about a component, but the property used for reduction may be an abstraction of the whole program. A similar use is in *interface theories*, where a quotient operator is used to identify the interface specification a specific component must satisfy, which can be exploited for input-output conformance testing on that component (examples include work by Luthmann et al. [2017] and Noroozi et al. [2013]). This allows one to focus the activity of model-based testing onto components, rather than considering the whole system at once, allowing for more thorough testing.

Quotients have also proven useful in the field of *control theory* ([Ramadge and Wonham, 1989] surveys this area), where quotients are used to identify the part of a specification not ensured by a program, and for which a controller can be synthesised. For example, Arnold et al. [2003] use this to adapt the behaviour of a program to make it compliant with a specification, while Martinelli [2003] use it to secure a program from an external unknown attacker. In these kinds of approaches the objective is not verification but control of a program, where the

residual specification is not satisfied by the original program and a controller is synthesised to coax the program into satisfying it. Raclet [2008] go beyond this and show how an implementation of a component can be synthesised from such a residual, instead of a controller.

The main difference in application between existing work and our work is that our quotients decompose the property into a part that has already been proven and a part that remains to be proven of the program, rather than parts corresponding to certain structural components. Moreover, here we gave general conditions for appropriate quotient operators, unlike existing work that use specific quotients (e.g. Raclet [2008] appear to use a quotient corresponding to a reduced version of what we identify as the largest quotient, while Luthmann et al. [2017] use the smallest quotient).

3.5 Discussion

In our foundation we have limited ourselves to two analysis steps, however Theorem. 3.2.1 can be generalised. Assuming n verification techniques (for example, n kinds of abstraction techniques at different precision levels), we can use the following rule:

$$\text{QUOTSAT}^{+}\ \frac{\dfrac{\forall\, 0 < i \leq n \cdot P \vdash_{V_i} \pi_i}{\forall\, 0 < i \leq n \cdot P \vdash \pi_i} \qquad \forall\, 0 \leq i < n-2 \cdot \pi_{i+2} \in \pi_i \div \pi_{i+1} \qquad \pi_{n-1} = \pi_0}{P \vdash \pi_0}$$

We have also not discussed the possible different power of verification techniques. The rules we defined that combine static analysis and runtime verification will still fail if the quotient property we attempt to runtime verify is not monitorable. Thus this framework in practice requires further analysis, namely some classification of properties into either monitorable and non-monitorable properties. Different notions of monitorability exist in literature. Pnueli and Zaks [2006] characterise monitorability of a property in terms of the existence of an execution for which a verdict can be given on the property at runtime. Others like Falcone et al. [2012a] and Francalanza et al. [2017a] take a universal approach and define monitorability of a property in terms of when all executions can be given a verdict at runtime. Here we focus on this latter notion of *full monitorability*. Aceto et al. [2019a] and Aceto et al. [2019b] describe different historical approaches to monitorability in more detail.

Non-monitorable properties can then be attempted to be proven using static analysis tools, along with any monitorable properties. If non-monitorable properties are left but no further static analysis techniques are left then our attempt at

verification has failed. However, if we can certify that the residual property is a monitorable property then runtime verification suffices. For example Alpern and Schneider [1987] show how a Büchi automata can be decomposed into the conjunction of a safety and a liveness property, where then if the liveness part can be proven pre-deployment then the safety part (or a residual of it) can be left for runtime.

A pertinent aspect of specification logics is the type of semantics chosen for these, specifically whether it is a *linear* or *branching* time semantics. In linear time logics we are interested in the current execution at runtime (making it more popular for RV). On the other hand, branching time logics may also be interested in *possible futures*, making it a more popular logic for pre-deployment verification techniques. There is work that explores these two approaches, considering monitorability in each case [Aceto et al., 2017, 2019a; Francalanza et al., 2017b; Jantsch et al., 2019].

We leave these considerations in the context of our theory for future work, and take the approach of simply limiting our specification language to a monitorable one. The languages we choose in the rest of this work will in fact be limited to (co-)safety properties, i.e. properties that for each trace have a finite witness (which are monitorable as shown by Francalanza et al. [2017a] and Aceto et al. [2019b]).

Here we have limited ourselves to residual properties, while in Chapter 2 we also identified that a partial verification technique can identify the part of the program that is left to prove. In an instantiation of the formal theory presented here this could be encoded as part of the property, where a property can be instantiated as a pair of items, one identifying the part of the program left to verify and the other identifying the behaviour this part must satisfy, defining appropriately the operators piecewise over these.

3.6 Conclusions

In this chapter we have presented an abstract theory for verification, and defined a semantics for properties in terms the programs that satisfy them. We have shown how this forms a boolean algebra. Using a notion of refinement of properties corresponding to the subset relation of the programs they respect, and conjunction in terms of program set intersection, we defined a notion of a quotient of properties. We showed how this quotient can be used to reduce the property that remains to be verified of the program, i.e. given knowledge that program P satisfies π_1, then we can find a property π_2 that is equivalent to π when knowing that P is in the program space accepted by π_1.

We instantiated this framework both for state-based properties, and for event-based properties, exploring the residual analysis that can be made in each case. Although we showed how $\pi /\!\!\backslash \pi_1$ is the smallest quotient in terms of the program semantics, in the instantiations considered we illustrated how this does not ensure that $\pi /\!\!\backslash \pi_1$ is the easiest (the least computationally expensive) quotient to verify, but that some simplifications can aid in reducing it further.

4

An Industrial Case Study

In this section we describe an application of residuals to a real-world industrial context, based on work published in [Azzopardi et al., 2016c,d, 2017a, 2018a].

Taking a step back, we claim there are two complementary main uses for partial verification: (i) providing partial guarantees about a specification; and (ii) reducing work for subsequent verification attempts. This chapter deals with a case study that focuses on the first kind, where the primary objective is not to reduce the overheads of post-deployment analysis (which we deal with in Part II), but instead to be able to give at least some static guarantees that can provide some feedback about the program under verification. Thus we will not be evaluating this approach in terms of how much less computation the second verification step needs to do, but with a qualitative discussion of the benefits added by augmenting the business process of the case study with partial verification.

4.1 Summary

We were engaged by Ixaris Ltd. to develop a compliance engine for an *open payments ecosystem* (OPE)[1], essentially a server that acts as backend for multiple applications that wish to perform payments (e.g. create virtual credit cards and use them). In this context the main issue was how to convince financial service providers to make their license available for applications wishing to perform

[1]The Open Payments Ecosystem has received funding from the European Union's Horizon 2020 research and innovation programme under grant number 666363.

some financial service. Nominally this sounds like a job for pre-deployment verification methods, however this proved impractical for a couple of reasons.

To not limit the number of potential clients, a requirement was to remain technology-agnostic and not put any artificial limitations on the language an application could be developed in. Developing analysis tools for each possible language is not a viable option, given the number of possible languages and the time constraints of the project. Moreover, at runtime an application merely communicates with the OPE and may be hosted on a third-party server. This means that we cannot ensure that the application analysed before deployment is identical to that executing after deployment. With traditional verification then the only option here is to runtime verify for well-behaviour. This leaves us without any static pre-deployment assurances, which increases the risk for financial service providers and decreases their appetite for taking on low-profit applications. Moreover misbehaviour of the applications, which cannot always be prevented at runtime, can have serious legal ramifications for the company.

To mitigate this pre-deployment the developer is required to provide a *model of the assured runtime behaviour* of the application. This model acts like a real-world contract, where different stakeholders evaluate whether they want to support the application by the appropriateness of the contract, or model. If it is found appropriate then the application's behaviour will be controlled for at runtime to ensure it respects the model. The model language used here is *resource-based*, and describes several financial resources used by the application (e.g. credit cards). It can be used to describe both attributes of these resources, relations between them, and some temporal properties. This language was judged by the OPE project team to be sufficient to provide service providers with a description of what the application's business process is, at an appropriate level of abstraction.

In the context of financial services the business process is however not the only concern, there are also the intersecting requirements set by various laws and regulations. Our task was to enable specification and verification of these regulations in the described model-based context. At the outset, with traditional verification techniques, it appears here we have two options, either verify a regulation: (i) against a model; or (ii) against the application's interactive behaviour with the OPE at runtime. Verifying against a model is sound here since we know the it will be enforced at runtime, and thus it is a sound representation of the interactive behaviour with the OPE at runtime.

Ideally then we are able to verify regulations against a model and communicate to the service provider what regulations the application will respect and those it will violate. An issue with this picture is that we found the model language to operate at a different level of abstraction than that required for verifying regulations. This meant that some regulations were independent of the model,

i.e. for such a regulation then there are two programs that each satisfy the model but one which satisfies the regulation and the other that violates it. This meant that while we could give feedback to the service provider, if the application is accepted for onboarding then we have to runtime verify (and sometimes enforce) for these unverified regulations. Moreover, we found that we could also return a residual regulation by pruning from it parts that could be proven against a model.

Compliance with relevant legal regulations is one of the criteria for appropriateness of a model, for which we employ residual analysis to both identify when a model is inconsistent with regulations, when it ensures them, and when it has no bearing on compliance with the regulations. To specify these regulations we developed the *Financial Services Regulations Controlled Natural Language* (FSRCNL) [Azzopardi et al., 2018a]. This language can be used to specify rules in the form of quantified propositions, including several payment-related types and predicates over these. The surface form of the language is a subset of the English language, allowing for more easy interoperability between quality assurance teams and legal experts since the specifications act as their own documentation. Specifications in this language are compiled to a check on the model that either returns a verdict or a residual runtime monitor specified in either the LARVA [Colombo et al., 2009] or VALOUR [Azzopardi et al., 2017a] specification language.

We explore further the described approach in the rest of this section, employing some formal notation where useful for explanation. We start by giving some background on the OPE in Section 4.2, and describing briefly the promised models in Section 4.2.1. We then describe our partial verification framework in Section 4.3. We discuss the work in contrast with related work in Section 4.4 and conclude in Section 4.5.

4.2 The Open Payments Ecosystem

The OPE's aim was to bring together different players in the payment services field, with the ultimate aim of making small-scale payment programmes viable by automating a large part of the business and compliance processes involved.

Payment programmes here effectively are schemes that involve credit or debit cards (examples include gift card or corporate payment schemes), whether physical or virtual. These programmes are expected to be provided through an application (or app), with the payment aspect implemented through the OPE. The business process starts with the *developer* proposing an app that is then matched with an appropriate *programme manager* (PM) and *service provider* (SP). Certain financial services are heavily regulated and thus cannot be carried out by anyone, but only by appropriately licensed persons and organisations. The

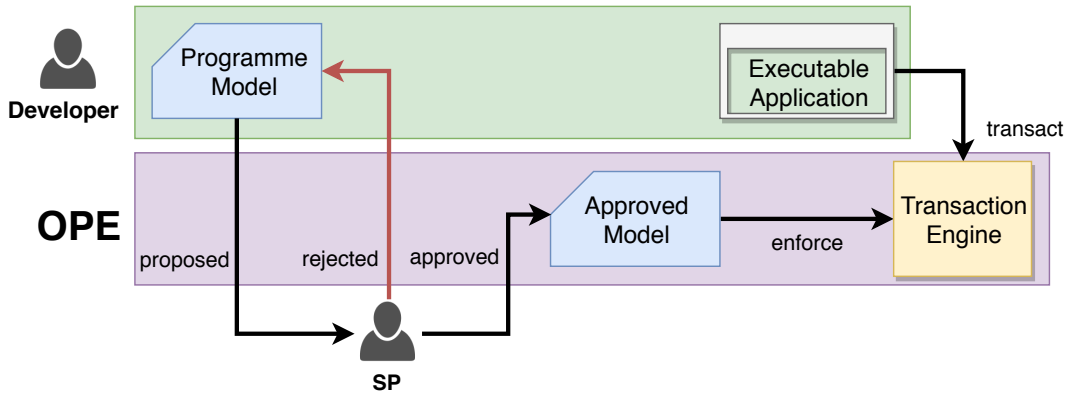


Figure 4.1: OPE business process.

SP, in particular, either accepts or not to provide the app's required payment services under its payment services license. If an appropriate and willing SP is found for an app (possibly after an iterative process of the proposal's refinement) then it can be deployed and its services provided through the OPE. This process includes legal compliance concerns, since an SP will not willingly take on an app that violates applicable regulations. Figure 4.1 illustrates this business process. Our role in this project centered on developing a methodology and associated tools to verify an app's compliance with these regulations, to inject quality and reliability into the ecosystem.

As discussed previously, to remain technology-agnostic the developer is asked to provide a model of the app's runtime behaviour, as a behavioral contract. Pre-deployment we wanted to analyse this model to create a report on the regulations an app respecting the model would violate. This information can both be used by the developer in the development phase, and by the SP when judging whether to accept an app. The specification language for these models (created by the OPE developers) allows for the promised behaviour to be controlled for at runtime, guaranteeing the validity of pre-deployment verdicts. For regulations we created a different more expressive controlled natural language, that maintains a logical structure and semantics while having a surface form close (to an extent) to the original regulations. Given this difference in expressiveness, however, not all regulations turned out to be provable pre-deployment in their entirety, requiring the use of runtime verification techniques post-deployment. We describe briefly the model language to motivate the use of a more expressive regulation language.

Listing 4.1: PAML to describe a card that can belong to consumers or businesses.

```
1  - name: defaultCard
2    type: managedCard
3    instanceCardinality: *
4    relations:
5      - name: owner
6        target: defaultConsumerIdentity
7    constraint:
8      - type: if
9        constraints:
10       - type: eq
11         arguments:
12           - ${state}
13           - ACTIVE
14       - type: eq
15         arguments:
16           - ${owner.state}
17           - ACTIVE
```

4.2.1 Payment Application Models

A developer is required to submit a model of their app written in the *Payment Application Modeling Language* (PAML), which describes the payment aspects of the app as envisaged by the developer in a JSON-like format. This language was developed in-house by Ixaris Ltd, and is not a contribution of this thesis, but we describe it briefly for a better understanding of the context.

Listing. 4.1 is an example of the model of a card written in PAML. It specifies the name of the type of cards being specified (`defaultCard`, line 1), the type they inherit (`managedCards`, line 2), the number of instances there can be of the card (zero or more instances, see line 3), some attributes (the owner attribute is limited to be a consumer, see lines 4-6), and with some constraints on its attributes or those inherited from its supertype (lines 7-17 specify that if the card is active, i.e. not blocked, then the owner is also active). In the lifecycle of the OPE, such models go through an iterative process between the developer and SP, possibly being refined according to SP requirements and may finally be taken on by the SP or rejected.

Here we do not define this language but it should be clear that a PAML model defines constraints on basic OPE constructs.

Our analysis rests on the assumption that at runtime the app respects this model, as performed by the OPE. This controlling was implemented by the OPE developers, and we assume it is implemented correctly here. For example considering Listing. 4.1, any attempt by the application to create a `defaultCard` for a business owner (instead of a consumer as required by lines 5-6) will fail at runtime, while the constraint can be enforced by simply changing the card's state

to an active one once its owner's state is not active.

This language is aimed at the specific exercise of constraining sets of basic OPE types to reflect the promised behaviour of the app at runtime. It is appropriate for this use, and constraints can be used to define a wide range of properties to identify allowed subsets of type. However PAML is limited and cannot be used to define interesting temporal properties, and it turns out to not be enough to encode the regulations we want verify. Moreover PAML is aimed towards developers; it is not immediately communicable to other non-technical stakeholders who are interested in the promised runtime behaviour.

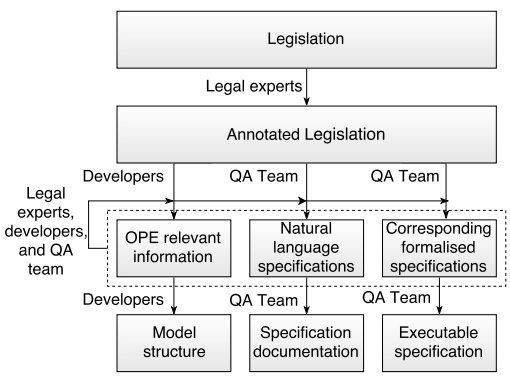
Regulations however require temporal notions. To this end we defined a separate language aimed at regulation specification and verification. Moreover this language can be more easily used to communicate the behaviour that is being verified to non-technical stakeholders, since it uses a a subset of the English language as its surface form. We describe this briefly in the next section, while describing formally the partial evaluation of this language against a model.

4.3 A Partial Verification Framework

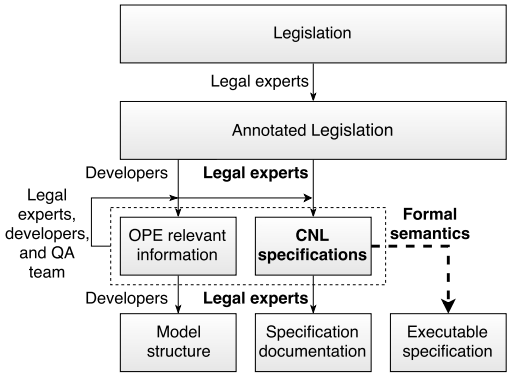
In this section we describe both the design and implementation of a partial verification framework intended for industrial use. A prototype was developed and integrated with the OPE. We focus first on the specification process and language we used. After which we describe the initial approach to pre- and post-deployment verification. We then finish with an overview of how these verification stages where combined using partial verification.

4.3.1 Specification Process and Language

This project brought together financial law experts and developers to identify the applicable regulations that apps must be compliant with. Through an iterative process with these two stakeholders we identified several of these regulations, specifying them both in informal English and formally, as illustrated in Figure. 4.2(a). Logical languages are however not a paradigm of specifications that lawyers and developers are familiar with, each being respectively used to regulations and specifications in natural language. However formality and preciseness is essential here, to remove any ambiguity of natural language. To bridge these two competing interests (precise specifications and understandable specifications) we constructed a *controlled natural language* (CNL) (see [Kuhn, 2014] for a survey of CNLs) — the CNL's surface form is a subset of English, as are the regulations we consider, while sentences in it are deterministically translatable into an executable specification. We call this CNL the *Financial Services Regulations*



(a) Iterative process with legal experts without FSRCNL.



(b) Iterative process with legal experts with FSRCNL.

Figure 4.2: Regulation specification process both without FSRCNL and with automated executable specifications creation using FSRCNL.

CNL (FSRCNL), and Figure. 4.2(b) shows the simplified iterative process using FSRCNL as the base specification [Azzopardi et al., 2018a].

4.3.1.1 Financial Services Controlled Natural Language

We considered several regulations as candidates for automated compliance checking. These ranged from regulations about e-money and payments services, to acts about money laundering and other criminal activities. Not all the clauses in these regulations were relevant to the limited scope of the OPE, while other relevant regulations were not necessarily fully verifiable in an automated manner. For example, we cannot verify fully that the terms and conditions coming with a programme are properly delivered to a customer, or that they conform to every legal requirement. However we can verify whether redeeming (or cashing e-money) occurs within the legally mandated time period. Thus the regulations we wanted to identify in the iterative processes shown in Figure 4.2 were the ones that we could map to OPE-specific constructs. Table 4.2 lists the regulations we considered and the number of verifiable regulations we identified from each. We identified thirty-one such clauses in all.

These regulations include definitions of payment services specific terminology, that we replicate in our controlled natural language, ensuring a correspondence between their definition in the regulations and their semantics with respect to the OPE. For example, the following regulation defines what e-money is.

EMR2(1) *"electronic money" means electronically (including magnetically) stored monetary value as represented by a claim on the electronic money issuer which (a) is issued on receipt of funds for the*

Table 4.2: List of UK regulations considered.

Regulation	Acronym	Verifiable Clauses
The Electronic Money Regulations 2011 (SI 2011/99)	EMR	11
The Payment Services Regulations 2009 (SI 2009/209)	PSR	14
The Money Laundering Regulations 2009 (SI 2009/209)	MLR	4
European Commission's Proposal for a Directive Amending MLD4	MLD5	2

purpose of making payment transactions; [. . .]

We use this definition to identify which OPE cards or instruments deal e-money and those that do not, a distinction that did not exist at the level of the OPE's code and structure.

Other clauses set constraints on certain cards on instruments, for example the following regulation does not allow any benefit to be given on e-money depending on the length of time the e-money is held.

EMR45 *An electronic money issuer must not award (a) interest in respect of the holding of electronic money; or (b) any other benefit related to the length of time during which an electronic money holder holds electronic money.*

This kind of regulation can be checked for easily by checking whether the model allows an e-money instrument to have interest or not. To specify these kinds of regulations then we needed our CNL to be able to refer to common payment constructs, such as interest, money, cards, and transactions.

More specific regulations layout exact bounds based on some monetary limit, for some amount calculated over a certain time period, that cannot be expressed using PAML. For example, the following regulation specifies a limit for reloadable instruments:

ML13(7)(d)(ii) *[. . .] if the device can be recharged, a limit of 2,500 euro is imposed on the total amount transacted in a calendar year, except when an amount of 1,000 euro or more is redeemed in the same calendar year by the bearer [. . .]*

Our CNL thus was designed to incorporate structures and keywords that were expressive enough to specify these kinds of regulations. We detail this language in the next section.

An example of an FSRCNL sentence is the following:

> **For each** programme p and instrument i, **where** p is regulated in the UK, i is an instrument of p, and i deals with e-money, **then** i is prepaid.

Most of the rules defined using FSRCNL are rules specific to a single programme (i.e. all the other variables are limited to constructs of one single programme), instrument, or transaction, while we have one example of a rule over

1. *For each programme p, and instrument i, where p is regulated in the UK, and i deals with e-money, then i does not give time-based rewards.*

2. *For each instrument i, where i deals with e-money, i has expired less than 12 months ago, then e-money in i is redeemed without fees.*

3. *For each programme p, and transaction t, where t is carried out in EUR, t is a cross border payment, the transfer of t is carried out in EUR, and the destination account owner of t is a consumer, micro-enterprise, or charity, then the cash of t is available by one business day after receipt.*

Figure 4.3: Some example regulations specified in FSRCNL.

all the programmes supported by one service provider. Figure 4.3 illustrate some examples of regulations written in FSRCNL.

We gave this language a semantics in terms of runtime monitors. Initially we compiled FSRCNL sentences into DATEs (the LARVA specification language [Colombo et al., 2009]), a powerful language and tool that inlines monitors with code. However, the OPE required the verification process to be implemented in a microservice, precluding inlining of monitors. The VALOUR RV tool (with a guarded command language [Azzopardi et al., 2017a]) was developed for this purpose. Instead of inlining code VALOUR simply processes a stream of events. For example, Listing. 4.2 is the automatically generated VALOUR monitor specification of the second regulation in Figure 4.3.

4.3.2 Partial Verification

The target of verification here is the application's interaction with the OPE's API, which as discussed can be observed at runtime. However, we also have available a promised model of behaviour at runtime, which is actually controlled for by the OPE. Then we can exploit the model by using it as a sound representation of the behaviour that will be observed at runtime and try to verify regulations against it.

For example, if we determine that there are no cards/instruments defined in a model M that deal with e-money then all the rules but rule 3 in Figure 4.3 can immediately be determined to hold true for apps controlled for M (consider that the presence of e-money is required for each of them).

However, since FSRCNL can specify temporal properties of the system and PAML cannot then not all properties are verifiable in this manner. Consider for example that rule 2 is dependent on the length of time from when the instrument expired, while rule 3 sets constraints on the period of time within which the cash

Listing 4.2: Automatically generated VALOUR version of the FSRCNL sentence *For each instrument i, where i deals with e-money, i has expired less than 12 months ago, then e-money in i is redeemed without fees.*

```
declarations {
    category INSTRUMENT indexed by Integer

    event error(String info) = {
        monitor trigger error(String info)
    }
}

replicate {

}
foreach Instrument i {

    declarations {
        event withdrawalStart(Instrument i) = {
            system controlflow trigger Instrument i.withdrawalStart()
            belonging to INSTRUMENT indexed by Integer
        }
    }

    withdrawalStart(i)
        | {{i.isEmoney
            && !i.noFees
            && LocalDate.now().minus(Period.ofMonths(12))
                                    .compareTo(i.expiryDate) > 0}}
        -> {error("Reedemal from instrument i is with fees.")}
}
```

of a transaction has to be made available. A traditional approach to verification would then take a model M and a rule r and try to determine whether the model satisfies the property, possibly returning the unknown verdict when an FSRCNL sentence is in the subset not specifiable by a PAML model. However partial reasoning can aid us here to reduce a rule further.

Consider rule 1 and a model M. Model M represents one programme or application, and specifies some constraints on the programme and on all the instruments that will be part of the application at runtime. In rule 1 we are also talking about all instruments of the programme. Thus if we can match a constraint of instruments in the model with some of the basic sentences parametrised by an instrument in rule 1 we can at least resolve their value. For example, if we resolve the value of *i does not give time-based rewards* to *true* for some model M then we have proven the property follows from the model (consider that if we

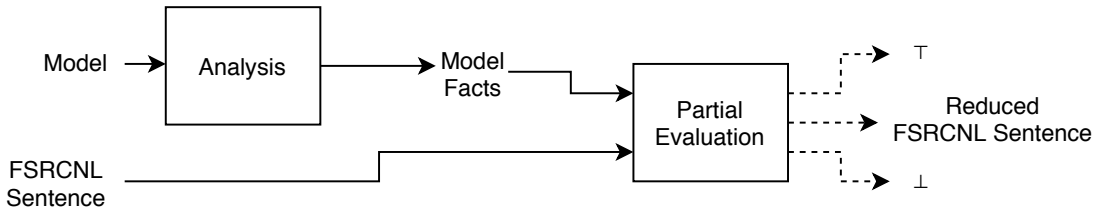


Figure 4.4: High-level view of partial verification process through partial evaluation.

have $a \wedge b \Rightarrow c$ but know c to be true then we can conclude that $a \wedge b \Rightarrow c$ holds true).

This may not necessarily be possible for each basic sentence, since not all basic sentences directly correlate to a constraint expressible in PAML. And even if a basic sentence *fs* is expressible as a PAML constraint, we are not assured that the model will enforce *fs* or enforce $\neg$*fs*, but it may simply allow apps with both. Consider the previous example, and that we manage to resolve only *p is regulated in the UK* against a model, then the property is still undetermined, but we can create a residual property instead of simply returning an unknown verdict, i.e. we can return *For each programme p, and instrument i, where i deals with e-money, then i does not give time-based rewards*. We sketch briefly how such a partial verifier can be constructed next.

4.3.2.1 A Partial Verifier for FSRCNL

Consider that we can infer certain facts from a model that hold universally over all apps that satisfy the model. We can then *partially evaluate* an FSRCNL sentence against the facts computed from model, which we use both as our verification strategy and to produce a residual FSRCNL sentences, as illustrated in Figure 4.4.

For illustrative purposes and simplicity instead of talking about FSRCNL we can simply talk about a propositional language. Note how variable declarations in FSRCNL are not needed since basic sentences are typed and thus we can easily infer the types of identifiers. We can also infer quantification, since we only have the universal quantifier.

Then consider a simple language over some set of propositions: $\mathbb{P}$, with conjunction, disjunction, negation, and implication: $L(\mathbb{P})$. Using *AppExec* for the set of possible application runtime executions of the app in question, we can characterise monitoring as a predicate that evaluates a sentence against an execution.

Definition 4.3.1. *A monitor evaluates sentences against an app execution at runtime:* $rv : L(\mathbb{P}) \times AppExec \mapsto Bool$.

We can similarly characterise a model as a predicate over application executions accepting only those executions that satisfy it. We also assume that the model can be analysed, returning a set of basic propositions that are true or false of the applications respecting the model.

Definition 4.3.2. *A* payment model *is a predicate over an application. The set of payment models is represented by* $\mathbb{M} \subset AppExec \mapsto Bool$, *and a payment model by* $M \in \mathbb{M}$.

An analysis of a model M is a set of proposition, $analysis(M) \in 2^{\mathbb{P}}$, *that hold on any app that respects the model:* $M(exec) \Rightarrow \forall\, p \in analysis(M) \cdot rv(p)(exec)$.

We can then define a reduction operator on a sentence from $L(\mathbb{P})$ that partially reduces/evaluates it according to a model. An example of how this operator works is illustrated by considering the sentence $a \wedge b$ and the set of propositions $\{a\}$, where we can reduce $a \wedge b$ by considering that a is true and thus the truth of $a \wedge b$ only depends on the value of b. This kind of reasoning can be used to partially evaluate a sentence pre-deployment leaving either a verdict or a residual sentence to be verified at runtime.

Definition 4.3.3. *The partial evaluation of a sentence from* $L(\mathbb{P})$ *with respect to a set of propositions* $P \subseteq \mathbb{P}$ *is the function* $pv : L(\mathbb{P}) \times 2^{L(\mathbb{P})} \mapsto \{\top, \bot\} \cup L(\mathbb{P})$ *defined as follows:*

1. $s \Rightarrow s'$ *is immediately satisfied by P iff either s is violated by P, or* s' *is satisfied by P. It is immediately violated if s is satisfied by P but* s' *is violated. Otherwise, if s is satisfied by P the partial evaluation of* s' *is returned, and if not the partial evaluation continues on both s and* s'.

$$pv(s \Rightarrow s', P) \stackrel{\mathsf{def}}{=} \begin{cases} \top & pv(s, P) = \bot \\ \top & pv(s', P) = \top \\ \bot & \begin{array}{l} pv(s, P) = \top \\ \wedge\, pv(s', P) = \bot \end{array} \\ pv(s', P) & pv(s, P) = \top \\ pv(s, P) \Rightarrow pv(s', P) & otherwise \end{cases}$$

2. $s \wedge s'$ *is violated by P iff either s or* s' *is violated, and satisfied if both are satisfied. Otherwise, if only one of s or* s' *satisfies, partial evaluation continues in the other sentence, and if neither is satisfied the partial evaluation continues on both s and* s'.

$$pv(s \wedge s', P) \stackrel{\text{def}}{=} \begin{cases} \bot & pv(s,P) = \bot \\ \bot & pv(s',P) = \bot \\ \top & pv(s,P) = \top \\ & \wedge\ pv(s',P) = \top \\ pv(s',P) & pv(s,P) = \top \\ pv(s,P) & pv(s',P) = \top \\ pv(s,P) \wedge pv(s',P) & \textit{otherwise} \end{cases}$$

3. *$s \vee s'$ is satisfied by P iff either s or s' is satisfied, and violated if both are violated. Otherwise, if only one of s or s' is violated, partial evaluation continues in the other sentence, and if neither is satisfied the partial evaluation continues on both s and s'.*

$$pv(s \vee s', P) \stackrel{\text{def}}{=} \begin{cases} \top & pv(s,P) = \top \\ \top & pv(s',P) = \top \\ \bot & pv(s,P) = \bot \\ & \wedge\ pv(s',P) = \bot \\ pv(s',P) & pv(s,P) = \bot \\ pv(s,P) & pv(s',P) = \bot \\ pv(s,P)\ \textbf{\textit{or}}\ pv(s',P) & \textit{otherwise} \end{cases}$$

4. *If a sentence is just a proposition p, then if it is contained in P the satisfaction verdict is returned. If the negation of p is contained in P then the violation verdict is returned. Otherwise p is returned.*

$$pv(p, P) \stackrel{\text{def}}{=} \begin{cases} \top & p \in P \\ \bot & \neg p \in P \\ p & \textit{otherwise} \end{cases}$$

Then we can use such a partial evaluator on an FSRCNL sentence and the analysis of a model, producing a verdict or a smaller property.

For example, assume a model M that contains only cards respecting the PAML script in Listing. 4.1 and a rule *For each instrument i, where the owner of i is a consumer and CDD is not performed on the user of i, then the amount redeemed from i within a calendar year is less than EUR 1000*[2]. Analysing the model, against the qualified sentences of the rule, we can conclude that *the owner of i is a consumer* is ensured by the model, i.e. *the owner of i is a consumer* $\in$ *analysis*(M). Then the partial evaluator will discharge *the owner of i is a consumer* statically, leaving

[2]CDD here means *customer due diligence*, which is a check on the background of the owner of an instrument

For each instrument i, where CDD is not performed on the user of i, then the amount redeemed from i within a calendar year is less than EUR 1000 for runtime.

We can then show that this partial verifier gives results that are connected to the model at runtime. Recall how the model language used here is not able to represent temporal properties, while the specification language can. Some propositions in $\mathbb{P}$ may then be temporal propositions which we may not be able to prove from the propositions extracted from the model.

Theorem 4.3.1.

1. *If the partial verifier signals compliance then this compliance carries over at runtime for apps that satisfy the model:* $pv(fs, M) = \top \Rightarrow (M(exec) \Rightarrow rv(reg, exec))$.
2. *If the partial verifier signals violation then this violation carries over at runtime for apps that satisfy the model:* $pv(reg, M) = \bot \Rightarrow (M(exec) \Rightarrow \neg rv(reg, exec))$.
3. *If the partial verifier reduces a sentence then monitoring for the reduced sentence is equivalent to monitoring for the original sentence for apps that satisfy the model:* $pv(reg, M) = reg' \Rightarrow (M(exec) \Rightarrow (rv(reg, exec) \Leftrightarrow rv(reg', exec)))$.

Proof The result follows easily by case analysis on the definition of *pv*.

Note how the third point in the theorem corresponds to the notion of a residual as we define in Chapter 3.

4.4 Discussion and Related Work

In this case study the approach is slightly different from the motivation given in the previous chapter. Instead of proving a property of a program and using this to reduce a required specification we are assuming that a certain model (M) will be verified or controlled for post-deployment, and we exploit this pre-deployment to decompose other rules (e.g. ρ) with respect to this model so that we do not replicate work at runtime (i.e. we do not monitor parts of ρ that are already ensured by M). In the notation used in the previous section, this approach can be represented as follows (where $\div$ corresponds to our partial evaluation):

$$\frac{\rho' \in M \div \pi \qquad \dfrac{P \vdash_{RV} M}{P \vdash M} \qquad \dfrac{P \vdash_{RV} \pi'}{P \vdash \pi'}}{P \vdash \pi}$$

In literature we find similar approaches to deal with un-analysable code. *Model carrying code* is presented by Sekar et al. [2003] in the context of untrusted binary code (as opposed to source code) that is not easily analysable. Instead it is paired by its developer with a model of behaviour pre-deployment, checked at the same stage for consistency with certain security policies, and the program proved to respect the model at runtime. Dragoni et al. [2007] propose a similar approach, *security-by-contract*. In the context of mobile applications they propose that an application is distributed with a certain security policy contract, that can be enforced at runtime. Aktug and Naliuka [2008] present an automata-based language to specify such contracts and policies for programs, inlining checks into the application when its source code is available and assuming appropriate hooks otherwise. The difference in our case is that the code is simply not available in a trusted version, while we partially evaluate the required properties with respect to the assured model, instead of simply attempting to wholly prove or disprove it. In our context we are also more limited, since we cannot directly instrument a program, but can only monitor its interaction with a certain interface. This limits what we can hope to prove, e.g. we cannot detect that two applications are communicating with each other through some other means, which may be relevant to detect suspicious behaviour. Our properties then are only contracts about how the interface should be used and what it should allow, rather than security policies.

Our approach to partial evaluation here can be applied to any propositional language with conjunction, disjunction, negation, and/or disjunction. For such logics we find a more limited notion of *partial entailment* in literature, characterised by Zhou and Zhang [2011] as a relation that identifies when a proposition entails part of another proposition (e.g. $x \wedge y$ partially entails $x \wedge z$, since x is present in both). With respect to this work, we can say that partially evaluation has a reduction effect if the FSRCNL sentence is partially entailed by a fact of the model, or by the negation of such a fact. The procedure *pv* is in fact a generalisation of the semantics of the Kleene three-valued valued logic (the classical Boolean logic with an unknown truth-value) [Kleene, 1952], where instead of returning an unknown verdict we return a statement that remains to be proven. In the temporal world we also similarly find LTL_3, a three-valued logic used for runtime verification, where if a finite trace cannot be given a verdict then an inconclusive result is given. This method corresponds to partial evaluation of source code [Jones et al., 1993], where a program is specialised to certain variable assignments, which can be used to optimise a program for certain environments. In effect this is the same motivation here, except we are optimising a specification that corresponds to a first-order propositional language instead of program source code.

Here we have presented a simplified view of the approach we implement. Consider that a regulation can quantify over all transactions. In the simple context presented we are simply declaring an application as violating when it has one violating transaction, where for verification purposes we can then stop monitoring since we have shown the application to be violating. However, in a real-world context a certain level of violations is tolerated, and thus monitoring continues, detecting any further violations. In fact, upon receiving a request for a transaction, the OPE blocks it for a set amount of time, and if it receives a violating verdict from a monitor it cancels the transaction. However, if the set of time elapses the transaction is allowed to end successfully. Here then optimising monitoring is a benefit to the system, in that it can lead to more timely evaluations of transactions. The notion of partial evaluation of regulations presented here is one approach we took to attempt to reduce the size of such monitors.

This work was developed with the view of being part of the OPE's business process. By the time of writing the whole process has not been deployed for real-world clients, and although we have a working property, we have thus not had the opportunity to quantitatively evaluate the approach in a real-world application. However this case study shows how partial verification has valuable applications in giving partial static guarantees about the behaviour of programs.

4.5 Conclusions

We have illustrated an industrial application of partial verification methods using residuals. This approach allows us to attempt to provide some static guarantees without the presence of source code, given a promised model of runtime behaviour. These pre-deployment conclusions in fact will be used as part of the iterative process between a service provider and an application developer, where the service provider is kept abreast of the regulations satisfied, violated or not ensured by the current iteration of the model. Moreover, the use of residuals allows us to reduce checks that have to be done at runtime, which is important given the real-time nature of the environment.

5

Conclusions

We have motivated the need to combine different verification methods, given limited resources and the suitability of different techniques to handle different kinds of properties. We reviewed literature presenting approaches that allow partial reasoning by producing artifacts to signal the progress made towards a given verification problem. We classified these approaches as either property transforming or state space transforming, and identified that there is no formal general characterisation in this literature of the notion of a residual property.

To fill this gap we described an abstract formal framework of properties and programs, and characterised the notion of residuals in terms of property quotients. By plugging in different formalisms in this framework (which turns out to be quite simple for state- and event-based formalisms, as illustrated in Section 3.3) we have immediately two ways to construct residuals: the smallest quotient $\pi /\!\!\mathbb{M} \pi_1$ and the largest quotient $\pi \mathbb{W} \neg \pi_1$. While to show that any other strategy to construct a quotient is correct we can simply show the residual property is between (w.r.t. property refinement) these two residuals. In literature we see examples of use of quotients in the context of interface theory, to identify the specification required out of unknown components [Luthmann et al., 2017; Raclet, 2008], while they can also be used to synthesize implementations or controllers for such components [Costa et al., 2018]. To fill this gap we described an abstract formal framework of properties and programs, and characterised the notion of residuals in terms of property quotients. By plugging in different formalisms in this framework (which turns out to be quite simple for state- and event-based formalisms, as illustrated in Section 3.3) we have immediately two ways to construct residuals: the smallest quotient $\pi /\!\!\mathbb{M} \pi_1$ and the largest quotient $\pi \mathbb{W} \neg \pi_1$. While to show that any other strategy to construct a quotient is correct

we can simply show the residual property is between (w.r.t. property refinement) these two residuals. In literature we see examples of use of quotients in the context of interface theory, to identify the specification required out of unknown components [Luthmann et al., 2017; Raclet, 2008], while they can also be used to synthesize implementations or controllers for such components [Costa et al., 2018].

Our framework does not commit itself to a certain formalism, but instead simply assumes a certain structure out of the property space, namely a boolean algebra. This allows our arguments to hold over different formalisms with the required properties. In our use of the identified quotients we have focused on the smallest quotient $\pi \mathbin{/\!\!\wedge} \pi_1$, which for event-based properties simply allowed us to use standard synchronous composition to compute the language intersection. In future work we can consider techniques to compute the largest quotient $\pi \mathbin{\backslash\!\!\vee} \neg \pi_1$ and evaluate how it compares in syntactic size and utility with the smallest quotient.

Part II

Residual Analysis for Automata with Variable State

6

Introduction

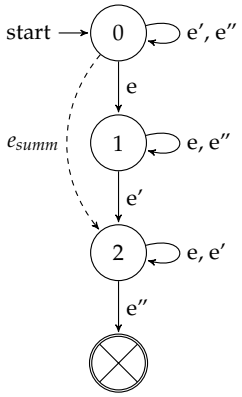
6.1 Context

In literature (see Chapter 2) we found many different approaches that produce residual verification problems. However these were mostly focused on the state-based paradigm of verification. Instead we are mainly interested in event-based verification, for which the approaches we identified were few and each with their limitations.

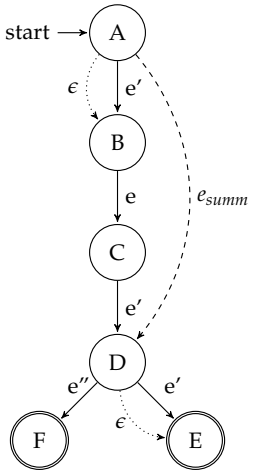
To motivate our work we briefly describe existing event-based partial verification methods using the example property and program in Figure 6.1, both as automata. The dashed and dotted transitions represent transformations made by the methods, as will be described further on. The property here specifies that no program trace can contain events e, e', and e'' in that sequence, possibly separated by other events. Note how in this toy example we can easily determine that the program branch leading to state F is violating, and thus RV is not needed in this case, but this may not be easy to determine for larger programs.

6.1.1 Existing Literature

One of approach is that of Bodden et al. [2010], who present the CLARA tool that is able to remove or silence event instrumentation that does not affect the verdict at runtime given at runtime. For example, consider the property specified by the non-dashed transitions in Figure. 6.1(a), and the program in Figure. 6.1(b). Consider that the transition from state A to state B in the program never causes a change in state in the property, but only ever matches the looping transition



(a) Monitor automaton specifying as bad any traces interleaved with the trace e; e'; e''; (recall that a crossed state represents a bad state). The dashed transition is added after Dwyer and Purandare [2007] analysis.



(b) Automaton representing program behaviour. CLARA analysis replaces transitions between A and B, and between D and E with the dotted transition, while Dwyer and Purandare [2007]'s analysis replaces the sub-automaton between A and D with the dashed transition.

Figure 6.1: Example property and program automata, with dotted and dashed transitions representing the effects of existing residual analysis approaches.

at state 0 in the property. Then this program transition can be ignored by the monitor at runtime, since it has no effect on the property. This is represented in the figure by the dotted transition labeled by with the silent event ϵ, representing no event instrumentation. A similar result holds for the transition between states D and E. This analysis can be lifted to sequences of such transitions, that always loop to the same state in the property automaton.

A more general approach is that of Dwyer and Purandare [2007] in the same context of analysis of Java programs, where they propose to summarise sequential parts of the program that are determined statically to always behave in the same way. This is a more general approach that takes into account not just sequences of program instrumentation that have no effect together, but that enables the determination that the effect of such sequences can be pre-computed pre-deployment. For example, consider the non-dashed transitions between states A, B, C, and D in the program Figure. 6.1(b). Note how we can determine that this sub-automaton always has the same entry property state (i.e. 0), and the same exit property state (i.e. 2). Then we can encode this statically compiled information by replacing these program transitions with the dashed transition

tagged by a new event e_{summ}, and add a similar transition in the property between the determined entry and exit property states. At runtime then the program triggers the monitor only once before state D.

Both these approaches are applied to typestate properties, i.e. properties about objects that can be represented by parametrising the monitors by variables and instantiating them at runtime. Monitors are then replicated at runtime for each such instantiation. For static analysis a sound static abstraction of the possible instantiations is used, to allow for a determination of when events can occur on the same object (and thus can occur in the same monitor). Another feature of these approaches is that the kind of events they consider are events corresponding to method calls, while events in general can also encode some notion of state.

Consider that for a method `multiple(int x, int y)` we can specify an event that matches each call of this method, denoted by $multiple(x, y)$. We can also define an event that matches calls to this method only when one of the arguments is even, which we can represent with by adding a guard as follows: $multiple(x, y)$ | `x % 2 == 0 || y % 2 == 0`. The approaches proposed by Bodden et al. [2010] and Dwyer and Purandare [2007] are validated with events without such guards, while in fact adding event guards can decrease their applicability. This is since in these approaches statically there is no abstraction of the program variable state that allows us to determine the possible values of a guard at a certain point in the program. Then, when a program event can activate a monitor transition associated with a guard we have to branch in two directions in the monitor, representing the cases that the guard holds (i.e. transitioning to the next state of the monitor) and that it does not (i.e. staying in the state of the monitor). The effect these guarded events will have on the monitor at runtime can be more varied than vanilla events. This reduces the probability of finding corresponding instrumentation that can be statically determined to always have the same effect (since the effect of events with guards depends on the program variable state). Then to analyse properties with such event guards we need further tools that abstract the program variable state, which Bodden et al. [2010] and Dwyer and Purandare [2007] do not attempt.

We identified one approach that can resolve such guards, to an extent. Chimento et al. [2015] employ the `KeY` theorem prover [Ahrendt et al., 2016] in their StaRVOOrs tool to prove automata that correspond to functional contracts of method calls, i.e. pre- and post-conditions of a certain method. These automata have the following simple form: $0 \xrightarrow{entry(method)|pre} 1 \xrightarrow{exit(method)|!post} \times$, where the events respectively match the entry into and exit from a method. The analysis performed is not *context-sensitive*, i.e. it only looks at the implementation of the method referenced, ignoring the points in the program that may call it. This may be enough to prove these automata, while in other cases it may not be. In

the latter case the pre-conditions can be made stricter (to identify the cases left to prove) while the post-conditions can be made weaker (to identify parts of it that were not proven to always hold). However, this approach is limited, since the automata considered are very simple. A property can be more sophisticated and can specify behaviour of different methods in relation to each other, e.g. we may want to specify post-conditions on a method when it is called by another method. STARVOORS is then data-oriented, but lacks pre-deployment analysis of properties with more sophisticated control-flow, leaving them for runtime. Differently from Bodden et al. [2010] and Dwyer and Purandare [2007], here STARVOORS reduces the property directly instead acting on the instrumentation associated with monitoring the property.

6.1.2 Unexplored Research Areas

From this work we can identify two areas for further research:

(i) the residual analysis of other kinds of monitor specification languages; and

(ii) the residual analysis of automata with guarded events combining both control-flow and data-oriented static analysis.

Other approaches to specifying monitors include regular expressions, and temporal logic-based approaches. These have intimate connections with automata, in fact it is a standard result that regular expressions can be translated into finite-state automata, while Giannakopoulou and Havelund [2001] show how the standard translation from linear temporal logic (LTL) to Büchi automata can modified to translate LTL into finite-state automata that are appropriate for RV. Then, residual analysis for automata can be applied to these approaches by using such a translation. Another approach to RV is the rule-based one [Barringer et al., 2007], where a set of facts is maintained about the program and the monitor is described with a set of rules that possibly activate upon a program event and act on the set of facts, which is a more general approach than automata with transitions tagged with guarded events.

6.2 Contributions

Here we will be focusing on (ii), where for properties we consider a slight extension of finite-state automata with guarded events and with a monitoring variable state, allowing for transitions tagged by rules similar to those used in rule-based monitoring. To represent the control-flow of a program we also use automata, as used in the example here. However to be able to reason about a program's

variable state we also extend this representation to make explicit the program variable state. Moreover we allow transitions to be guarded by conditions on this variable state (allowing for conditional branching) and to possibly transform it, while also allowing for calls between such automata (representing method calls).

Our approach combines different aspects of existing approaches:

- event- and state-based verification paradigms;
- static and runtime verification;
- control- and data-based analysis of programs and properties; and
- property and program residuals.

6.3 Outline

We define some formal preliminaries in Chapter 7, by defining the DEA and CFA formalisms. In Chapter 8 we present our main contribution, the residual analysis, while in Chapter 9 we evaluate this approach with several programs in both Java and the Solidity smart contract language. We discuss this approach and conclude in Chapter 10. This part is based on work published in [Azzopardi et al., 2016b, 2017b,c, 2019, 2020a,b].

7

Properties and Programs

This chapter provides the formal background necessary for the analysis we present. The view of programs and monitors taken here is that of, respectively, *event producers* and *event consumers*, where a program produces some events that the monitor consumes while possibly updating its internal state. A monitor is passive and only acts upon being triggered by the program. In some work a monitor can also act on the program state to prevent a violation [Azzopardi et al., 2018b; Easwaran et al., 2006], but here we assume a monitor can only act on its internal state for simplicity.

We will introduce *Dynamic Event Automata* (or DEAs) as automata with some internal variable state, and transitions tagged with events, guards on the internal and program variable state, and actions on the internal state. These have an appropriate operational semantics that is used to define what it means for a program trace to be compliant with a DEA monitor. DEAs correspond to safety properties of a program, i.e. properties identifying a set of finite traces that should not be the prefix of any program execution.

We present a generic representation of sequential programs in the form of a variant of control-flow graphs, which we call *control-flow automata* (CFAs). These automata model conditional branching, program statement execution, and event instrumentation by operations on transitions (in a similar fashion to DEAs), while they also allow for calls to other CFAs (modeling function calls) and silent events. These automata allow us to present our program analysis approach in a rigorous technology-agnostic manner (with the techniques we present applying for any program that can be represented as a CFA).

Both DEAs and CFAs here are given an operational semantics. The techniques presented will be agnostic of the specific background theory chosen for the

variable states, in fact we leave these underspecified. We characterise compliance of a CFA with a DEA in terms of the event traces produced by a CFA not being included in the bad traces accepted by a DEA. Our main contribution here will be a residual analysis for properties as DEAs and programs as CFAs. This is based on the synchronous composition of these two automata, and the fact that this composition abstracts the monitored system at runtime. By analysing this composition we can identify the parts of the DEA that can be used by the CFA to reach a verdict at runtime, enabling us to reduce the DEA appropriately. Moreover we show how we can dually remove event instrumentation in the CFA. However, since the composition may be expensive to compute for a flattened CFA of large programs instead we consider abstractions of these programs based on their method CFAs, an approach inspired by Bodden et al. [2010]'s approach and extended here to allow us to reduce the DEA. Moreover, to tackle the data-oriented aspect of DEAs and CFAs we analyse the composition for transitions that cannot be taken at runtime through the use of a satisfiability modulo theory (SMT) solver (to identify DEA guards that cannot hold at a certain program state), enabling us to further reduce a DEA and also to remove guards from some DEA transitions.

We introduce DEAs in Section 7.1, and CFAs in Section 7.2. Furthermore, in Section 7.3 we discuss correctness condition residual property and instrumentation in this context.

7.1 Dynamic Event Automata

Dynamic Event Automata (DEAs) are extensions of finite-state automata with some variable state, and with transitions tagged by triples of: (i) program events, (ii) guards over both the monitor's and the program's variable state, and (iii) actions that can change monitor's variable state.

DEAs can also include a notion of typestate, which we leave formally unspecified here for simplicity. Figure 7.1 is an example of a DEA, where transitions are tagged with events, guards, and actions in a pattern `e | g` $\mapsto$ `a`, and some object type (representing a variable for which the DEA must be replicated for at runtime). A brief note on the semantics of DEAs is that each transition triggers on a program event, where if there is no matching transition then the DEA remains in the same explicit state.

Figure 7.1 also specifies some local variable state, namely an integer variable `length`. The DEA specifies that `next()` function of the iterator should not be called more times than the length of the list (i.e. the violating state labeled by $\times$ should not be reached). This example illustrates how DEAs are infinitely more succinct that finite-state automata, allowing for an arbitrary number of explicit

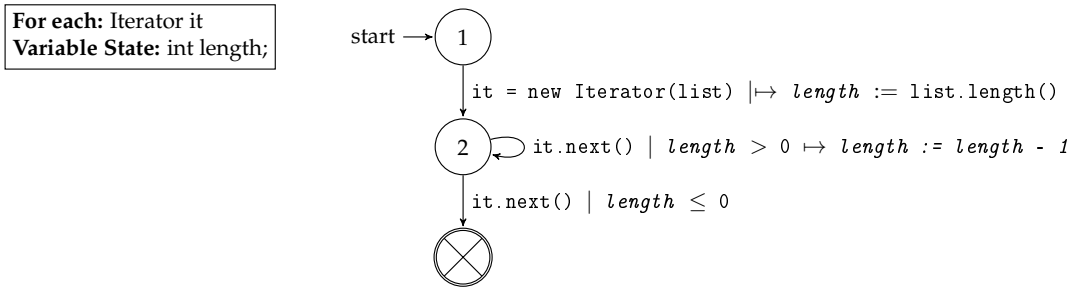


Figure 7.1: DEA specifying that an iterator over a list should not be queried for more elements than it has.

states to be symbolically encoded. Note how Figure 7.1 cannot be expressed without a monitoring variable state for an *a priori* unknown length.

Another motivation for DEAs is that they allow us to be more precise in our specifications. Consider the finite-state property illustrated in Figure. 7.2(a). This specifies that `hasNext()` has to be called on an iterator before `next()` is called. However, this popular example does not actually capture bad usage of an iterator. For example, the code in Listing. 7.1 will be judged as satisfying against Figure. 7.2(a), since `hasNext()` is called before `next()`, however the other two snippets in Listing. 7.2 and Listing. 7.3 will also be judged as satisfying although they obviously exhibit unwanted behaviour.

The subtle differences between these snippets of code are not captured by an automaton with vanilla events like Figure. 7.2(a). Using a DEA however we can also capture these incorrect and dangerous patterns: Figure. 7.2(b) in fact allows a `next()` to occur successfully only when `hasNext()` is true before it, and detects a violation otherwise. Thus Listing. 7.2 would be classed as unsafe, since `hasNext()` always returns false before `next()` is called, while the safety of Listing. 7.3 depends on the runtime state of the program.

(a) Finite-state automaton specifying that `hasNext()` should always be called before `next`.

(b) DEA specifying that an iterator over a list should only be queried for the next element when it also signals that it has a next element.

Figure 7.2: `hasNext()` property as a finite-state automaton and a more powerful version as a DEA.

Listing 7.1: Snippet using iterator correctly.

```
1 if(it.hasNext()){
2     it.next();
3 }
```

Listing 7.2: Snippet using iterator incorrectly by only calling `next()` when there is no next element.

```
1 if(!it.hasNext()){
2     it.next();
3 }
```

Listing 7.3: Snippet using iterator dangerously by not conditioning the call `next()` on there being a next element.

```
1 it.hasNext();
2 it.next();
```

Currently we can give these determinations for compliance with a DEA by instrumenting the code with the DEA's logic and returning any determined violating verdicts at runtime. DEAs as described here are in fact a kernel sub-language of the monitoring specification languages used for multiple programming languages in the LARVA suite of runtime monitoring tools [Azzopardi et al., 2018b; Colombo et al., 2008]. These languages also include: (i) timers and time-triggered events; (ii) channels that monitors can use to communicate with each other; and (iii) typestate. Here we ignore these, although we describe informally how they can be dealt with in Chapter 10.

DEAs are similar to *event automata* as used by Barringer et al. [2012], except that here we are leaving the variable state abstract instead of working concretely with variables. In effect both of these are a form of extended finite-state automata [Alagar and Periyasamy, 2011].

7.1.1 Definitions

Dynamic event automata extend finite-state automata with a symbolic monitoring state and guarded commands on transitions between explicit states. In practice the variable state is generally some list of assigned variables defined in the language of the program being monitored, while transition guards and actions are statement blocks in the same programming language (with the requirement that guards have no side-effects). Formally guards are represented as predicates over pairs of program and monitoring variable states, actions as transformations of monitoring variable states, also parametrised by a program variable state.

Explicit states of a DEA can be *accepting* or *bad*, where a monitor in an accepting states cannot be violated given any continuation, while a monitor in a bad state signals a violation of the property. The formal definition of DEAs along this informal description follows.

Definition 7.1.1. *A* dynamic event automaton *(DEA) over a set of system events* Σ *and a set of symbolic program variable states* Ω *is an automaton* $\pi = \langle \Theta, Q, q_0, \theta_0, B, A, \rightarrow \rangle$*, where:*

(i) Θ *is a (possibly infinite) set of symbolic variable states,*

(ii) Q *is the finite set of explicit monitoring states,*

(iii) $q_0 \in Q$ *is the initial explicit state,*

(iv) $\theta_0 \in \Theta$ *is the initial symbolic variable state,*

(v) $B \subseteq Q$ *is the set of explicit bad states,*

(vi) $A \subseteq Q$ *the set of explicit accepting states, and*

(vii) $\rightarrow \subseteq Q \times \Sigma \times Guard \times Act \times Q$ *is the deterministic finite transition function that is triggered upon some system event, if a guard* ($Guard \stackrel{\text{def}}{=} \Omega \times \Theta \mapsto Bool$) *on the program and monitoring variable state holds, while if triggered it performs an action* ($Act \stackrel{\text{def}}{=} \Omega \times \Theta \mapsto \Theta$) *on the symbolic monitoring state.*

We write $q \xrightarrow{e|g \mapsto a} q'$ *for* $(q, e, g, a, q') \in \rightarrow$*, and use* $\Rightarrow$ *for its transitive closure. We write* $q \Rightarrow q'$ *for* $\exists\, t \in (\Sigma \times Guard \times Act)^* \cdot q \stackrel{t}{\Rightarrow} q'$*. Moreover we use false and true respectively for the always false guard* ($\lambda\, \omega, \theta.\ false$) *and the always true guard* ($\lambda\, \omega, \theta.\ true$)*, and skip for the identity action* ($\lambda\, \omega, \theta.\ \omega$)*.*

Throughout we use e for events in Σ*, g for guards in Guard, and a for actions in Act, possibly labeled by some index. We use* Π *for the class of DEAs.*

We expect that a program will produce event and program variable state tuples, which upon being received by a DEA will possibly cause it to update its internal state (explicit and/or symbolic). We call these *program traces*.

Definition 7.1.2. *A* program trace *is sequence of event and variable state pairs, ews* $\in \Sigma \times \Omega$.

We then give DEAs an operational semantics with configurations as pairs of property explicit and variable states, and transitions between these labeled by pairs of property event and program variable states. This semantics allows for transitions from non-accepting and non-bad states in the DEA to be taken only if its guard holds on the program and monitor variable state, with the internal state evolving according to the transition's action. Otherwise the transition will have no effect. Note how this means that any configuration with an accepting or bad state stops evolving.

Definition 7.1.3. *The operational semantics of a DEA is given as a labelled transition system over configurations of type* $Q \times \Theta$ *with transitions labelled by labels from* $\Sigma \times \Omega$*, and characterized by the following rules:*

(i) Given a transition $q \xrightarrow{e|g\mapsto a} q'$*, some program state* ω *and some symbolic monitoring state* θ*, then if* q *is not an accepting or bad state, the condition of the transition holds for* ω *and* θ*,* (q, θ) *transitions to* $(q', a(\omega, \theta))$ *upon event e with program state* ω*:*

$$\frac{q \xrightarrow{e|g\mapsto a} q' \qquad q \notin A \cup B \qquad g(\omega, \theta)}{(q, \theta) \xrightarrow{e,\omega} (q', a(\omega, \theta))}$$

(ii) if the previous rule does not hold then an event and variable state pair transition to the same configuration:

$$\frac{(q \in A \cup B) \vee (\forall q', e, g, a \cdot (q \xrightarrow{e|g\mapsto a} q') \Rightarrow \neg g(\omega, \theta))}{(q, \theta) \xrightarrow{e,\omega} (q, \theta)}$$

Note how by the second rule here DEAs are monotonic, i.e. once they give a verdict it is maintained. Note how by the second rule here DEAs are monotonic, i.e. once they give a verdict it is maintained.

This operational semantics is a *concrete semantics* for DEAs as opposed to a *symbolic semantics*. Note how the symbolic label ω in our case represents one value and not multiple values as in other approaches [Calder and Shankland, 2001; Francalanza, 2017].

From this operational semantics we can identify when a trace is compliant with the property or not by checking whether it reaches a bad state or not.

Definition 7.1.4 (Program Trace Compliance)**.** *The set of violating traces of a DEA are the set of program traces that reach a bad state:* $V(\pi) \stackrel{\text{def}}{=} \{ews \mid (q_0, \theta_0) \stackrel{ews}{\Longrightarrow} (q, \theta) \wedge q \in B\}$.

A program trace $ews \in \Sigma \times \Omega$ *is said to satisfy a DEA* π *if and only if it is not a violating trace of pi:* $ews \vdash \pi \stackrel{\text{def}}{=} ews \notin V(\pi)$.

7.1.2 Structural Analysis

A DEA is not necessarily syntactically optimal, in fact it may have useless (in terms of the semantics) states and transitions. Figure. 7.3(a) illustrates such a DEA. It is unlikely that a specification designer would create such a DEA, however the residual operations we define later on may produce similar DEAs motivating the need for minimising operations for DEAs.

Here we discuss when structural reductions to DEAs are safe, in that the same violating traces remain violating, and give examples of such reductions. These operations will be essential to produce an optimised residual of a DEA against a CFA.

7.1.2.1 Safe Structural Reductions

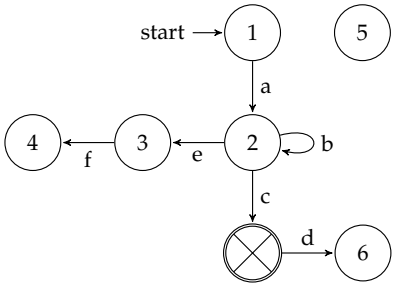
To carry out these reductions safely we first give a notion of DEA equivalence — two DEAs will be considering as semantically equivalent if they are associated with the same violating traces.

Definition 7.1.5. *A DEA* π *is said to be* semantically equivalent *to a DEA* π' *iff they have the same violating traces:* $\pi \equiv \pi' \stackrel{\text{def}}{=} V(\pi) = V(\pi')$.

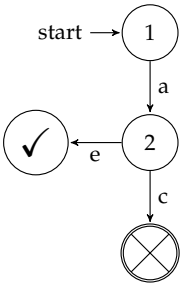
Here we will limit ourselves to *structural reductions* to a DEA, where transitions and states are pruned. We want any such reduction to maintain semantic equivalence, but also to be structurally equivalent or smaller. Then we first identify formally when a DEA is a sub-structure of another by considering their respective states, transitions, and variable state.

Definition 7.1.6. *A DEA* π *is said to be a sub-structure of a DEA* π' *iff* π *is defined over a subset of the states of* π'*, botg have the same initial explicit and variable states, the bad states of* π *are a subset of those of* π'*, and the transitions of* π *are a subset of those of* π'*:* $\pi \sqsubseteq \pi' \stackrel{\text{def}}{=} Q(\pi) \subseteq Q(\pi') \wedge q_{0_\pi} = q_{0_{\pi'}} \wedge \theta_{0_\pi} = \theta_{0_{\pi'}} \wedge B(\pi) \subseteq B(\pi') \wedge \rightarrow_\pi \subseteq \rightarrow_{\pi'}$.

Note that we do not require that the accepting states of a sub-structure are a subset of the original DEA, since we will be identifying transformations that



(a) Unoptimised DEA.



(b) Optimised DEA.

Figure 7.3: DEA before and after optimisations.

may bring an acceptance forward, allowing for an accepting verdict to be given earlier.

We then define structural reductions as reductions of a DEA that preserve this sub-structure relation, while producing a DEA that is semantically equivalent.

Definition 7.1.7. *A* structural reduction *of DEAs is a semantic-equivalence-preserving transformation between DEAs where the reduced DEA is a sub-structure of the original DEA. If* $SR : \Pi \mapsto \Pi$ *is a structural reduction then* $\forall\, \pi \in \Pi \cdot \pi \equiv SR(\pi) \wedge SR(\pi) \sqsubseteq \pi$.

We can then start identifying some structural reductions to a DEA that allow us to produce an optimally reduced DEA. The claims made in this section are quite trivial, and the proof method should be clear from the text.

A simple structural reduction is one for reachability, where we can remove any state that is not reachable from the initial state. For example, in Figure. 7.3(a) we can remove state 5 without changing the semantics of the property.

Definition 7.1.8. *The* reachability reduction *of a DEA* π, $R : \Pi \mapsto \Pi$, *removes any states that are not reachable from an initial state:*

$$
\begin{aligned}
Q_{R(\pi)} &\stackrel{\text{def}}{=} \{q \in Q_\pi \mid q_0 \Rightarrow q\} \\
B_{R(\pi)} &\stackrel{\text{def}}{=} \{q \in B_\pi \mid q_0 \Rightarrow q\} \\
A_{R(\pi)} &\stackrel{\text{def}}{=} \{q \in A_\pi \mid q_0 \Rightarrow q\} \\
\rightarrow_{R(\pi)} &\stackrel{\text{def}}{=} \{(q, e, c, a, q') \in \rightarrow_\pi \mid q, q' \in Q_{R(\pi)}\}
\end{aligned}
$$

It easily follows that this is a structural reduction since such states cannot be reached from the initial state of the property and thus cannot be involved in identifying a program trace as violating.

Proposition 7.1.1. *R is a structural reduction.*

Moreover, consider that the given semantics of DEAs treats accepting and bad states as sink states, where once a configuration with an accepting or bad state is reached then it stops evolving. Thus we can ignore any DEA transition outgoing from an accepting or violating state, and reduce the DEA for reachability. Note how in Figure. 7.3(a) the transition from the bad state to state 6 will never be triggered.

Definition 7.1.9. *An* after-verdict reduction *of a DEA π, $AV : \Pi \mapsto \Pi$, removes any transitions and states that are only used or reachable from a violating or satisfied state.*

We define an intermediate DEA π' identical to π' but without any transitions outgoing from accepting or bad states, i.e. with the following transition relation:

$$\rightarrow_{\pi'} \stackrel{\text{def}}{=} \{(q, e, c, a, q') \in \rightarrow_{\pi} | \; q \notin A \cup B\}$$

Then we define $AV : \Pi \rightarrow \Pi$ as the reachability reduction of π': $AV(\pi) \stackrel{\text{def}}{=} R(\pi')$.

Removing these transitions is again a structural reduction, since such transitions from accepting or bad states will never trigger.

Proposition 7.1.2. *AV is a structural reduction.*

We can also identify states in a DEA that are in effect accepting but are not marked as such, namely states that cannot reach a bad state, and mark these as accepting. State 3 is such a state in Figure. 7.3(a), where although there is another state reachable from it no bad state can be reached.

Definition 7.1.10. *An* acceptance identification transformation *of a DEA π, $AI : \Pi \mapsto \Pi$, marks any state that cannot reach a bad state as accepting:*

$$A_{AI(\pi)} \stackrel{\text{def}}{=} A \cup \{q \in Q_{\pi} \mid \nexists q' \in B \cdot q \Rightarrow q'\}$$

This is clearly a structural reduction since the relevant sub-structures are not changed, and any state that is made accepting could not cause a violation by virtue of not being able to reach a bad state[1].

Proposition 7.1.3. *AI is a structural reduction.*

The final reduction we define removes single transition loops on the same states without actions, since these are already handled implicitly by the DEA semantics. We can also remove transitions with a *false* guard. The transition around state 2 in Figure. 7.3(a) is of this form.

[1] In some cases we may still want a monitor to be able to log some events even after an acceptance verdict can be given, however here we simply are interested in violating traces.

Definition 7.1.11. *A* no effect reduction *of a DEA π, $NE : \Pi \mapsto \Pi$, removes any looping transition that has no effect on the variable state, and any transition with a false condition:*

$$\rightarrow_{NE(\pi)} \stackrel{\text{def}}{=} (\rightarrow_\pi \setminus \{(q, e, c, skip, q) \in \rightarrow_\pi\}) \setminus \{(q, e, false, a, q') \in \rightarrow_\pi\}$$

Since both of these transitions never can be used to transition to a new configuration then removing them does not affect the semantics of the DEA.

Proposition 7.1.4. *NEL is a structural reduction.*

We then assume throughout that any given DEA is in optimal form. We use $OR : \Pi \mapsto \Pi$ to denote the optimal reduction $OR(\pi) = AV(AI(NE(\pi)))$. Applying this to Figure. 7.3(a) results in the smaller but equivalent DEA illustrated in Figure. 7.3(b).

7.1.2.2 Structural Union and Intersection

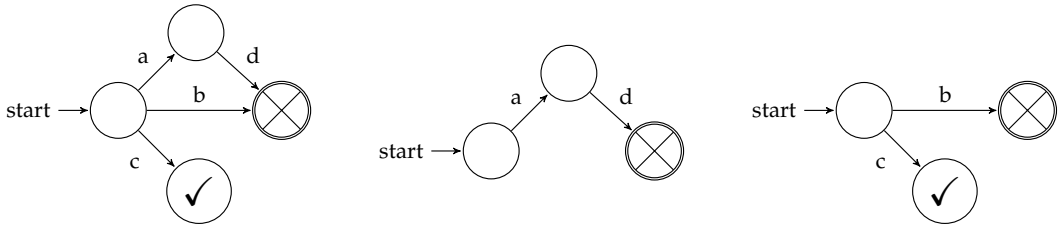
The residuals we produce will be produced piece-wise with respect to each method of the CFA. To join these residuals into one residual for the whole CFA we need to define the structural union of two properties, while we further define the reduced union that ensures two properties are optimally reduced before being joined. The structurally union of two properties will be defined as the property constructed from the union of each feature of a property.

Definition 7.1.12 (Structural Union)**.** *The* structural union *of a DEA π' with a DEA π'', where both are sub-structures of a DEA π, is defined as the union of their respective elements: $\pi' \sqcup\!\sqcup \pi'' \stackrel{\text{def}}{=} \langle \Theta_\pi \cup \Theta_{\pi'}, Q_\pi \cup Q_{\pi'}, q_0, \theta_0, B_\pi \cup B_{\pi'}, A_\pi \cup A_{\pi'}, \rightarrow_\pi \cup \rightarrow_{\pi'} \rangle$.*

The reduced union *of a DEA π' with a DEA π'', where both are sub-structures of a DEA π is defined as the union of their optimal reductions: $\pi' \sqcup \pi'' \stackrel{\text{def}}{=} OR(\pi') \sqcup\!\sqcup OR(\pi'')$.*

We will also illustrate how we can create multiple abstractions of the same set of executions, from which we may infer different residuals. In this case we will show how we can consider the common features of each residual rather than their union, and for this we will require a notion of structural intersection, which is dual to the structural union.

Definition 7.1.13 (Structural Intersection)**.** *The* structural intersection *of a DEA π' with a DEA π'', where both are sub-structures of a DEA π, is defined as the intersection of their respective elements: $\pi \sqcap\!\sqcap \pi' \stackrel{\text{def}}{=} \langle \Theta_\pi \cap \Theta_{\pi'}, Q_\pi \cap Q_{\pi'}, q_0, \theta_0, B_\pi \cap B_{\pi'}, A_\pi \cap A_{\pi'}, \rightarrow_\pi \cap \rightarrow_{\pi'} \rangle$.*



(a) Original DEA, π (and $\pi' \sqcup \pi''$).

(b) Sub-DEA, π'.

(c) Sub-DEA, π''.

Figure 7.4: Example DEA (assume each transition is tagged with a *true* condition and the *skip* action) with two sub-structures.

The reduced intersection *of a DEA π' with a DEA π'', where both are sub-structures of a DEA π is defined as the intersection of their optimal reductions: $\pi' \sqcap \pi'' \stackrel{\text{def}}{=} OR(\pi') \sqcap\!\sqcap OR(\pi'')$.*

Note that this structural union (resp. intersection) does not necessarily result in an automaton that accepts the union (intersection) of the violating traces of the original DEAs. For example, consider the DEAs in Figure 7.4, where $\pi', \pi'' \sqsubseteq \pi$. We clearly have that the trace $\langle cad \rangle$ is in the violating traces of π', but it is not in the violating traces of $\pi' \sqcup \pi''$ (which is equal to π in this case).

In this section we have identified some safe structural reductions to DEAs, and other operations that will be useful for the residual analysis we present. The reductions identified are general and applicable to an individual DEA, which is usually limited in size. In the general case we are however interested in monitoring a DEA in the context a program, and by taking into account the control-flow of the program we can tailor the DEA monitor for that program. We formalise a notion of programs as automata in the next section which we shall be using to this end.

7.2 Control-flow Automata

A natural representations of programs that is used commonly for static analysis is its control-flow graph (e.g. Beyer et al. [2018b] use control-flow automata with operations on transitions to represent programs in the context of creating a residual program). Such a graph makes explicit the control-flow between the statements of a program, creating an abstraction for all its possible executions. Here we consider a variant of these graphs which we call *control-flow automata* (CFAs),

that are similar to DEAs but instead of specifying the disallowed behaviour they encode the actual behaviour of a sequential program.

CFAs, like DEAs, are extensions of finite-state automata with variable state, and with transitions tagged by triples of: (i) conditions over the program's variable state; (ii) statements that can transform the program's variable state; and (iii) an event triggered upon the statement's execution. At this level, CFAs are quite similar to DEAs, both having transitions tagged with events, predicates, and variable state transformers, with the only difference being that CFAs are self-contained, while DEAs also depend on the state of the CFA. Differently from DEAs, we allow for CFAs to reference other CFAs at states, encoding method calls. CFAs will be used to encode to actual behaviour of an implementation, as opposed to DEAs that explicitly the encode a set of disallowed behaviour. In fact, a CFA can be encoded as a DEA, if it is flattened and by introducing appropriate transitions to a bad state, however it easier and simpler to reason about a specific program's behaviour by encoding what it does rather than what it does not do. Calls in a CFA also allow us to define programs piece-wise, which in turn will allow for piece-wise residual analysis foregoing the need to expend computation and memory to create a flat representation of a program, as we shall see.

Consider the Java method specified in Listing. 7.4. This method iterates through an integer `List` and extracts the even numbers in it. It uses both loops, if-then-else branching, and calling methods (e.g. the call to the `ArrayList` constructor on line 2). Figure 7.5 illustrates a CFA corresponding to this method, instrumented for event set $\Sigma = \{hasNext, next\}$, with the silent event ϵ denoting no event, and a shaded state denoting a call site with an associated call label. Note how this CFA models some method calls with call sites while leaving others as simple conditions or statements (e.g. the constructor `new ArrayList<Integer>()` and the iterator method `numbers.iterator` are both modeled as call sites, while calls to `hasNext` are treated as conditions). In general we will only model method calls with call sites when the called method can activate internally events of interest. We will be assuming a method call stores the return value to an appropriate location in the variable state (e.g. for the call associated with state A the associated return value is pointed to by the appropriately typed variable **callsARet** used in its outgoing transition.)

7.2.1 Definition

These automata maintain some internal program state, which we leave ungrounded here, leaving our contributions agnostic of the background theory, as we do for DEAs. This internal variable state can be updated by statements, which we model as transformations of the variable state. A control-flow au-

Listing 7.4: Example Java function.

```
1  public List<Integer> extractEvenNumbers(List<Integer> numbers){
2      List<Integer> evenSubList = new ArrayList<Integer>();
3
4      Iterator it = numbers.iterator();
5
6      while(it.hasNext()){
7          int no = it.next();
8          if(no % 2 == 0){
9              evenSubList.add(no);
10         }
11     }
12
13     return evenSubList;
14 }
```

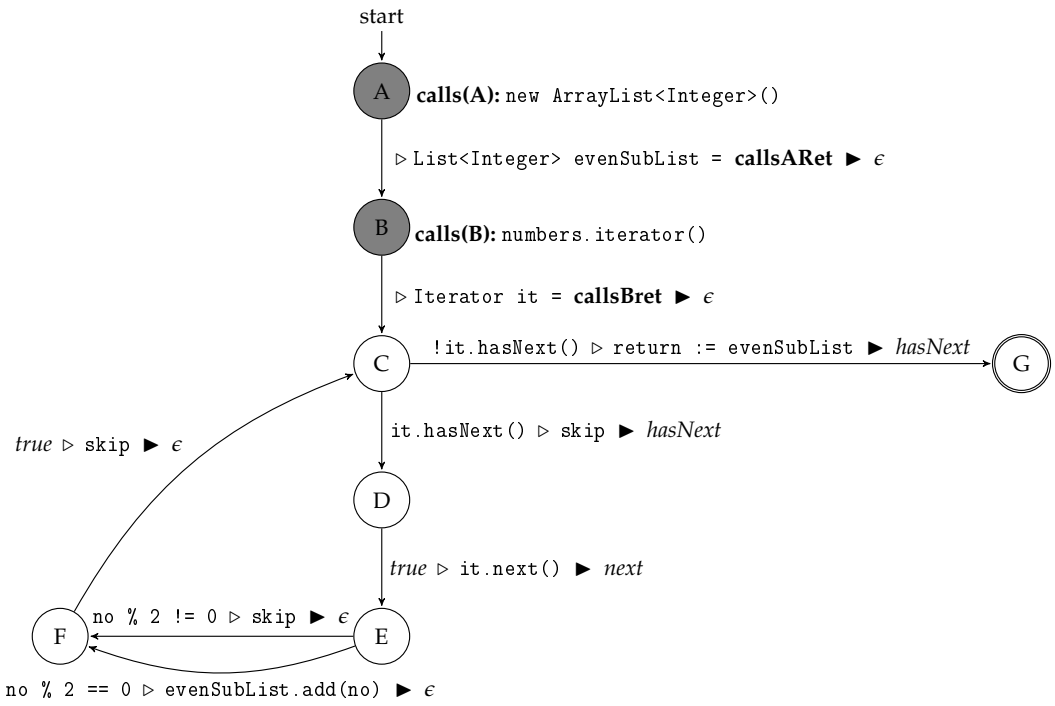


Figure 7.5: Listing. 7.4 as CFA.

tomaton will have a set of transitions tagged with these statements. To allow for conditional branching these transitions can also be tagged with predicates on the variable state, and to mark the points in the program that trigger some property event we also allow transitions to be tagged by a property event or the silent ϵ

Listing 7.5: Example Java code that depends on the runtime state.

```
1 public void f(Object o1, Object o2){
2     if(o1.equals(o2)){
3         <some-logic>
4     }
5 }
```

event. We use Σ_ϵ for this event set, i.e. $\Sigma_\epsilon \stackrel{\text{def}}{=} \Sigma \cup \{\epsilon\}$. Then, CFA transitions are tagged by a triple $c \triangleright st \blacktriangleright e$, where c is a condition on the program variable state, st is a program statement, and e is the event triggered after st is executed.

Moreover, since in real-world programs methods can call other methods, here explicit program states can be *call states*. These call states, at runtime, may call other control-flow automata. Which automaton called is sometimes a decision that has to be pushed to runtime — consider the Java method in Listing 7.5, which accepts two objects and calls the `equals` method of the first object. The method call at runtime depends on the dynamic type of `o1` rather than the implementation associated with the `Object` class. This is an example of a *virtual method*, where dynamic dispatch is required to resolve which concrete method is called at runtime. In our representation thus a call is dependent on the dynamic variable state at the time of the call. We give a formal definition of these automata along these lines.

Definition 7.2.1. *A* control-flow automaton *(CFA) parametrised by a set Ω of program variable states, and a set of events Σ, is a tuple $M = \langle S, s_0, E, calls, \to \rangle$, where:*

(i) *S is a finite set of states;*

(ii) *$s_0 \in S$ is the initial state;*

(iii) *$E \subseteq S$ is a set of end states, we use s_E for an element of this set;*

(iv) *$calls : S \nrightarrow (\Omega \mapsto CFA)$ is a partial function from states to calls; and*

(v) *$\to \subseteq (S \setminus E) \times \mathtt{Cond} \times \mathtt{STMT} \times \Sigma_\epsilon \times S$ is the transition relation between non-end states that is activated when the condition holds true ($\mathtt{Cond} \stackrel{\text{def}}{=} \Omega \mapsto Bool$), where then the statement is executed ($\mathtt{STMT} \stackrel{\text{def}}{=} \Omega \mapsto \Omega$) and an event possibly triggered (if the silent event ϵ is not used).*

We write $s \xrightarrow{c \triangleright st \blacktriangleright e} s'$ for $(s, c, st, e, s') \in \to$, and $\Rightarrow$ for its transitive closure. We use $\mathtt{true}$ for a condition that returns true for every program state ($\lambda\, \omega.true$), and $\mathtt{skip}$ for the identity statement ($\lambda\, \omega.\omega$). $s \xrightarrow{\triangleright st \blacktriangleright e} s'$ is to be interpreted as $s \xrightarrow{\mathtt{true} \triangleright st \blacktriangleright e} s'$, and similarly we write $s \xrightarrow{c \triangleright \blacktriangleright e} s'$ for $s \xrightarrow{c \triangleright \mathtt{skip} \blacktriangleright e} s'$.

We use labels P and M, possibly indexed, to refer to elements of CFA.

Here we will be assuming one entry-point into the program for simplicity. Throughout then we will be assuming a set of methods as a set of control-flow automata, with a main method that we call the program P. We identify the set of all methods of a program P, which we require further on.

Definition 7.2.2. *The* methods of *a CFA P is the smallest set including P and the methods it calls transitively: $methodsOf(P) \stackrel{\text{def}}{=} \{P\} \cup \bigcup_{call \in ran(calls)} \bigcup_{M \in ran(call)} (\{M\} \cup methodsOf(M))$.*

To produce the events required for monitoring and define when a CFA is compliant with a DEA we define an operational semantics for CFAs. A natural choice for program configurations are pairs of explicit and variable states. This is however not fully adequate in the presence of calls. Upon reaching a call state control must be relinquished to another automaton, until an end state is reached upon which it must relinquish control back to the caller state. Thus during the called execution the configurations must keep track of the caller state. One solution is to use stacks of these explicit-symbolic pairs, popping off the top pair when an end state is encountered. Another simpler method is simply to step over, or take a big-step over calls. In practice we have found the latter approach easier to reason and prove with, and thus this is the choice we take here.

To identify when a call has been entered and when it finishes we augment explicit states with an arrow denoting whether a call has been entered or not: $S^{\uparrow\downarrow} \stackrel{\text{def}}{=} S \times \{\uparrow, \downarrow\}$. Then a transition from a configuration $(s^{\downarrow}, \omega)$ to a configuration $(s^{\uparrow}, \omega')$, where s is a call state, will denote that the call has the effect of transforming the variable state from ω to ω'. For non-call states these transitions will not have any effect on the variable state. This solves the problem without requiring complex data structures which may be harder to reason about. However it will make the definition of our operational semantics recursive.

To encode these big-steps over calls we allow for transitions in operational semantics to be tagged by program traces, rather than a single event and variable state pair. Transitions in the operational semantics are then of the form: $\rightarrow$: $(S^{\uparrow\downarrow} \times \Omega) \times (\Sigma \times \Omega)^* \times (S^{\uparrow\downarrow} \times \Omega)$. Moreover, instead of considering the standard transitive closure of $\rightarrow$, which would create a relation of the type $(S^{\uparrow\downarrow} \times \Omega) \times ((\Sigma \times \Omega)^*)^* \times (S^{\uparrow\downarrow} \times \Omega)$, we consider a flat version of the transitive closure of $\rightarrow$. This corresponds to the usual transitive closure with concatenation of traces.

Definition 7.2.3. *The* flat transitive closure *of a relation $\rightarrow$: $(S^{\uparrow\downarrow} \times \Omega) \times (\Sigma \times \Omega)^* \times (S^{\uparrow\downarrow} \times \Omega)$ is the smallest relation $\Rightarrow$: $(S^{\uparrow\downarrow \times} \Omega) \times (\Sigma \times \Omega)^* \times (S^{\uparrow\downarrow} \times \Omega)$ respecting the*

following conditions given $s^-, s_{\bar{1}}^-, s_{\bar{2}}^-, s_{\bar{3}}^- \in S^{\uparrow\downarrow}$ *and* $\omega, \omega_1, \omega_2, \omega_3 \in \Omega$*: (i)* $(s^-, \omega) \overset{\langle\rangle}{\Rightarrow} (s^-, \omega)$*; and (ii)* $((s_{\bar{1}}^-, \omega_1) \xrightarrow{ews} (s_{\bar{2}}^-, \omega_2) \wedge (s_{\bar{2}}^-, \omega_2) \overset{ews'}{\Longrightarrow} (s_{\bar{3}}^-, \omega_3)) \Rightarrow (s_{\bar{1}}^-, \omega_1) \overset{ews +\!\!+ ews'}{\Longrightarrow} (s_{\bar{3}}^-, \omega_3)$.

Note that we are re-using the transition relation symbols used for DEAs, however it should always be clear from the context of use which instance we are referring to.

We can then give the formal definition of the operational semantics of control-flow automata. This semantics takes into account the execution of a CFA transition and executions of calls.

Definition 7.2.4. *The operational semantics of a control-flow automaton is a transition system over configurations of type* $S^{\uparrow\downarrow} \times \Omega$*, and transition labels of the form* $(\Sigma \times \Omega)^*$ *is characterized as the smallest relation obeying the following rules with* $\Rightarrow$ *as its flat transitive closure:*

(i) *A configuration* $(s_1^{\uparrow}, \omega)$ *transitions to a configuration* $(s_2^{\downarrow}, st(\omega))$ *if there is a transition between* s_1 *and* s_2 *with a condition that holds on* ω*, and tagged with the statement* st*. The transition is tagged by both the event triggered and the program state after the statement execution:*

$$\frac{s_1 \xrightarrow{c \triangleright st \blacktriangleright e} s_2 \qquad s_1 \notin E \qquad c(\omega)}{(s_1^{\uparrow}, \omega) \xrightarrow{\langle (e, st(\omega)) \rangle} (s_2^{\downarrow}, st(\omega))}$$

(ii) *If* s *is not a call state then configuration* $(s^{\downarrow}, \omega)$ *transitions to a configuration* $(s^{\uparrow}, \omega)$*:*

$$\frac{s \notin dom(calls)}{(s^{\downarrow}, \omega) \xrightarrow{\langle\rangle} (s^{\uparrow}, \omega)}$$

(iii) *If* s *is a call state then configuration* $(s^{\downarrow}, \omega)$ *transitions to a configuration* $(s^{\uparrow}, \omega')$ *only if the execution of the function called by* s *ends in an end state with the* ω' *variable state:*

$$\frac{s \in dom(calls) \qquad M = calls(s)(\omega) \qquad \exists\, s_E \in E_M, \omega' \in \Omega \cdot (s_{0_M}, \omega) \overset{ews}{\Longrightarrow} (s_E, \omega')}{(s^{\downarrow}, \omega) \xrightarrow{ews} (s^{\uparrow}, \omega')}$$

The traces of a program P *are the traces induced by some variable state from the initial state to an end state of* P*:* $traces(P) \overset{\text{def}}{=} \{ews \mid \exists\, s_E \in E_P, \omega_0, \omega \in \Omega \cdot (s_0^{\downarrow}, \omega_0) \overset{ews}{\Longrightarrow} (s_E^{\uparrow}, \omega)\}$.

Note throughout then we will be assuming that conditions have no side-effects, which is not generally a condition required out of conditional branching in real-world programming languages. Any side-effect however can be encoded in the statement part of the transition label.

We wish to determine whether the traces produced by a CFA satisfy a DEA, however CFAs include silent events, which are not considered by DEAs. Then we define a projection operator from traces of $(\Sigma_\epsilon \times \Omega)$ to $(\Sigma \times \Omega)$.

Definition 7.2.5. *The projection of a program trace of an alphabet $\Sigma_\epsilon \times \Omega$ onto the alphabet $\Sigma \times \Omega$ removes any pairs in the trace with the ϵ event:*

$$\begin{aligned} \langle\rangle \restriction_\Sigma &\stackrel{\text{def}}{=} \langle\rangle \\ (\langle(\epsilon, \omega)\rangle : ews) \restriction_\Sigma &\stackrel{\text{def}}{=} ews \restriction_\Sigma \\ (\langle(e, \omega)\rangle : ews) \restriction_\Sigma &\stackrel{\text{def}}{=} \langle(e, \omega)\rangle : (ews \restriction_\Sigma) \end{aligned}$$

We overload this for sets of traces $ewss \restriction_\Sigma \stackrel{\text{def}}{=} \{ews \restriction_\Sigma \mid ews \in ewss\}$.

We can then finally talk about programs and monitors together by defining what it means for a CFA to satisfy a DEA, in terms of the CFA not exhibiting violating traces of the DEA.

Definition 7.2.6. *A CFA P is said to satisfy a DEA π iff it cannot generate a violating trace of π: $P \vdash \pi \stackrel{\text{def}}{=} traces(P) \restriction_\Sigma \cap V(\pi) = \varnothing$.*

Note that we may use program traces on the DEA transitive closure, e.g. $(q_0, \theta_0) \stackrel{ews}{\Longrightarrow} (q, \theta)$. This should be take as equivalent to $(q_0, \theta_0) \stackrel{ews\restriction_\Sigma}{\Longrightarrow} (q, \theta)$.

As part of our contributions we will be analysing CFAs to identify events that can be silenced (i.e. replaced with the silent action ϵ), thus reducing the amount of times a monitor is called at runtime. The instrumentation transformations we will consider are reductions, in that they simply silence some events, unlike other approaches that also summarise sequences of programs [Dwyer and Purandare, 2007].

Definition 7.2.7. *A program P' is said to be an instrumentation reduction of a program P, written $P' \sqsubseteq P$, if P' has the same structure as P while possibly some events are replaced with the ϵ event, as captured by the following rules:*

(i) *A transition is in P iff it is in P' or it is replicated in P' with the silent event:* $s \xrightarrow{c \rhd st \blacktriangleright e}_P s' \Leftrightarrow (s \xrightarrow{c \rhd st \blacktriangleright e}_{P'} s' \vee s \xrightarrow{c \rhd st \blacktriangleright \epsilon}_{P'} s')$;

(ii) *Every call in P is replicated in P′ with an instrumentation reduced version of it:* $\forall\, s \in dom(calls_P) \cdot s \in dom(calls_{P'}) \wedge \forall\, \omega \in dom(calls_P((s)) \cdot calls_{P'}(s)(\omega) \sqsubseteq calls_P(s)(\omega)$.

This finishes the presentation of the formal objects upon which the presented residual analyses will be based on.

In Chapter 3 we abstractly identified properties as program acceptors. Here instead we have a concrete notion of properties and programs in terms of automata. In the next section we discuss the appropriateness of the residual condition in this context, and motivate the need for it to be strengthened in this context to give stronger verification guarantees.

7.3 Correctness of Reductions

In Chapter 3 we defined a method to combine verification methods using the notion of a residual, where given knowledge that a program satisfies a property π, and another property π_1 we wish to prove of the program, then we can construct another property $\pi_2 \in \pi \div \pi_1$. This π_2 will accept all programs that π_1 accepts, while it ideally is smaller and easier to prove at runtime than π_1.

In our case we will not be explicitly constructing a model of a CFA (π) as a DEA, instead we will leave π_1 implicit and simply analyse the CFA against π, producing the residual π_2[2]. In this section then we consider when a reduced property is a residual of the property to prove and the program CFA. Moreover, in Chapter 3 we did not have a concrete notion of a program, to keep the theory general, however having presented the class of CFAs we can now also discuss how to un-instrument programs (through some transformation) in a manner that does not effect the verdict given at runtime. In this section we then give different equivalence conditions at different levels of strictness, and consider which is appropriate for our context.

At a high-level we want a property reduction to judge the program under verification in the same way as the original property. Definition. 7.1.5 gives us this, by chracterising property equivalence in terms of when properties have the same violating traces. However, when we are only interested in a single program, we can weaken this condition, by parametrising this equivalence relation with respect to the program.

Definition 7.3.1. *Two properties π and π' are said to be P-*equivalent *if P satisfies π only if P satisfies π': $\pi \equiv_P \pi' \stackrel{\text{def}}{=} P \vdash \pi \Leftrightarrow P \vdash \pi'$.*

[2] π_1 is in effect the CFA and the level of abstraction we are analysing it at.

An instrumentation reduction should obey a dual condition, in that reducing instrumentation does not change the way the new program is judged by the property, ensuring a correct verdict.

Definition 7.3.2. *Two programs P and P' are said to be* π-equivalent *if P satisfies π only if P' satisfies π: $P \equiv_\pi P' \stackrel{\text{def}}{=} P \vdash \pi \Leftrightarrow P' \vdash \pi$.*

At a high-level these conditions work — they ensure verdicts are preserved at the level of programs. However these conditions only ensure the same verdicts for a whole program. Consider a program with two branches that both violate the property. An instrumentation reduction can be defined that silences one branch, leaving a reduced program that is still violating, but that is not violating in the same way. Similarly, if the two branches in the program violate the property in different ways, then the property can be reduced in such a way as to detect only one type of violation, where now the program is still not compliant with the property but not in the same way. These issues affect runtime verification, where without further conditions such a reduced property may give a different verdict from the original property on the same trace.

Then we define a stricter equivalence condition for both properties and instrumentation that checks that compliance is maintained in *lockstep* by the properties, i.e. at every time-step the same verdict is maintained. To allow us to define this we will be using the notation $pre_i(ews)$ to denote the prefix of length i of the program trace ews.

Definition 7.3.3. *The trace of length i of an execution of an program P given a starting variable state ω is either: (i) the prefix of length i of the respective finite execution; or (ii) the empty trace if there is no prefix of such length:*

$$trace_i(P, \omega) \stackrel{\text{def}}{=} \begin{cases} pre_i(ews) & \exists\, \omega' \in \Omega, s_{\in} E, \cdot (s_0^{\downarrow}, \omega) \stackrel{ews}{\Longrightarrow} (s_E^{\uparrow}, \omega') \\ \langle\rangle & otherwise \end{cases}$$

Using this we can then define when two programs are equivalent at the level of program state, by considering the traces associated with such a state.

Definition 7.3.4. *Two programs P and P' are said to be* π-lockstep-equivalent *if for any initial variable state ω and natural number i then the trace of length i with respect to ω of one program is violating iff the trace of the same length of the other program is also violating: $P \stackrel{\pi}{=} P' \stackrel{\text{def}}{=} \forall\, \omega \in \Omega, i \in \mathbb{N} \cdot trace_i(P, \omega)\lfloor_\Sigma \in V(\pi) \Leftrightarrow trace_i(P', \omega)\lfloor_\Sigma \in V(\pi)$.*

Note how this equivalence condition is stricter than the previous.

Theorem 7.3.1. $P \stackrel{\pi}{=} P' \Rightarrow P \equiv_\pi P'$

Proof

Consider that $P \not\models \pi$, then there is a trace of P that is violating. This means it has a violating prefix, which by Definition. 7.3.4 means P' has a trace with a violating prefix, and then $P' \not\models \pi$. Then we can conclude $P \not\models \pi \Rightarrow P' \not\models \pi$. The same argument can be used to conclude that $P' \not\models \pi \Rightarrow P \not\models \pi$, and then our result follows by taking the contrapositive of both of these.

We can define a similar relation for properties, and prove a similar result.

Definition 7.3.5. *Two properties π and π' are said to be* P-lockstep-equivalent *if for any starting variable state, the execution prefix of a program is violating in π iff it is also violating in π':* $\pi \overset{P}{=} \pi' \overset{\text{def}}{=} \forall\, \omega \in \Omega, i \in \mathbb{N} \cdot trace_i(P, \omega)\lfloor_\Sigma \in V(\pi) \Leftrightarrow trace_i(P, \omega)\lfloor_\Sigma \in V(\pi')$.

This condition is also stricter than the first condition for property equivalence.

Theorem 7.3.2. $\pi \overset{P}{=} \pi' \Rightarrow \pi \equiv_P \pi'$

Proof

Consider that any π-violating trace of P has a violating prefix, and then it must also be π'-violating by Definition. 7.3.5. The argument applies in the other direction.

We will be using the latter stricter correctness conditions for our reductions of instrumentation and properties, since they ensure instrumentation is not transformed in a way that affects compliance at runtime, which is a desired property for us since we want to prove the residual properties at runtime.

7.4 Conclusions

In this section we have presented a view of properties as DEAs, and programs CFAs, and defined structural reductions of DEAs and instrumentation reductions of CFAs. We have also explored two correctness conditions for these reductions, settling on a condition that allows for equivalent verdicts on every program trace

at runtime. In the next section we presenting our main contribution: a residual analysis of DEAs as properties and programs as CFAs.

8

Residual Analysis

In this chapter we describe an analysis to produce residuals of both CFA instrumentation and DEAs. Our work was inspired by that of CLARA [Bodden et al., 2010] for typestate properties as finite-state automata, although we take a different approach in that we focus primarily on producing a reduced DEA but also in the process remove some event instrumentation. We abstract away from typestate here, for a simpler and cleaner presentation.

The motivation behind the techniques we present is largely to reduce the overheads of monitoring by reducing both the points of instrumentation and the computation that has to be done by the monitor upon being triggered. We also desired such an analysis to be carried out during the instrumentation process of monitoring, allowing for smarter instrumentation. This requirement places constraints on the type of analysis possible — time- and memory-intensive analyses, while useful, would place undue overheads on the instrumentation process. Thus our motivation was towards automatable and light static analyses that can be integrated in an almost seamless manner with the process of instrumentation.

To this end the analysis we consider is primarily intended to be *intraprocedural*, in that it acts on each method of the program while ignoring the rest of the program. While this will require us to over-approximate the behaviour possibly occurring outside a method, this abstraction can be discarded after analysis on the method finishes. An *interprocedural* analysis on the other hand would require more memory at any given point in time. However, our residual analysis can also be applied interprocedurally, but we leave an implementation of this and comparison of it with an intraprocedural analysis in terms of computational and memory expense for future work.

The analysis we define in effect analyses a method by creating an abstraction of all the possible program traces that pass through that method. By taking into consideration these abstractions in synchronicity with the DEA we can acquire a sound over-approximation of the monitored system at runtime. From each such system we collect both the transitions of the DEA used locally in the method (i.e. not at chaotic states) and the useful instrumentation in the method (instrumentation that can affect a verdict). Then, if the analysis is performed for all the methods, a new residual DEA can be created by considering the union of locally used DEA transitions, while instrumentation can be reduced to the collected useful instrumentation points. This can be iterated until a fix-point is reached. To optimise this approach we also consider the addition of certain interprocedural information (e.g. knowledge that a certain event cannot occur before a method). We further illustrate how an SMT solver can be used to prune unviable transitions from the abstract monitored system and to identify when DEA guards must always resolve to *true* at runtime.

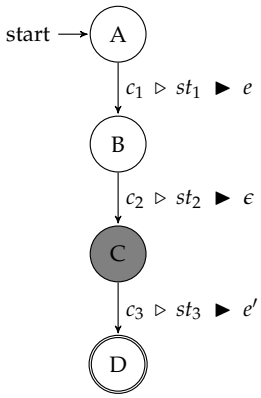
In Section 8.1 we illustrate how we create abstractions from CFAs, inspired by the work of Bodden et al. [2010], while in Section 8.2 we consider several ways property residuals and instrumentation reductions can be created from these abstractions. We conclude the presentation of these contributions in Section 8.3.

Claims made in this chapter with longer proofs are proven in Section **??**, while otherwise shorter proofs or proof sketches are presented here.

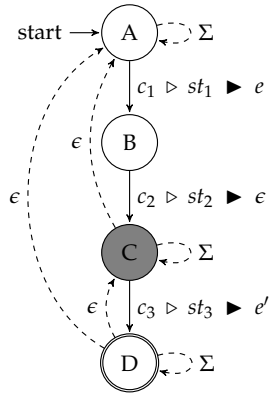
8.1 Intraprocedural Abstractions

In this section we describe formally an approach towards creating program abstractions by individually considering methods of the program. This is based on work by Bodden [2009], who motivates the use of such an analysis in the context of typestate analysis since it can be enough for a large number of objects that are local to a single method. The CLARA variant of this abstraction is described using a worklist algorithm by Bodden [2009], here we describe it in a formal manner with respect to CFAs.

Consider the simple CFA specified in Figure. 8.1(a). This CFA can represent a method in a program, possibly called at some point during the execution of the program. Then, before Figure. 8.1(a) is executed some events may be triggered (i.e. before state A). Similarly some events may be triggered after the method finishes executing (i.e. after state D), and during the execution of call sites of the method (i.e. during the call at state C). This outside behaviour can simply be over-approximated by making the respective states chaotic, i.e. by adding a looping transition around each such state for each event from Σ, as illustrated in



(a) Method CFA example, with state 3 being a call site.



(b) Method CFA with chaotic outside behaviour, recursive call site 3, and that can be re-entered.

Figure 8.1: Example CFA with abstracted version.

Figure. 8.1(b) (the dashed transitions represent the transitions added to model possible outside behaviour). Moreover, a method may be re-entered more than once during a single execution or it can be re-entered recursively from a call site. This is modeled respectively through transitions from the end state to the start state, and a transition from a possibly recursive call site to the initial state and another transition from an end state to the call site, simulating relinquish of control back to the called CFA. This abstraction is very coarse, but does not require any possibly intensive analysis of the program, satisfying our requirement for a lightweight analysis.

In Section 8.1.1 we describe formally how this abstraction is created through an extension of CFAs with an abstract transition function, and relate it to the behaviour of the program at runtime.

8.1.1 A Control-flow Abstraction

To abstract a CFA M we will be augmenting M with an abstract transition function $\dashrightarrow \in S \times \Sigma_\epsilon \times S$ that will be used to model maximally the possible events occurring outside M as part of an execution of another method.

Definition 8.1.1 (Abstracted CFA)**.** *An* abstracted control-flow automata *is a CFA augmented with an abstract transition function $\dashrightarrow \in S \times \Sigma \to S$. We denote these automata by the class $ACFA \stackrel{\text{def}}{=} CFA \times (S \times \Sigma_\epsilon \to S)$. The transitive closure of these automata, $\Rightarrow$, also includes the abstract transition function. We refer to the original transition function of the CFA as the concrete or local transition function.*

As illustrated in Figure. 8.1(b), such an abstraction is constructed to allow the behaviour external to M to be chaotic, while taking into account recursive calls into M from call sites, and multiple calls to M in the same execution. However, after an iteration of an analysis we may be able to fine-tune this abstraction. For example, we may be able to conclude that event e only occurs in method M, and thus an abstraction of M should not include this event e on abstractions modeling outside behaviour. Then we define the abstraction in a parametric manner by assuming functions $before, after, during \in S \to 2^{\Sigma}$ that respectively answer the question of what events can occur before, after, and during a method execution. Moreover we assume a function $recursive \in S \to Bool$ that indicates whether a call state can call back into its method, and another function $calledMultipleTimes \in CFA \times CFA \to Bool$ that indicates when a program P can call a method M more that once in the same execution. Initially we assume maximally sound values for these parameters, i.e. *before*, *after*, and *during* respectively associate initial, end, and call states with Σ, while any call state is assumed to be recursive, and a method to be called multiple times.

We define formally the parametrised abstraction constructed along these lines.

Definition 8.1.2. *The* intraprocedural abstraction *of a method M in the context of a program P, denoted by $A(P)$, parameterised by the functions $before, after, during \in S \to 2^{\Sigma}$, $recursive \in S \to Bool$ and $calledMultipleTimes \in CFA \times CFA \to Bool$, is the abstracted control-flow automaton with its transition functions defined as the smallest transition function obeying the following rules:*

1. *Any transition in the original automaton is replicated in the abstract model:*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s'}{s \xrightarrow{c \triangleright st \blacktriangleright e} s'}$$

2. *A looping transition is introduced on each call state for each event that can occur during an execution of a call:*

$$\frac{s \in dom(calls) \qquad e \in during(s)}{s \overset{e}{\dashrightarrow} s}$$

3. *A looping transition is introduced on the initial state for each event in the alphabet before the initial state:*

$$\frac{e \in before(s_0)}{s_0 \overset{e}{\dashrightarrow} s_0}$$

4. *A looping transition is introduced on each final state for each event in the alphabet after the final state:*

$$\frac{e \in after(s_E)}{s_E \xrightarrow{e} s_E} \; s_E \in E_M$$

5. *A transition from a call state to the initial state and from each end state to the initial state, tagged with the silent event, is introduced for each call state that is recursive:*

$$\frac{s \in dom(calls) \qquad recursive(s)}{s \xrightarrow{\epsilon} s_0 \qquad s_E \xrightarrow{\epsilon} s} \; s_E \in E_M$$

6. *If a method M can be called again after being called once in a CFA P then any final state is allowed to transition back into the initial state:*

$$\frac{calledMultipleTimes(P, M)}{s_E \xrightarrow{\epsilon} s_0} \; s_E \in E_M$$

We write $s \stackrel{t}{\Rightarrow} s'$ $(t \in \Sigma^*)$ *as the smallest relation such that: (i) if* $t = \langle\rangle$ *then* $s = s'$ *or* $\exists\, s'' \cdot s \xrightarrow{\epsilon} s'' \wedge s'' \stackrel{\langle\rangle}{\Rightarrow} s'$*; (ii) if* $t = \langle e \rangle$ *then* $\exists\, s'', s''' \cdot s \stackrel{\langle\rangle}{\Rightarrow} s'' \wedge s'' \xrightarrow{c \,\triangleright\, st \,\blacktriangleright\, e} s''' \wedge s''' \stackrel{\langle\rangle}{\Rightarrow} s'$*; or if* $t = t' +\!\!+ t''$ *then* $\exists\, s'' \cdot s \stackrel{t'}{\Rightarrow} s'' \wedge s'' \stackrel{t''}{\Rightarrow} s'$.

From such an abstraction we can extract a set of event traces by simply using the transitive closure of both transitions of the automaton.

Definition 8.1.3. *The* traces *of an abstracted CFA AP are the traces induced by the transitions of the automaton from the initial state to an end state:* $atraces(AP) \stackrel{\text{def}}{=} \{es \mid \exists\, s_E \in E_{AP} \cdot s_0 \stackrel{es}{\Rightarrow} s_E\}$.

In the next section we consider the manner in which these abstractions can be considered to be over-approximations of the program. In effect we will conclude that they are not necessarily over-approximations of the whole program, but only of the executions that include calls to the method being used a base for the abstraction.

8.1.1.1 Relation to Program

We will be showing that the intraprocedural abstraction of each method, when considered in conjunction generate a set of traces that abstracts the traces of

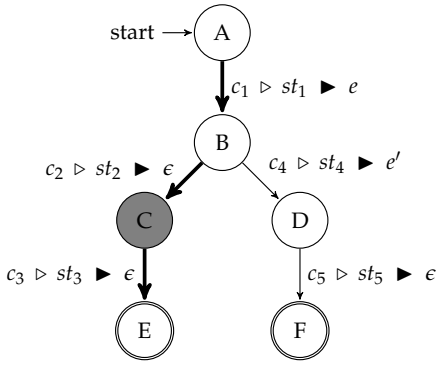


Figure 8.2: Example program with a call state (C) and with different branches.

the original program. To aid us with the statement of these claims we define a function that abstracts symbolic event traces into event traces by simply removing the symbolic states.

Definition 8.1.4. *The* event abstraction *of a program trace is the function that discards the symbolic states in traces, leaving an event trace:*

$$\begin{array}{lll} \alpha(\langle\rangle) & \stackrel{\text{def}}{=} & \langle\rangle \\ \alpha((e,\omega) : ews) & \stackrel{\text{def}}{=} & \langle e\rangle +\!\!+ \alpha(ews) \end{array}$$

We can show that $A(M)$ over-approximates the traces of M, since it contains all the concrete transitions of M and abstracts any call states of M. We can show that any transition in the operational semantics of CFAs has a direct abstract counterpart in this abstraction, which we then can use to prove M abstracts $A(M)$.

Theorem 8.1.1. *Any execution of method M, is reflected in its abstraction $A(M)$:* $\forall\, \omega_1, \omega_2 \in \Omega, s \in s_M \cdot (s^{\downarrow}_{0_M}, \omega_1) \stackrel{ews}{\Longrightarrow} (s^b, \omega_2) \;\Rightarrow\; s_0 \stackrel{\alpha(ews)}{\Longrightarrow} s$

Proof We can show the result for the general one-step case $\forall\, \omega_1, \omega_2 \in \Omega, s_1, s_2 \in s_M \cdot ((s^a_1, \omega_1) \stackrel{ews}{\longrightarrow} (s^b_2, \omega)) \;\Rightarrow\; (s_1 \stackrel{\alpha(ews)}{\Longrightarrow} s)$, by considering cases on equality between s_1 and s_2:

Case 1: $s_1 = s_2$, and then, by the operational semantics of CFAs, either: (i) s_1 is not a call site (from which the result easily follows); or (i) s_1 is a call site. In the abstraction $A(M)$ there are looping transitions around s_1 for each event that could possibly be present in $\alpha(ews)$, by the second rule of Definition. 8.1.2. Then $s_1 \stackrel{\alpha(ews)}{\Longrightarrow} s$ holds in the abstraction $A(M)$.

Case 2: $s_1 \neq s_2$. There are two cases here then, by the operational semantics of CFAs: (i) either $ews = \langle\rangle$ (i.e. no event is activated), and there is an ϵ-transition between s_1 and s_2,from which the result easily follows by the first rule of Definition. 8.1.2; or (ii) $ews = \langle e, \omega\rangle$ and there is a transition $s_1 \xrightarrow{c \triangleright st \blacktriangleright e} s_2$, from which the result also easily follows by the first rule of Definition. 8.1.2. □

This result projects easily onto the transitive closure, giving us the required result.

If M is not the program then this theorem is not very helpful. We can make this stronger by considering that the abstraction of a method M is not necessarily an abstraction of the whole program, but rather an abstraction of the traces of the program that pass through M. For example, consider Figure 8.2, which has a call state to some method M at state labelled by C. The abstraction of M will produce an over-approximation only of the paths of the program through state C, illustrated by the thicker edge, while the other branch remains unmodeled. Here then we want identify these exactly these program traces that M over-approximates.

Given an execution of a program, we can identify when this passes through a certain method M by considering the operational semantics and looking at whether the execution passes through a state of M. We formally characterise this.

Definition 8.1.5. *A program trace ews passes through a method M in a program P if either:*

(i) *M is P and ews is a trace of P:* $via_P(ews, P) \stackrel{\text{def}}{=} \exists\, s_E \in E_M, \omega_0, \omega \in \Omega \cdot (s_0^{\downarrow}, \omega_0) \stackrel{ews}{\Longrightarrow} (s_E^{\uparrow}, \omega)$*; or*

(ii) *ews is a trace of P and can be re-written as* $ews' +\!\!+ ews'' +\!\!+ ews'''$ *such that* ews'' *corresponds to a trace induced by a call in P, and* ews'' *passes through M in the called method:* $via_P(ews' +\!\!+ ews'' +\!\!+ ews''', M) \stackrel{\text{def}}{=} \exists\, s \in S_P, s_E \in E_P, \omega_0, \omega, \omega', \omega'' \in \Omega \cdot (s_0^{\downarrow}, \omega_0) \stackrel{ews'}{\Longrightarrow} (s^{\downarrow}, \omega) \xrightarrow{ews''} (s^{\uparrow}, \omega') \stackrel{ews'''}{\Longrightarrow} (s_E^{\uparrow}, \omega'') \wedge via_{calls(s)(\omega)}(ews'', M)$.

We define the set of traces via a method M as those traces that pass through M: $tracesVia(P, M) \stackrel{\text{def}}{=} \{ews \mid via_P(ews, M)\}$.

The abstraction of a method will then over-approximate exactly those traces that pass through it. We can show this by considering that any trace that passes through M can be re-written as trace of M with a certain prefix and suffix, which are modeled by the chaos at the initial and end states in the abstraction, and similarly for any call sites.

Theorem 8.1.2. *With sound abstraction parameter functions, then the set of event-abstractions of the traces through a method M are approximated by the set of event traces induced by the abstraction of M:* $\alpha(tracesVia(P,M)) \subseteq atraces(A(M))$.

Proof Consider that any trace in $tracesVia(P,M)$ can be re-written as a trace $st_1 +\!\!+ st_2 +\!\!+ st_3$ by Definition. 8.1.5, where st_2 represents one execution of M. st_1 and st_3 are modeled by the chaos at the initial and end states of $A(M)$, reducing the problem to showing that st_2 is modeled by $A(M)$, which is ensured by Theorem. 8.1.1.

Then, a single abstraction is not an abstraction of the whole program, however we want to be able to make conclusions about the whole program, not just single methods. We then require an abstraction of the whole traces of a program. A simple such abstraction is the abstraction of the main method P. In general, we can create such an abstraction by considering *covers* of P, i.e. a cover is a set of methods for which all executions of P pass through at least one method in the cover.

Definition 8.1.6. *The set of covers of P includes a subset of the methods of P such that if for all executions of P it contains a method that the execution passes through:* $coversOf(P) \stackrel{\text{def}}{=} \{Ms \subseteq methodsOf(P) \mid \forall t \in traces(P) \cdot \exists M \in Ms, \omega \in \Omega \cdot t \in tracesVia(P,M)\}$.

Then, it easily follows from Theorem. 8.1.2 that a cover of P induces an event trace set that covers the abstract event traces of P.

Corollary 8.1.1. *With maximally sound abstraction parameter functions, and a cover* $Ms \in covers(P)$ *then:* $\alpha(traces(P)) \subseteq \bigcup_{M \in Ms} atraces(A(M))$.

Proof Consider that each trace in P passes through at least one method in the cover Ms. Then, since from $t \in traces(P)$ we can conclude that $\alpha(t) \in \alpha(traces(P))$ (by Definition. 8.1.4), and by Theorem. 8.1.2 and Definition. 8.1.6 we can conclude that $\exists M \in Ms \cdot \alpha(t) \in atraces(A(M))$.

Such covers could be constructed through a depth-first search of the program automaton (exploring also the possibly called automata), which is known to be linear in the size of the graph. In this search we would explore every symbolic path from the initial state of the program and end state of the program and note the set of methods called by states in that path. Then a cover of the program can be created by choosing a method from each such set.

Here we have considered a purely control-flow abstraction of a CFA, however a CFA, like a program, also encodes some variable state. In the next section we

Listing 8.1: Example of compliant use of `Iterator`.

```
1  public static Integer safeNext(Iterator<Integer> it){
2      int val;
3      if(it.hasNext()){
4          val = it.next();
5      } else{
6          val = -1;
7      }
8
9      return val;
10 }
```

consider how variable state abstractions can be encoded in our framework, and define a simple but useful abstraction.

8.1.2 A Variable State Abstraction through Propagation

Consider Listing. 8.1, that returns the next value in an iterator if present, and -1 otherwise. Figure 8.3 represents a CFA that represents the behaviour of Listing. 8.1. If we want to be able to verify a property like *`it.next()` cannot be called unless `it.hasNext()` is true* (as illustrated in Figure. 7.2(b)) here we must be able to determine that at state B in Figure 8.3 `it.hasNext()` holds true. This can be easily concluded by analysing the condition on the transition into B, and the knowledge that calling `it.hasNext()` has no side-effects (and thus it continues holding). Then, to be able to do this automatically we will describe a simple condition propagation algorithm that associates states with conditions.

Conditions cannot be propagated forward in an unconstrained manner, since actions may be performed to change the symbolic state in such a way that a condition no longer holds. For example in Figure 8.3 executing the statement `it.next()` possibly affects the value returned by *it.hasNext()*, i.e. if it is called again after state B it may return false. Here we assume that we have an operator $\otimes$ that soundly relates a condition and a statement if execution of the statement does not affect the condition, or if the statement *preserves* the condition. We characterise this in terms of program symbolic states, while we also overload it for methods and calls to deal with outside behaviour that may affect a condition.

Definition 8.1.7. *A condition c is said to be* preserved *by a statement st if when the condition holds on a program symbolic state then the state produced by applying the statement still respects the condition: $c \otimes st \Rightarrow (\forall\, \omega \in \Omega \cdot c(\omega) \Rightarrow c(st(\omega)))$.*

We overload this for methods by considering the set of used by a method's transitions

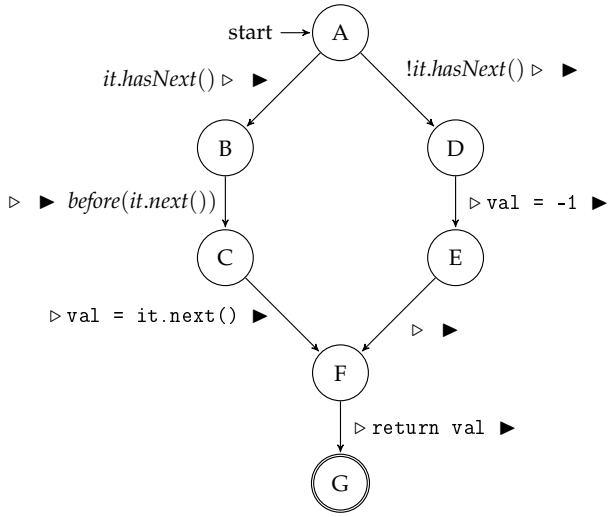


Figure 8.3: CFA representing Listing. 8.1.

($stmts(M) \stackrel{\text{def}}{=} \{st \mid \exists s, s', c, st, e \cdot s \xrightarrow{c \triangleright st \blacktriangleright e} s'\}$), and those on the methods it possibly calls[1]*:* $c \otimes M \Rightarrow (\forall M' \in methodsOf(M), st \in stmts(M') \cdot c \otimes st)$.

We overload this also for calls: $c \otimes call \Rightarrow (\forall \omega \in \Omega \cdot c \otimes call(\omega))$.

The preservation relation defined here abstractly could for example involve looking at whether a statement modifies variables used in the condition.

More sophisticated analyses consider how a statement affects a condition, e.g. given a condition `x < y` and an assignment `x = x + 1` the condition could be transformed into `x` $\leq$ `y`. This can however complicate things, especially with regards to loops where loop invariants have to be identified for a sound and finite analysis [Furia et al., 2012]. By only propagating conditions (which act as *state invariants* here) until a statement that affects them is executed we avoid this problem, at the price of weaker state invariants.

We will thus not be encoding the precise effects of statements, however instead we assume a simple analysis that extracts conditions that always hold true after the execution of a statement. A simple example is that a statement `x = 7` implies the condition `x == 7`. We can characterise such an analysis in terms of a relation by considering the effects a statement has on a symbolic state.

Definition 8.1.8. *A condition c is said to be a* post-condition *of a statement st if it always holds true after st is applied:* $\forall \omega \in \Omega \cdot c(st(\omega))$. *We assume an analysis that produces a subset of conditions related with a statement in this way:* $post(st) \subseteq \{c \mid \forall \omega \in \Omega \cdot c(st(\omega))\}$.

[1] Recall this includes itself.

This kind of analysis is common in verification, e.g. for the purposes of symbolic execution Baldoni et al. [2018].

Such an analysis can easily and cheaply be created by analysing the syntactic structure of a statement, and treating any operations that possibly require more intensive analysis (e.g. setting the value of a variable to the result of a function) as simply implying the `true` condition. Consider a statement block `x = 7; x = x + 1;`. We will be able to conclude that `x == 7` after the first statement, however we cannot give a condition that is a post-condition of the second statement, given the recursive reference. On the other hand, if the program is in single assignment form, i.e. `x0 = 7; x1 = x0 + 1;` we would be able to have a more precise analysis. Here however we remain agnostic of the program form. Then we shall be annotating each state in a CFA abstraction by condition sets, which are updated given the conditions and statements with which transitions are annotated. We define a function to update a condition set based on these.

Definition 8.1.9. *A condition set cs is updated given a pair of a condition c and statement st by adding c to the set and removing from it any condition that is not preserved by st, and adding any post-condition of st:* $update(cs, (c, st)) \stackrel{\text{def}}{=} \{c' \in cs \cup \{c\} \mid c' \otimes st\} \cup post(st)$.

We can then use these relations to propagate conditions in a CFA abstraction until a statement is met that affects the condition, while removing conditions at states modeling behaviour external to the method being abstracted. A variable abstraction will be represented using a relation $\rightsquigarrow \subseteq S \times 2^{Cond}$, and will be used to associate states with condition sets. Then, given a configuration of a state at runtime we will be able to show how at least one of the associated condition sets will have all its conditions hold true at runtime.

Definition 8.1.10. *The* simple propagation symbolic state abstraction *of an abstract CFA M, used by a program CFA P, is the pair of smallest relations* $\rightsquigarrow \subseteq S \times 2^{Cond}$ *obeying the following rules:*

1. *The initial state is associated with the empty set of conditions.*

$$\frac{}{s_0 \rightsquigarrow \varnothing}$$

2. *For a concrete transition from a state s to a non-call state* s'*, with condition c and statement st, where s is related to cs, then* s' *is related to the update of cs with* (c, st)*.*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s' \qquad s \rightsquigarrow cs \qquad s' \notin dom(calls) \qquad s' \notin E_M}{s' \rightsquigarrow update(cs, (c, st))}$$

3. *For a concrete transition from a state s to a call state s', with condition c and statement st, where s is related to cs, then s' is related to the update of cs with (c, st), but without any conditions not preserved by the calls of s'.*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s' \qquad s \rightsquigarrow cs \qquad s' \in dom(calls)}{s' \rightsquigarrow \{c \in update(cs, (c, st)) \mid \forall\, call \in calls(s) \cdot c \otimes call\}}$$

4. *For a concrete transition from a state s to an end state s', with condition c and statement st, where s is related to cs, then s' is related to the update of cs with (c, st), but without any conditions not preserved by other methods of the program*[2]*.*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s' \qquad s \rightsquigarrow cs \qquad s' \in E_M}{s' \rightsquigarrow \{c \in update(cs, (c, st)) \mid \forall M' \in methodsOf(P), M' \neq M \wedge c \otimes M'\}}$$

Note that an optimisation here would be to reduce the symbolic state abstraction by removing a condition set associated with a state s if there is another condition set associated with s that is stricter (i.e. that implies it).

Then, we have described a method to associate CFA states with sets of conditions, where given any execution prefix to such a state at runtime then at least one of the invariant sets will hold true at runtime (e.g. if a state s is associated with sets $\{c_1, c_2\}$ and $\{c_3, c_4\}$ then the condition $(c_1 \wedge c_2) \vee (c_3 \wedge c_4)$ is always true for configurations of s during an execution).

Theorem 8.1.3. $\forall\, \omega \in \Omega \cdot (s_0^{\downarrow}, \omega_0) \Rightarrow (s^{-}, \omega) \;\Rightarrow\; \exists\, s \rightsquigarrow cs \cdot \forall\, c \in cs \cdot c(\omega)$

Proof We prove this by induction on the structure of the big-step transition.

For base case the result easily follows by the first rule, since the empty condition set trivially satisfies the right-hand side of the required implication.

For the inductive hypothesis we can assume the required statement up to a certain transition sequence.

For the inductive case $(s_0^{\downarrow}, \omega) \Rightarrow (s^{x}, \omega) \rightarrow (s_1^{y}, \omega')$ we can take cases on the values of $x, y \in \{\uparrow, \downarrow\}$.

Case 1 : $(x, y) = (\uparrow, \downarrow)$

This implies there is a transition between s and s_1, by the semantics of CFAs: $s \xrightarrow{c \triangleright st \blacktriangleright e} s_1$. By the inductive hypothesis we can also conclude there is some cs_s where $s \rightsquigarrow cs_s$ and $cs_s(\omega)$.

Consider that s_1 is not an end state.

We then consider two further cases, i.e. whether s_1 is a call state or not.

[2]Here we can make this more precise by adding some interprocedural analysis to consider only the statements possibly executed after a method.

Case 1.1 : $s_1 \notin dom(calls)$

In this case rule 2 applies.
By the semantics of CFAs then we can conclude that $\omega' = st(\omega')$, and that $c(\omega)$. The result easily follows by considering the definition of *update* in terms keeping only conditions not affected by st and adding post-conditions of st.

Case 1.2 : $s_1 \in dom(calls)$

In this case rule 3 applies.
The same argument as in the previous case applies here, since the condition set added by rule 3 is a subset of the condition set added by rule 2. Note that in this case a call has not been executed yet.

Case 2 : $(x, y) = (\downarrow, \uparrow)$

This case implies that $s = s_1$. There are two cases.

Case 2.1 : $s \notin dom(calls)$

In this case then by the semantics of CFAs $\omega = \omega'$ and then the result is exactly the inductive hypothesis.

Case 2.2 : $s \in dom(calls)$

In this case rule 3 is relevant.
Note how the condition set associated with call states by rule 3 removes any conditions possibly affected by th potential calls at runtime.

We are left to prove the case that s_1 is an end state. In that case rule 4 applies. The argument about *update* as used in Case 1.2 applies, since we are simply pruning the updated condition set. The point of this pruning is to remove conditions that do not apply after execution of the CFA, but that has no bearing on this theorem.

□

This is simply one variable abstraction that we present for both illustrative purposes and use for its simplicity and lack of computational expense. The basic framework behind this abstraction (i.e. a relation that associates states with sets of conditions, and an update function for this) is general enough to record the information of more powerful data-flow analyses.

In the next section we present a residual analysis based on the synchronous composition of a CFA abstraction with a DEA. We exploit variable abstractions to reduce this composition, by removing transitions that are not viable. An interesting aspect of a residual we present is that it is computed piece-wise over each method of the program.

8.2 Verification with Residuals

In this section we describe and motivate some residual analyses for properties as DEAs and programs as CFAs. The analysis we present will allow us to check for compliance of a CFA with a DEA and failing that produce a residual property.

We defined compliance of a CFA against a DEA in terms of the whether any traces of the CFA are violating traces of a DEA. For simplicity assume a flat CFA (without call states), then we can soundly check for this compliance by analysing the synchronous composition of the CFA and DEA. States in this composition are pairs of CFA and DEA states. Then, if we can determine compliance if no pair of states where the DEA state is a violating state is reachable in the composition. However CFAs in general are not flat.

We propose that instead of a CFA we can consider a CFA abstraction, and compose it in a similar way with the DEA. If we consider the CFA's main method in this way we can make conclusions about the program. If instead we consider a CFA of a method called by the main method we can only make conclusions about the executions that pass through that method. Here we can use the notion of covers, to collect conclusions made from each method to a conclusion about the program, based on the notions presented in Section 8.1.1.1.

The residual analyses we present will proceed in this manner, where we analyse the synchronous composition of an abstract CFA with a DEA, which we term an *abstract monitored system*. We choose to focus on each method, instead of a flattening of the main method CFA, trading a singular, very large but precise composition for multiple smaller but coarser compositions. Each abstract monitored system can be analysed to identify the CFA and DEA transitions that are useful to give a verdict, and then dually transitions that are irrelevant to give a verdict and be ignored. This will allow us to reduce both the property and the instrumentation.

Here we will first present abstract monitored systems in Section 8.2.1, showing how these intraprocedural systems can be analysed to conclude compliance of the whole program and how further analysis using variable abstractions can make the abstraction finer. We show how these systems can be analysed to identify instrumentation that can be turned off safely, and to create different property residuals in Section 8.2.2.

8.2.1 An Abstract Monitored System

Given a method M, and its abstraction $A(M)$ we will be constructing an over-approximation of its behaviour in synchronicity with a monitor of a property π, where we call this abstraction the abstract monitored system of M given π.

This system is driven by transitions in $A(M)$, and possibly matches them with DEA transitions. At this stage matching is only done based on events, and thus for soundness we must consider both the possibility that a DEA transition guard matches and the possibility it does not (we shall be using variable abstractions to prune any non-viable transitions, in Section 8.2.1.2). Each transition in the abstract monitored system will then be tagged by a pair of labels representing a concrete CFA transition and a DEA transition, possibly one of these labels can be the empty label ($\square$) denoting either an abstract CFA transition match, or no DEA transition match. For example $(s,q) \xrightarrow[e|g\mapsto a]{c \triangleright st \blacktriangleright e} (s',q')$ represents the execution of the CFA transition (top) which causes the monitor transition (bottom) to activate also, while $(s,q) \xrightarrow[\square]{c \triangleright st \blacktriangleright e} (s',q)$ represents no DEA transition match, $(s,q) \xrightarrow[e|g\mapsto a]{\square} (s',q')$ represents an activation of the monitor by an abstract transition in the CFA, and $(s,q) \xrightarrow[\square]{\square} (s',q')$ represents an abstract transition in the CFA without a monitor activation.

Definition 8.2.1. *The abstract monitored system of a DEA π and a symbolic control-flow automaton M, given a program abstraction function $A \in CFA \to ACFA$, written $M\|_A\pi$, is the automaton with states from $S_M \times Q_\pi$ and with transitions tagged by a pair of CFA and DEA transition labels or a symbol $\square$, $\to \subseteq (S_M \times Q_\pi) \times (\{\square\} \cup (Cond \times STMT \times \Sigma)) \times (\{\square\} \cup (\Sigma \times Guard \times Act)) \times (S_M \times Q_\pi)$, and obeying the following rules:*

1. *Given a configuration (s,q), where s and q have outgoing transitions with the same event, then they both synchronously evolve together:*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s' \qquad q \xrightarrow{e|g\mapsto a} q'}{(s,q) \xrightarrow[e|g\mapsto a]{c \triangleright st \blacktriangleright e} (s',q')}$$

2. *Given a configuration (s,q), then any transition from s is taken without considering a DEA transition*[3]*:*

$$\frac{s \xrightarrow{c \triangleright st \blacktriangleright e} s'}{(s,q) \xrightarrow[\square]{c \triangleright st \blacktriangleright e} (s',q)}$$

3. *Any abstract transition is taken without tagging the transition with any explicit labels:*

[3]This models three cases: (i) when there is no transition with the same event from q; (ii) when $e = \epsilon$; and (iii) when no property guard matches at runtime.

$$\frac{s \overset{e}{\dashrightarrow} s'}{(s,q) \overset{\square}{\underset{\square}{\Rightarrow}} (s',q)}$$

4. *Given a configuration (s,q), and s having an outgoing abstract transition with an event matching a transition from q, the configuration after taking these transitions is updated appropriately, with the transition only tagged by the DEA label:*

$$\frac{s \overset{e}{\dashrightarrow} s \qquad q \xrightarrow{e|g \mapsto a} q'}{(s,q) \xrightarrow[e|g \mapsto a]{\square} (s,q')}$$

We use $\Rightarrow$ for the transitive closure of $\rightarrow$. We use x to refer to CFA transition labels, and y to refer to DEA transition labels.

Throughout we will find useful a different transition function, one that given $(s,q) \overset{y}{\underset{x}{\rightarrow}} (s,q)$ ensures that (s,q) is reachable from the initial pair of states.

Definition 8.2.2. *We write $(s,q) \overset{y}{\underset{x}{\twoheadrightarrow}} (s',q')$ for $(s_0,q_0) \Rightarrow (s,q) \overset{y}{\underset{x}{\rightarrow}} (s,q)$, and use $\Longrightarrow$ for its transitive closure.*

We can then show that any execution step of M, and the corresponding step in the monitor at runtime is replicated and abstracted in the abstract system.

Theorem 8.2.1. *Any execution of method M, inducing a certain property state in a property π, is reflected in the abstract monitored system $M\|_A\pi$: $\forall\, \omega_1, \omega_2 \in \Omega, \theta_1, \theta_2 \in \Theta, s_1, s_2 \in s_M, q_1, q_2 \in Q_\pi \cdot (s_1^a, \omega_1) \overset{ews}{\Longrightarrow} (s_2^b, \omega_2) \wedge (q_1, \theta_1) \overset{ews}{\Longrightarrow} (q_2, \theta_2) \;\Rightarrow\; ((s_1, q_1) \Rightarrow (s_2, q_2))$.*

Proof First we show this in the case of a small-step:: $\forall\, \omega_1, \omega_2 \in \Omega, \theta_1, \theta_2 \in \Theta, s_1, s_2 \in s_M, q_1, q_2 \in Q_\pi \cdot (s_1^a, \omega_1) \xrightarrow{ews} (s_2^b, \omega_2) \wedge (q_1, \theta_1) \overset{ews}{\Longrightarrow} (q_2, \theta_2) \;\Rightarrow\; ((s_1, q_1) \Rightarrow (s_2, q_2))$

Recall that while proving Theorem. 8.1.1 we showed $(s_1^a, \omega_1) \xrightarrow{ews} (s_2^b, \omega_2)$ is reflected in the abstraction.

Case 1 : $s_1 = s_2$.

Then, by the CFA operational semantics there are two other cases. Either s_1 is not a call state and the result follows. Or s_1 is a call site, and has looping transitions for each event.

By induction on the length of *ews* then we can attempt to prove the result.

For the base case, consider that $ews = \langle\rangle$, then the result easily follows (since every state is related to itself in the transitive closure).
Then for the inductive hypothesis we assume that the result holds.
For the inductive case consider that $ews = ews' +\!\!+ \langle e, \omega\rangle$, and then we have to prove that $(q_1, \theta_1) \stackrel{ews'}{\Longrightarrow} (q', \theta') \xrightarrow{\langle e,\omega\rangle} (q_2, \theta_2) \Rightarrow ((s_1, q_1) \Rightarrow (s_1, q_2))$.
By the inductive hypothesis we know that $(s_1, q_1) \Rightarrow (s_1, q')$, and then we have to prove that $(s_1, q') \Rightarrow (s_1, q_2)$. By case analysis of the DEA operational semantics on $(q', \theta') \xrightarrow{\langle e,\omega\rangle} (q_2, \theta_2)$ then either $q' = q_2$ and the result follows, or $q' \neq q_2$ and $\exists\, e, c, a \cdot q' \xrightarrow{e|c\mapsto a} q_2$. By rule 4 of Definition. 8.2.1 then the result follows (since s_1 is a call state and there is a looping transition for e around it).

Case 2 : $s_1 \neq s$.

There are two cases here then, by the operational semantics of CFAs: (i) either $ews = \langle\rangle$ (i.e. no event is activated), from which the result easily follows (where $q = q_1$) by the third rule of Definition. 8.2.1; or (ii) $ews = \langle e, \omega\rangle$. The second case implies there is a transition $s_1 \xrightarrow{c \triangleright st \blacktriangleright e} s$, from which the result easily follows by the first rule of Definition. 8.2.1.

The required results follows easily by definition of transitive closure. □

This abstract monitored system will be used as the basis of both instrumentation reduction and property reduction. We can use this abstract monitored system to not only produce results about a method, but when constructing this for all the methods of a program we can infer properties of the whole program.

Recall that the CFA abstraction approximates the set of all executions through the abstracted method. Then, if we abstract each method in this way we have an abstraction of every possible path (in fact we can do better than this by choosing only one candidate for each symbolic path), and from such a collection we can produce guarantees about the whole program.

8.2.1.1 Interprocedural Compliance from Intraprocedural Analysis

In this section we are interested in determining whether a program CFA is compliant with a DEA. We consider several conditions that allow us to determine this from the abstract monitored system.

A simple compliance condition is to simply check if there is any property bad state reachable in the abstract monitored system. If there is not then clearly the program cannot ever be in a violating state.

Proposition 8.2.1. $\nexists q_B \in B, s_E \in E_P \cdot (s_{0_M}, q_0) \Rightarrow_{P\|_A\pi} (s_E, q_B)) \Rightarrow P \vdash \pi$

Proof This follows easily from Theorem. 8.2.1. Take the contrapostive of Theorem. 8.2.1, then LHS of the proposition we are trying to prove implies that any execution at runtime cannot reach a bad state in P, and then by the definition of the traces of CFAs and $\vdash$ we can conclude that $P \vdash \pi$.

As alluded to before, we can even formulate a stronger condition by considering the covers of P, as defined in Definition. 8.1.6, instead of all the methods at once. Then, since every cover induces an over-approximation of the program, if we manage to find one such cover where there is no violation we can conclude that the program does not violate the property.

Theorem 8.2.2. $(\exists Ms \in coversOf(P) \cdot \forall M \in Ms \cdot \nexists q_B \in B, s_E \in E_M \cdot (s_0, q_0) \Rightarrow_{M\|_A\pi} (s_E, q_B)) \Rightarrow P \vdash \pi$

Proof Take the contrapositive of Theorem. 8.2.1, then the LHS of the proposition we are trying to prove implies that any execution at runtime cannot reach a bad state in any $M \in Ms$, and if the abstract traces of Ms cannot reach a bad state then neither can the abstract traces of P (since by Theorem. 8.1.1 the union of all abstracted traces of each method in Ms cover the abstract traces of P).

However, initial states, call states, and final states allow for very coarse behaviour, which makes these conditions very weak in practice. There will be many spurious violations that occur at coarsely approximated states which do not occur in the actual program at runtime.

Consider then that we are analysing each method in the program in a sound manner. Then, if the program violates the property there will be some method for which a state belonging to it is associated with a bad state of the property. More strongly, since we are analysing all the possible methods called at runtime, if the program violates the property then some method will induce a property bad state by taking a concrete transition (as opposed to an abstract transition). Consider then that if all violations in all method abstractions occur because of events recorded from the abstract transition function then there is no violation, since if such violations were real they would also occur using some local concrete transition in some other method.

Thus a stronger condition for compliance is to simply check that no method abstraction induces an internal violation.

Theorem 8.2.3. $(\forall M \in methodsOf(P) \cdot \nexists q \in (Q \setminus B), q_B \in B, s, s' \in S_M \cdot (s,q) \xrightarrow[y]{x} (s', q_B) \wedge x \neq \square) \Rightarrow P \vdash \pi$

Proof We prove this by considering its contrapostive.

Suppose $P \nvdash \pi$, then we have to show that $\exists M \in methodsOf(P) \cdot \exists q \in (Q \setminus B), q_B \in B, s, s' \in S_M \cdot (s,q) \xrightarrow[y]{x} (s', q_B) \wedge x \neq \square$.

Consider $P \nvdash \pi$, this means that there is some trace in P that is violating, i.e. $\exists ews \in traces(P), s_E \in E_P, \theta_B, \omega_0, \omega_E \cdot (s_0^{\downarrow}, \omega_0) \stackrel{ews}{\Longrightarrow} (s_E^{\uparrow}, \omega_E) \wedge (q_0, \theta_0) \stackrel{ews}{\Longrightarrow} (q_B, \theta_B)$.

Without loss of generality, we can re-rewrite ews to isolate the step that causes the violation, i.e. $\exists ews', ews'', ews'' \cdot ews = ews' +\!\!+ ews'' +\!\!+ ews''', q_1 \in (Q \setminus B) \cdot (s_0^{\downarrow}, \omega_0) \stackrel{ews'}{\Longrightarrow} (s_1^a, \omega_1) \stackrel{ews''}{\longrightarrow} (s_2^b, \omega) \stackrel{ews'''}{\Longrightarrow} (s_E^{\uparrow}, \omega_E) \wedge (q_0, \theta_0) \stackrel{ews'}{\Longrightarrow} (q_1, \theta_1) \stackrel{ews''}{\Longrightarrow} (q_B, \theta_B) \stackrel{ews''}{\Longrightarrow} (q_B, \theta)B$.

By Theorem. 8.2.1 in the abstract monitored system we have that $(s_1, q_1) \Rightarrow (s_2, q_B)$.

There are two cases here:

Case 1 : $s_1 \neq s_2$ and then by the operational semantics of CFAs $\exists c, st, e \cdot s_1 \xrightarrow{c \triangleright st \blacktriangleright e} s_2$, and by the definition of the operational semantics of DEAs $\exists g, a \cdot q_1 \xrightarrow{e|g \mapsto a} q_B$, and then by rule 1 Definition. 8.2.1 $(s_1, q_1) \xRightarrow[e|g \mapsto a]{c \triangleright st \blacktriangleright e} (s_2, q_B)$ and then the result follows with $M = P$.

Case 2 : $s_1 = s_2$ and the s_1 is a call site. Here we are assuming there is no infinite string of such calls, and then we can replicate the argument for the call at s_1.

□

The abstract monitored system is however very coarse, since we are branching on the possibility of a guard holding and not holding for every possible matching DEA transition. We can make this finer by considering a variable state abstraction, which we illustrate next.

8.2.1.2 Exploiting Variable Abstractions

Recall Listing. 8.1, a Java method that returns the next value in an iterator, if any, represented as the CFA Figure 8.3. While a property we desire to prove of this is

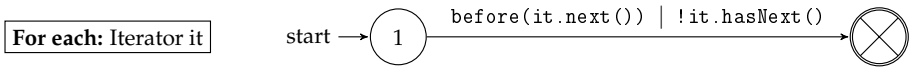


Figure 8.4: DEA specifying that an iterator over a list should only be queried for the next element when it also signals that it has a next element.

Figure 8.4. For simplicity we assume that operations on the iterator only occur in Listing. 8.1, and thus there is no behaviour outside the method to abstract and the abstraction of the CFA is the CFA itself. Consider the corresponding abstract monitored system in Figure 8.5. If we apply the described control-flow analysis to this example we would not be able to prove the property (although it clearly holds) since on the transition between state B and state C in Figure 8.3 we have to take both the transition between 1 and $\times$, and remain at state 1 in Figure 8.4. However, if we propagate the condition between state A and state B of the program then we can conclude that the transition to the bad state in the DEA cannot be taken.

Given a variable abstraction we then want to be able to detect whether a guard is contradicted by a condition set or not. Here we do this by employing an theorem prover, namely an SMT solver [Barrett and Tinelli, 2018]. Here we do not assume a specific solver or an approach to solving this problem. Instead, we simply assume such a procedure that checks whether there are symbolic states on which a guards can be true, where it may signal that this is possible ($\top$), that it is not ($\bot$) or that it was not able to make a decision (?). The latter case is possible since we are not assuming the background theory chosen for the symbolic states is satisfiable.

Definition 8.2.3 (Guard Satisfiability)**.** *We assume a procedure sat* $\in 2^{Guard} \mapsto \{\top, \bot, ?\}$ *that:*

(i) *returns* $\top$ *only when there are symbolic states for which the input guard holds:* $(sat(gs) = \top) \Rightarrow \exists\, \omega, \theta \cdot \forall\, g \in gs \cdot g(\omega, \theta)$*, and*

(ii) *returns* $\bot$ *only when there are no symbolic states for which the input guard holds:* $(sat(gs) = \bot) \Rightarrow \nexists \omega, \theta \cdot \forall\, g \in gs \cdot g(\omega, \theta)$.

We say two guards are compatible *if their conjunction is satisfiable.*

To be able to compare conditions with guards we then define a lifting from conditions to guards, by simply adding an unused DEA symbolic state parameter.

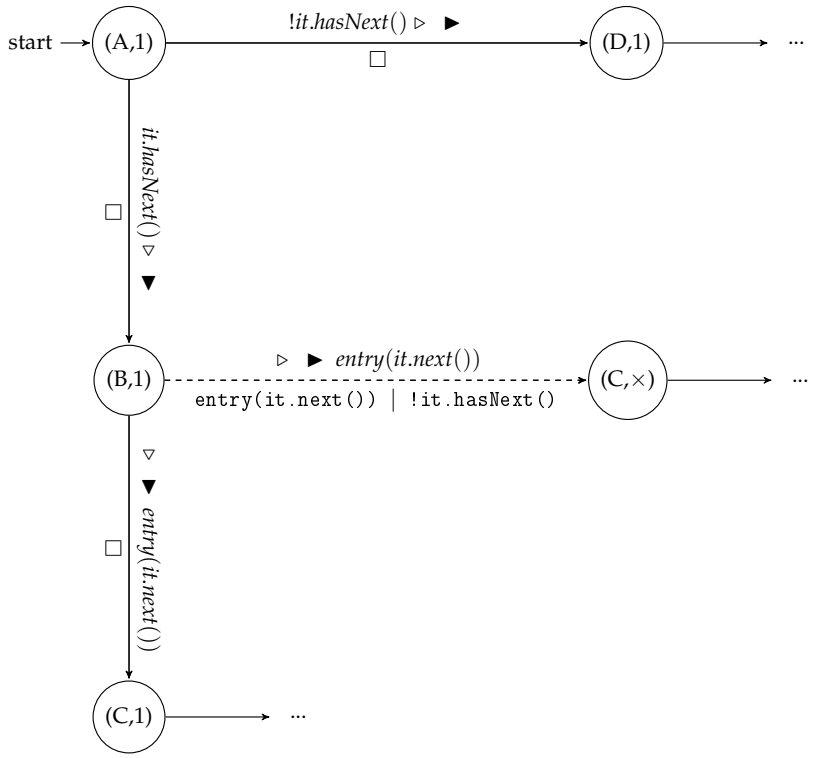


Figure 8.5: Partial view of the abstract monitored system of Figure 8.3 and Figure 8.4, with the dotted transition between $(B, 1)$ and $(C, \times)$ denoting the violating transition we wish to prune.

Definition 8.2.4. *The* lifting *of a program condition onto a property guard is the function* $\restriction_\Theta \in Cond \to Guard$ *defined as:* $(\lambda\, \omega.\ expr)\restriction_\Theta \stackrel{\text{def}}{=} \lambda\, \omega, \theta.\ expr$. *We overload this to sets of conditions, such that* $cs\restriction_\Theta \stackrel{\text{def}}{=} \bigwedge_{c \in cs} c\restriction_\Theta$.

Given such a lifting procedure we can prune an abstract monitored system, removing any non-viable transitions. We give a definition for this pruned system and discuss it afterwards.

Definition 8.2.5. *The abstract monitored system with respect to a variable state abstraction* $\rightsquigarrow \subseteq S \times 2^{Cond}$ *between a CFA* M *and property* π, *denoted by* $M\|^{\rightsquigarrow}_A \pi$, *is defined by the following rules:*

1. *A transition matching both a DEA and a CFA transition is kept only if the CFA state associated with its source configuration has a condition set whose update w.r.t. the*

transition is compatible with the guard.

$$\frac{(s,q) \xrightarrow[e|g\mapsto a]{c \triangleright st \blacktriangleright e} (s',q') \qquad \exists\, s \rightsquigarrow cs \cdot sat(g \wedge update(cs,(c,st))\restriction_\Theta) \neq \bot}{(s,q) \xrightarrow[e|g\mapsto a]{c \triangleright st \blacktriangleright e} (s',q')}$$

2. *A transition matching a DEA transition and an abstract transition is kept only if the CFA state associated with its source configuration has a condition set compatible with the guard.*

$$\frac{(s,q) \xrightarrow[e|g\mapsto a]{\square} (s',q') \qquad \exists\, s \rightsquigarrow cs \cdot sat(g \wedge cs\restriction_\Theta) \neq \bot}{(s,q) \xrightarrow[e|g\mapsto a]{\square} (s',q')}$$

3. *A transition matching no DEA transition is kept if the possibly matching DEA transitions do not have guards that cover all cases, i.e. their conjunction and negation is not incompatible with every abstraction.*

$$\frac{(s,q) \xrightarrow[\square]{c \triangleright st \blacktriangleright e} (s',q') \qquad \exists\, s \rightsquigarrow cs \cdot sat((\bigwedge_{q_1 \xrightarrow{e|g'\mapsto a} q_1'} \neg g') \wedge update(cs,(c,st))\restriction_\Theta) \neq \bot}{(s,q) \xrightarrow[\square]{c \triangleright st \blacktriangleright e} (s',q)}$$

4. *Any transition that does not match a property transition, and does not involve a local transition is kept.*

$$\frac{(s,q) \xrightarrow[\square]{\square} (s',q)}{(s,q) \xrightarrow[\square]{\square} (s',q)}$$

In the first case we consider that there is a concrete update in the CFA which matches a transition in the DEA. Note how a CFA state may have multiple condition sets associated with it, depending on the flow into the CFA, then to keep a transition here we need to check that the guard of the DEA transition is at least compatible with one such condition set. Note also however we are not using the condition set associated with the source state but the condition set after the statement *st* is executed. This is since the monitor is activated after this *st* is executed. A weaker version of this would be to consider the condition sets associated with s' since $update(cs,(c,st))$ is included in it (by rule 3 of Definition. 8.1.10), however there may be other condition sets associated with s' which we do not need to consider here.

In the second case, where an abstract update in the CFA matches a transition in the DEA we proceed similarly except that the condition sets associated with s' have already been pruned appropriately for statements that can occur outside of the method (by rules 3 and 4 of Definition. 8.1.10) and thus we can use it directly.

In the third case, when there is a concrete update in the CFA but no match in the DEA, we have to consider whether a no match is actually viable here. That is, we are checking whether the conjunction of the disjunction of every DEA transition outgoing from q that could have matched is satisfiable or not. If the guards are then determined to be mutually exclusive then there necessarily will be a transition triggered at this point and we can forego transitioning with no match. For example, consider that in the property we have $q \xrightarrow{e|g\mapsto a} q'$ and $q \xrightarrow{e|g'\mapsto a'} q''$ and no other transition from q. The third rule keeps a transition in the monitored system if there may be a variable state associate with s such that g and g' are both false when applied to $st(\omega)$. That is, there is the possibility that no transition matches.

For the last case, any abstract transition that matches no DEA transition is kept, since it may encode control-flow necessary to ensure a proper over-approximation.

We expect that the results using the purely control-flow abstract monitored system hold for this abstract monitored system, since the pruned DEA transitions are only removed when their guard cannot activate at runtime.

Theorem 8.2.4. *Any removed transitions in the abstract monitored system with respect to the variable abstraction cannot be exhibited at runtime.*

Proof We want to show that that a transition is not in the pruned abstract monitored system if it is impossible. We will be considering each pruning rule (1-3) separately. Throughout we will be using $(s,q) \overset{x}{\underset{y}{\not\rightarrow}} (s',q')$ to denote that the respective transition is in the original abstract monitored system but not in the reduced version.

1. $(s,q) \xrightarrow[e|g\mapsto a]{c \triangleright st \blacktriangleright e} (s',q')$

We want to show that such a transition is only removed if whenever the CFA transition triggers at runtime then the guard g does not hold on the runtime state.

By the first rule of Definition. 8.2.5, these types of transitions are only removed if there is no condition set associated with s such that the guard and the update of the condition set with the transition are compatible. That is we can conclude that $\forall\, s \rightsquigarrow cs \cdot sat(g \wedge update(cs,(c,st))\restriction_\Theta) = \bot$.

By definition of *sat* in Definition. 8.2.5, Theorem. 8.1.3, and the definition of *update* in Definition. 8.1.9 then the required result that g will never hold true on the concrete symbolic states easily follows.

2. $(s,q) \xrightarrow[e|g\mapsto a]{\square\!\!\!/} (s',q')$

We have several cases to consider, depending on whether the abstract transition is looping around an: (i) initial state; (ii) end state; or (iv) call state.

Consider throughout that such a transition is only removed, by rule 2, if $\forall\, s \rightsquigarrow cs \cdot sat(g \wedge cs\restriction_\Theta) = \bot$.

Case 1 : Initial state

The condition set associated with the initial state is always the empty set, thus no such transition is removed.

Case 2 : End state

In this case, for correctness, we have to show that there is no other (reachable) CFA in which the DEA transition in question can trigger. Consider that this is ensured by rule 4 of Definition. 8.1.10, since we are removing any conditions not invariant over the rest of the functions.

Case 2 : Call state

Here we require that any possibly called function cannot possibly activate the DEA transition in question. Consider that this is ensured by rule 3 of Definition. 8.1.10, since we are removing conditions that are not invariant over the possible calls.

3. $(s,q) \xrightarrow[\square]{c \triangleright st / \blacktriangleright e} (s',q')$

In this case we want to show that there must be a DEA transition that matches in the CFA transition in question at runtime

By the third rule of Definition. 8.2.5, these types of transitions are only removed if there is no condition set associated with s such that there is always some available guard that is compatible with the condition set. That is we can conclude that $\forall\, s \rightsquigarrow cs \cdot sat((\bigwedge_{q_1 \xrightarrow{e|g'\mapsto a} q_1'} \neg g') \wedge update(cs,(c,st))\restriction_\Theta) = \bot$.

If $e = \epsilon$ the result easily follows.

Consider then that $e \in \Sigma$.

By definition of the operational semantics of DEAs, Definition. 7.1.3, a CFA transition does not match a DEA transition if the program symbolic state it produces does not activate the guard of the transition, i.e. if the negation of the guard holds true. Then, the conjunction of all such negated guards represents the guard that must hold true for no transition to trigger. Consequently, if the

new guard is not compatible with the program state after the transition then there will always be a transition that matches.

Consider that these ensure the required result by the operational semantics of CFAs and DEAs, Definition. 7.1.3 and Definition. 7.2.4.

The results in the rest of the chapter do not depend on this reduction, but its application can optimise the results of residual analysis.

In the example we previously considered then the optimised abstract monitored system constructed is Figure 8.5 without the dotted transition, which allows us to determine that the program is compliant with the property.

Given we have discussed ways to infer compliance of the program at runtime, and shown how pruning the abstract monitoring system using a variable state abstraction can be useful we move on to describing the main novel contributions here, the production of residual instrumentation and properties for DEAs against CFAs.

8.2.2 Residual Analysis

In this chapter we discuss analyses of an abstract monitored system to identify instrumentation and DEA transitions that are not useful for a monitor at runtime and can be removed.

For simplicity, in this chapter we may use examples of CFAs with only event annotation, ignoring conditions and statements. Thus, whenever a CFA appears without conditions and statements in this chapter is to be assumed that there is a specific deterministic annotation that we are ignoring for illustrative purposes.

We will be using Figure 8.6 as a running example. This is a typestate property over a user u. In this section we assume the program has only one such user, leaving the case of multiple users for future work. This property is a specification of several privacy constraints in the context of a transaction system, using $\times$ to mark bad states, and $\checkmark$ to mark accepting states. It specifies that upon a user registering (the transition $1 \to 2$) they have a certain limit on transactions ($2 \to 2$ and $2 \to \times$) until they are authenticated ($2 \to 3$, we use *ret* as syntactic sugar to denote the return value of the `authenticate` function call). We also assume there are some functions in the program that can pass information to third parties, and that they can only be called if the privacy level of the user allows information flow to third parties ($3 \to \times$). Note, we let information loss events happen at state 2 since we assume sensitive information is only provided during authentication. Moreover upon de-registration, which is only allowed for authenticated users ($3 \to 4$) we require the data of the user maintained by the program to be sanitized

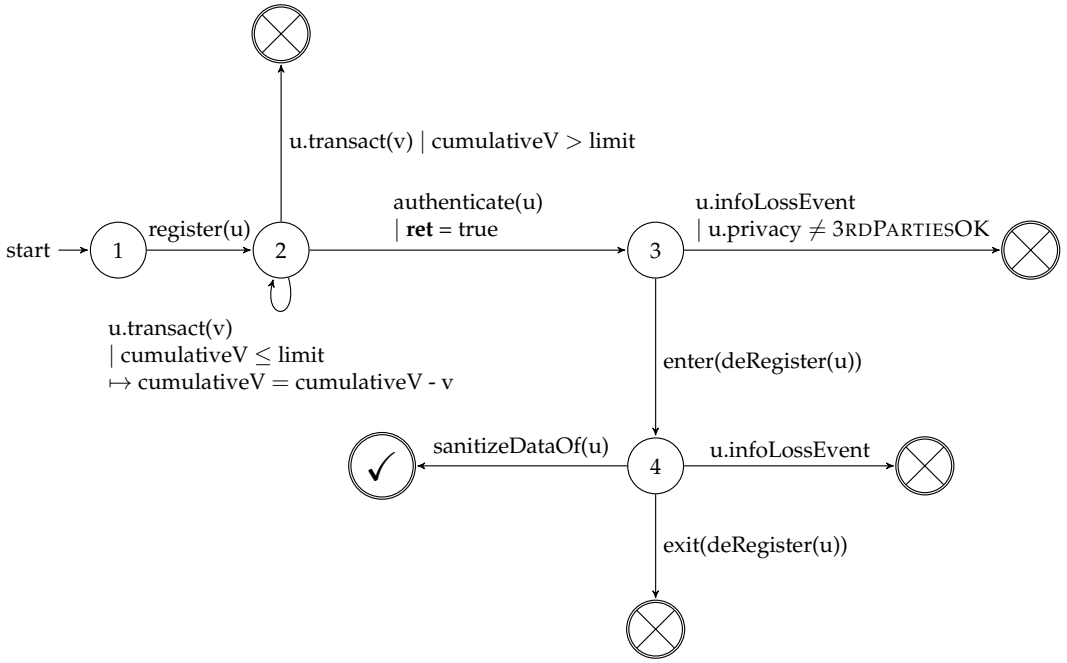


Figure 8.6: Running example of specification of a transaction system with certain data security and privacy guarantees.

$(4 \to \checkmark)$, otherwise if de-registration ends successfully without sanitation or if an information loss event occurs there is a violation $(4 \to \times)$.

8.2.2.1 Reducing Instrumentation

Consider that from an abstract monitored system we can identify those transitions of the CFA that never activate a DEA transition, i.e. they never cause a change in the property state in the configuration.

We characterise this by first considering the transitions of a method in M that are useful in the context of monitoring a DEA. Namely, we identify those transitions of the CFA whose source state can reach a bad state of the property. We also require that this source state is not already a bad state, since subsequent transitions will maintain this verdict and can be ignored.

Definition 8.2.6. *The useful transitions of a method M in the context of a program P, a property π, and a program abstraction $A \in CFA \to ACFA$ are those transitions of M that match some property transition in the abstract synchronous composition and whose source state is not a bad state but can reach a bad state: $useful(M, \pi) \stackrel{\text{def}}{=} \{s \xrightarrow{c \triangleright st \blacktriangleright e}_M s' \mid \exists s'' \in$*

$S_M, q \in (Q_\pi \setminus B_\pi), q' \in Q_\pi, q_B \in B_\pi, e \in \Sigma \cdot (s,q) \xrightarrow[e|g \mapsto a]{c \triangleright st \blacktriangleright e} (s',q') \wedge (s,q) \Rightarrow (s'',q_B)\}$.

Then silencing transitions in M that are not useful is a safe event reduction, since we are only removing instrumentation that affect verdict giving. We define this reduction.

Definition 8.2.7. *The* useless instrumentation reduction *of M with respect to a property π, in the context of a program P and a program abstraction $A \in CFA \to ACFA$, is the control-flow automaton $M \setminus \pi$ with a transition function containing all the useful transitions of M, but silencing event triggering for other transitions of M that are not useful: $\to_{M\setminus\pi} \stackrel{\text{def}}{=} useful(M,\pi) \cup \{s \xrightarrow{c \triangleright st \blacktriangleright \epsilon} s' \mid s \xrightarrow{c \triangleright st \blacktriangleright e}_M s' \wedge (s,c,st,e,s') \notin useful(M,\pi)\}$.*

Moreover calls in the reduced automaton are re-directed towards their respective instrumentation reduction: $calls_{M\setminus\pi}(s) \stackrel{\text{def}}{=} \lambda\, \omega.calls_M(s)(\omega) \setminus \pi$.

Note how we are keeping all those transitions from a state (s,q) that are reachable from the initial state, and that can reach a bad state (see Definition. 8.2.2). Consider there are two transitions that are only used in the following way in the abstract monitored system: $(s,q) \xrightarrow[e|g\mapsto a]{c \triangleright st \blacktriangleright e} (s',q') \xrightarrow[e'|g'\mapsto a']{c' \triangleright st' \blacktriangleright e'} (s'',q'')$ and (s,q) can be reached from the initial state and can reach bad state, but the latter conditions does not hold for (s',q'). Consider that since (s',q') cannot reach a bad state then upon having reached (s',q') in the composition we can determine satisfaction, and thus we can silence any future instrumentation. However, since (s,q) can reach a bad state we still need the transition to (s',q') since otherwise at runtime we may potentially affect the outcome of monitoring (i.e. if we silence the transition we will not reach a concrete version of (s',q') at runtime and may instead reach a state that can still violate).

For example, consider the running example and assume the following transitions is in its abstract monitored system Figure 8.6:

$$(s,4) \xrightarrow[sanitizeDataOf(u)]{\triangleright \texttt{sanitizeDataOf(u)} \blacktriangleright sanitizeDataOf(u)} (s',\checkmark) \xrightarrow[exit(deRegister(u))]{\triangleright \texttt{return;} \blacktriangleright exit(deRegister(u))} (s'',\checkmark)$$

If the transition between s' and s'' only matches in this part of the abstract monitored system, then we do not need to listen for *exit(deRegister(u))* at runtime, since a verdict will always have already been given upon it being activated.

Then we can conclude that this reduces the instrumentation of a method in a correct manner, since the removed transitions never cause progress towards a verdict.

Theorem 8.2.5. $P \stackrel{\pi}{=} P \setminus \pi$

Intuition Consider that $\overset{\pi}{=}$, by Definition. 7.3.4, requires that every possible trace prefix in the program is violating iff it is violating in the reduced program. However, $P \setminus \pi$, by Definition. 8.2.7, only removes transitions from states in the abstract monitored system that cannot reach a bad state. Then the removed instrumentation will never appear in a trace to a bad state.

Proof We need to show that $\forall\, \omega \in \Omega, i \in \mathbb{N} \cdot trace_i(P,\omega)\lfloor_\Sigma \in V(\pi) \Leftrightarrow trace_i(P \setminus \pi, \omega)\lfloor_\Sigma \in V(\pi)$.

Assume $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi)$, which by definition of $V(\pi)$ means $\exists\, q_B \in B_\pi, \theta_B \in \Theta \cdot (q_0,\theta_0) \xrightarrow{trace_i(P,\omega)\lfloor_\Sigma} (q_b,\theta_B)$.

By the operational semantics of CFAs, Definition. 7.2.4, then either there is a state s that can be reached using $trace_i(P,\omega)$, or $trace_i(P,\omega)$ reaches until the middle of a call:

$\exists\, s \cdot (s_0^{\downarrow},\omega) \xrightarrow{trace_i(P,\omega)} (s^a,\omega')$

or

$trace_i(P,\omega) = ews{+\!\!+}ews' \wedge \exists\, s \cdot (s_0^{\downarrow},\omega) \overset{ews}{\Longrightarrow} (s^{\downarrow},\omega') \xrightarrow{ews'{+\!\!+}ews''} (s^{\uparrow},\omega'')$

For the first case, by Theorem. 8.2.1 we can immediately conclude that $\exists\, s \in S_P, q_B \in B \cdot (s_0,q_0) \Rightarrow (s,q_B)$. This is also true for the second trace, since we can conclude $\exists\, q \cdot (s_0,q_0) \Rightarrow (s,q)$ and since s is a call state and by Definition. 8.1.2 we are allowing any action at s in the abstraction (and since we know q can reach a bad state), we can also state that $\exists\, s \in S_P, q_B \in B \cdot (s_0,q_0) \Rightarrow (s,q_B)$.

Now, assume for contradiction that $trace_i(P \setminus \pi,\omega)\lfloor_\Sigma \notin V(\pi)$.

If $trace_i(P,\omega) = trace_i(P \setminus \pi,\omega)$ there is a contradiction, since we know $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi)$.

Then consider that $trace_i(P,\omega) \neq trace_i(P \setminus \pi,\omega)$. Then, there is some position where in the second trace an event is silenced: $\exists\, j, e_j \in \Sigma \cdot trace_i(P,\omega) = (e_j,\omega_j) \wedge trace_i(P \setminus \pi,\omega) = (\epsilon,\omega_j))$.

Then for some $M \in methodsOf(P)$ there is a transition used by $trace_i(P,\omega)$ such that $s_1 \xrightarrow{c \triangleright st \blacktriangleright e}_M s_2$, that is silenced to $s_1 \xrightarrow{c \triangleright st \blacktriangleright \epsilon}_{M \setminus \pi} s_2$ and used by $trace_i(P \setminus \pi,\omega)$.

Consider that this M is P, then by Definition. 8.2.7 we can then conclude that $\nexists q,q' \in Q, q_B \in B, s_3 \in S \cdot (s_1,q) \twoheadrightarrow_{P\|\pi} (s_2,q') \wedge (s_1,q) \Rightarrow (s_3,q_B)$. By considering the contrapositive of Theorem. 8.2.1 we can then conclude that $\nexists ews,\omega,\omega', s_3^a, q_B \in B, \theta_B \cdot (s_0^{\downarrow},\omega) \overset{ews}{\Longrightarrow}_P (s_3^a,\omega') \wedge (q_0,\theta_0) \xrightarrow{ews\lfloor_\Sigma} (q_B,\theta_B)$. But this contradicts the assumption that $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi)$.

If M is not P but a method called by P, then we can repeat the same argument for M, but instead of starting from q_0 we start with the property state associated with the entry in M.

We have then shown that $trace_i(P, \omega) \in V(\pi) \Rightarrow trace_i(P \setminus \pi, \omega) \in V(\pi)$.

Assume that $trace_i(P \setminus \pi, \omega) \in V(\pi)$, and for contradiction that $trace_i(P, \omega) \notin V(\pi)$. Then the traces are different and there is some position where the first position has a silenced version of the pair in the other. However, using dual arguments as before, we can then conclude that $trace_i(P \setminus \pi, \omega)$ does not violate, since we can only turn off instrumentation that is not used on the way to a violation, which contradicts the assumption here.

□

Other instrumentation reductions are those of Bodden et al. [2010] and Dwyer and Purandare [2007]'s approaches, which we do not consider here.

In the next section we focus on property residuals, and show how property residuals can be produced by analysing the abstract monitored system. Producing residuals is used both to allow the user to analyse exactly what is left to prove of the original property, and to allow for subsequent proof attempts to exploit interprocedural information encoded in the reduction.

8.2.2.2 Property Residuals

The creation of residuals here follows the previous presentation of compliance checking, given that creating residuals is an extension of compliance checking.

Consider first that from each abstract monitored system we can infer a DEA, by considering all the DEA transitions used in the composition from states that can reach a bad state.

Definition 8.2.8. *The* simple residual *induced by a method M given a property π is the property $\pi \setminus M$ produced by considering all the transitions used in the respective abstract monitored system:* $\rightarrow_{\pi \setminus M} \stackrel{\text{def}}{=} \{q \xrightarrow{e|g \mapsto a} q' \mid \exists s, s', s'' \in S, q_B \in B_\pi, x \cdot (s, q) \xrightarrow[e|g \mapsto a]{x} \!\!\!\!\rightarrow (s', q') \wedge (s, q) \Rightarrow (s'', q_B)\}$. *We assume this is optimally reduced.*

For example, consider the abstraction Figure. 8.7(a) in the context of the running example Figure 8.6, our π here. Recall that Figure. 8.7(a) represents a CFA abstraction, but we are ignoring conditions and statements here for simplicity. Consider that $before(A) = after(D) = \Sigma$ then the simple residual will simply be equal to π, since all transitions in π would be used by the initial state A. However we can fine tune the abstraction of the outside behaviour

with a cheap and sound analysis by considering only the set of events that cannot occur in the other methods. Assume then that $before(A) = after(D) = \Sigma \setminus \{enter(deRegister(u)), exit(deRegister(u)), sanitizeDataOf(u)\}$. Then the simple residual of the abstraction with these parameters would be π without the transitions outgoing from state 4, and with state 4 as an accepting state (recall that the optimal reduction marks states that cannot violate as accepting), as shown in Figure. 8.8(a). To be useful then $\pi \setminus M$ requires at least some interprocedural information.

We can show then that the simple residual of π given P is itself a residual of π given P.

Theorem 8.2.6. $\pi \overset{\pi}{=} \pi \setminus P$

Intuition $\overset{\pi}{=}$, by Definition. 7.3.5, requires the two properties to agree on the prefixes of the traces of P that are violating. Consider that $\pi \setminus P$ does not contain only those transitions that cannot be used on the way to a bad state, and then they will never be used at runtime with a violating prefix, since the abstract monitored system is an over-approximation of the runtime monitored system by Theorem. 8.2.1.

Proof We need to show that $trace_i(P, \omega)\lfloor_\Sigma \in V(\pi) \Leftrightarrow trace_i(P, \omega)\lfloor_\Sigma \in V(\pi \setminus P)$.

Assume $trace_i(P, \omega)\lfloor_\Sigma \in V(\pi)$.

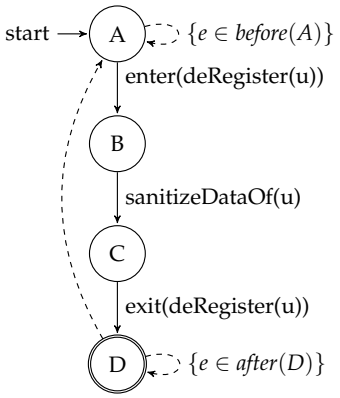
We can then conclude that $\exists\, q_B \in B, \theta_B \in \Theta \cdot (q_0, \theta_0) \xRightarrow{trace_i(P,\omega)\lfloor_\Sigma}_\pi (q_B, \theta_B)$ by definition of $V(\pi)$, Definition. 7.1.4.

Assume then that $trace_i(P, \omega)\lfloor_\Sigma \notin V(\pi \setminus P)$ for contradiction.

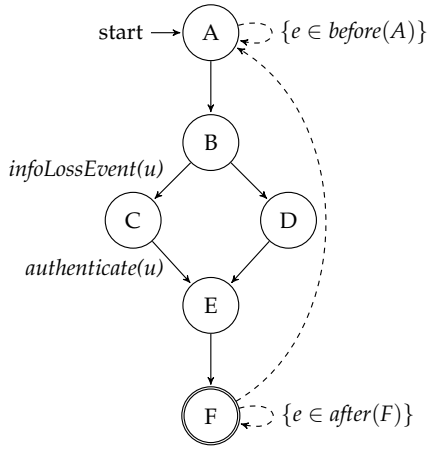
Then we can conclude from this and from Definition. 7.1.4 that this trace does not violate, i.e. $\nexists q_B \in B, \theta_B \in \Theta \cdot (q_0, \theta_0) \xRightarrow{trace_i(P,\omega)\lfloor_\Sigma}_{\pi\setminus P} (q_B, \theta_B)$.

Recall that $\pi \setminus P$ is a substructure of π and therefore the transitive closures operate over a subset of the transitions of π. Then at some point the two transitive closures diverge, and at this point there is some transition used in the execution in π that is not used in $\pi \setminus P$'s execution. That is $(q_0, \theta_0) \overset{ews}{\Longrightarrow}_\pi (q, \theta) \xrightarrow{ew}_\pi (q', \theta') \overset{ews}{\Longrightarrow}_\pi (q_B, \theta_B)$ but $\exists\, q_2 \notin B_\pi \cdot (q_0, \theta_0) \overset{ews}{\Longrightarrow}_\pi (q, \theta) \xrightarrow{ew}_{\pi\setminus P} (q, \theta) \overset{ews}{\Longrightarrow}_{\pi\setminus P} (q_2, \theta_2)$ such that there is some transition $q \xrightarrow{e|g\mapsto a}_\pi q'$ in π but $q \not\xrightarrow{e|g\mapsto a}_{\pi\setminus P} q'$.

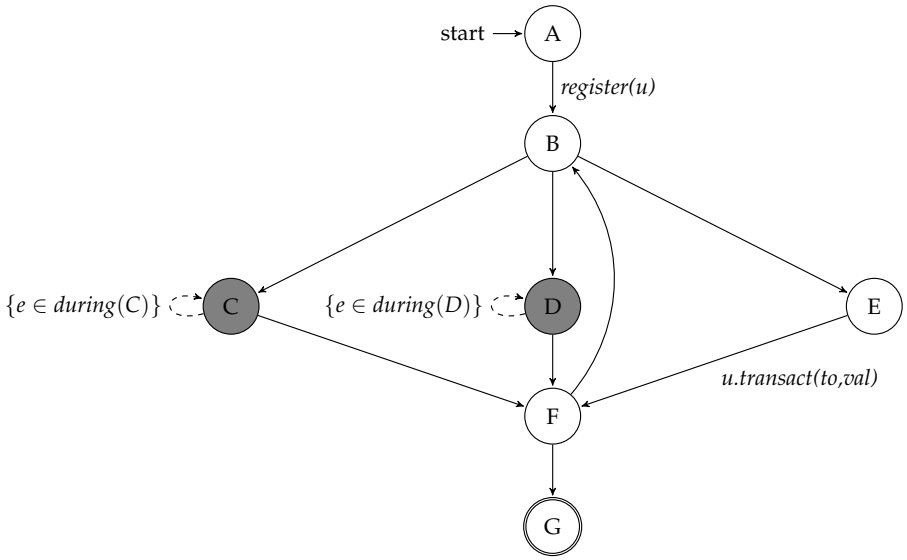
From $(q_0, \theta_0) \overset{ews}{\Longrightarrow}_\pi (q, \theta) \xrightarrow{ew}_\pi (q', \theta') \overset{ews}{\Longrightarrow}_\pi (q_B, \theta_B)$, the starting assumption, the definition of $trace_i$ (Definition. 7.3.3), and Theorem. 8.2.1 we can conclude that $\exists\, s', s'' \in S_P \cdot (s_0, q_0) \Rightarrow (s', q) \Rightarrow (s'', q_B)$. Here, if we are the



(a) Abstraction of the `deregister` method.

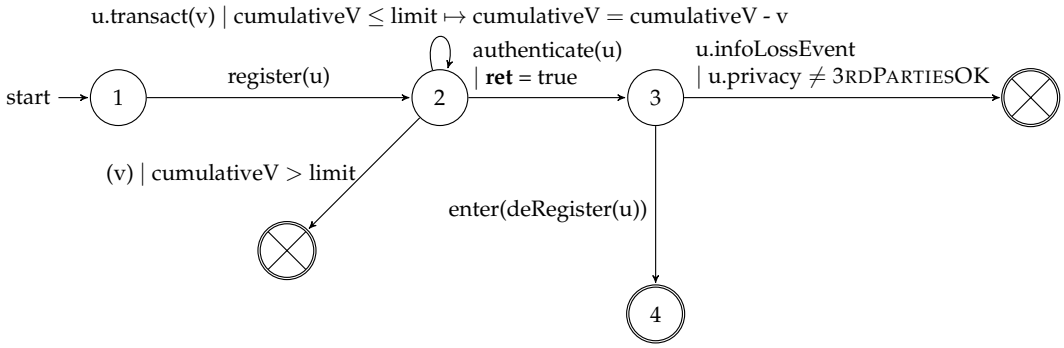


(b) Abstraction of `authenticationProcess` method that branches, and authenticates the user after the branches join.

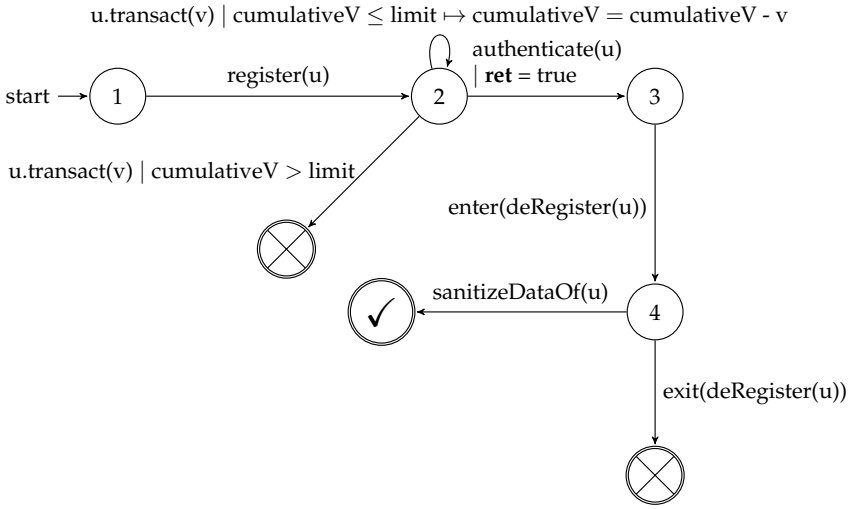


(c) Main method P, where state C calls `authorisationProcess` and state D calls `deregister`.

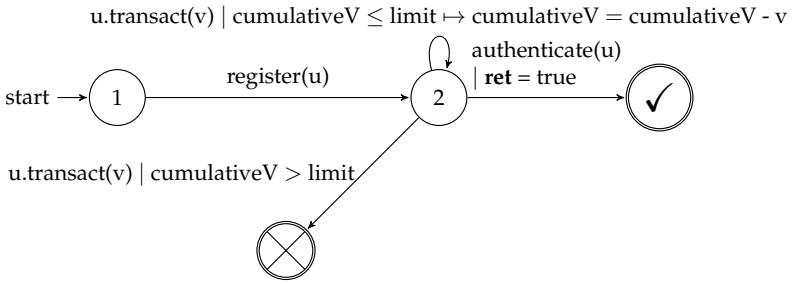
Figure 8.7: Example program.



(a) Residual of Figure. 8.7(a), with $before(A) = after(D) = \Sigma \setminus \{enter(deRegister(u)), exit(deRegister(u)), sanitizeDataOf(u)\}$.

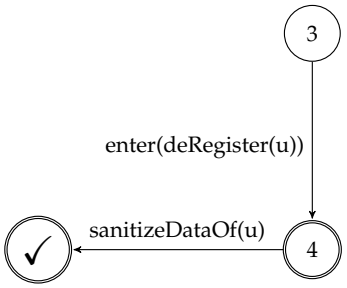


(b) Residual of Figure. 8.7(b), with $before(A) = after(F) = \Sigma \setminus \{infoLossEvent(u)\}$ and $during(C) = infoLossEvent(u)$.

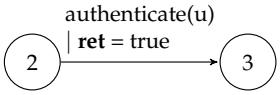


(c) Optimised structural intersection of residuals in Figure 8.8.

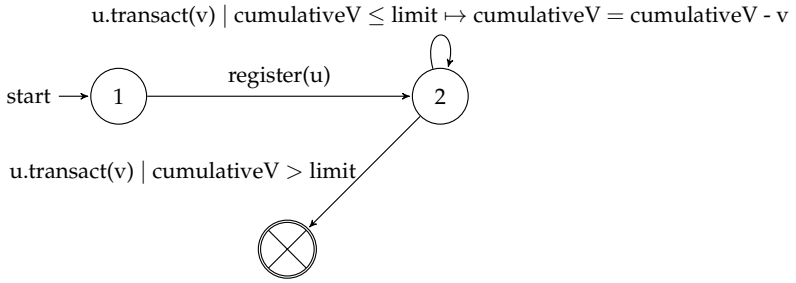
Figure 8.8: Simple residuals.



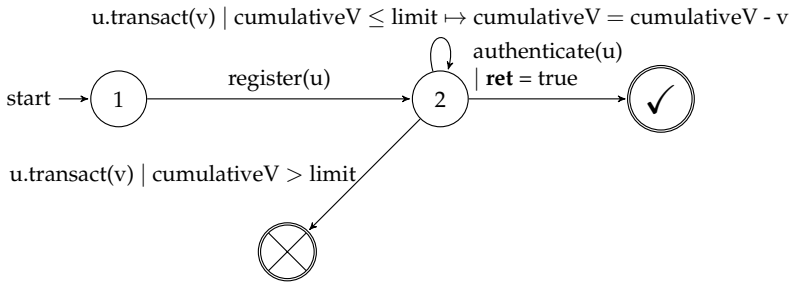
(a) Residual locally used transitions of Figure. 8.7(a), with $before(A) = after(D) = \Sigma \setminus \{enter(deRegister(u)), exit(deRegister(u)), sanitizeDataOf(u)\}$.



(b) Residual locally used transitions of Figure. 8.7(b), with $before(A) = after(F) = \Sigma \setminus \{infoLossEvent(u)\}$ and $during(C) = infoLossEvent(u)$.



(c) Residual locally used transitions of Figure. 8.7(c).



(d) Structural union and optimisation of locally used transitons.

Figure 8.9: Transitions used by each method, and resulting program residual.

execution prefix leads to configuration of P this works, while if it is part of a call s' is simply the state in P making the call.

However this allows us to elicit a contradiction, since from $q \xrightarrow{e|g\mapsto a}\!\!\!\!\!\!/\;\;\; _{\pi\setminus P}\, q'$ we can conclude by Definition. 8.2.8 that $\nexists s, s', s'' \in S, q_B \in B_\pi \cdot (s,q) \xrightarrow[e|g\mapsto a]{x} (s',q') \wedge (s,q) \Rightarrow (s'',q_B)$.

We have then proved that $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi) \Rightarrow trace_i(P,\omega)\lfloor_\Sigma \in V(\pi \setminus P)$.

Assume $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi \setminus P)$ and for contradiction that $trace_i(P,\omega)\lfloor_\Sigma \notin V(\pi)$. We can then conclude dually that the two executions diverge, where both reach some state q with the same trace, $(q_0,\theta_0) \overset{ews}{\Longrightarrow}_{\pi\setminus P} (q,\theta) \overset{ew}{\Rightarrow}_{\pi\setminus P} (q,\theta)$ and $(q_0,\theta_0) \overset{ews}{\Longrightarrow}_{\pi} (q,\theta) \overset{ew}{\Rightarrow}_{\pi} (q',\theta')$.

Then there is a transition $q \xrightarrow{e|g\mapsto a}_\pi q'$ by definition of Definition. 8.2.8 that is removed in $\pi \setminus P$, from which we can conclude that $\nexists s, s', s'' \in S, q_B \in B, x \cdot (s,q) \xrightarrow[e|g\mapsto a]{x}_{P\|\pi} (s',q') \wedge (s,q) \Rightarrow_{P\|\pi} (s'',q_B)$.

The first conjunct of this last statement is true then, by Definition. 8.2.2, since from the previous propositions we can also conclude, given Theorem. 8.2.1 and from the initial assumption, that $\exists\, s', q_B \in B \cdot (s_0,q_0) \Rightarrow_{P\|(\pi\setminus P)} (s,q) \Rightarrow_{P\|(\pi\setminus P)} (s',q_B)$ (if $trace_i(P,\omega)$ leads to a state that is not in P this still holds since call states in the abstraction are chaotic per Definition. 8.1.2). And then the second conjunct must be false.

From that, using the contrapositive of Theorem. 8.2.1, we can then conclude that $\nexists s_1^a, \omega_1, q_B, \theta_B \cdot (s^\downarrow,\omega) \overset{ews}{\Longrightarrow} (s_1^a,\omega_1) \wedge (q_0,\theta_0) \overset{ews}{\Longrightarrow} (q_B,\theta_B)$. But this contradicts the previous statement.

We can then conclude that $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi \setminus P) \Rightarrow trace_i(P,\omega)\lfloor_\Sigma \in V(\pi)$, from which we conclude the theorem. □

This residual however uses only the main method's CFA, while we can also use the CFA's called by it, as done for compliance checking. Recall that a method abstraction only abstracts the executions that pass through it, and not necessarily all the program, and that we can abstract the whole-program's behaviour by considering covers of the program.

Theorem 8.2.7. $\forall\, Ms \in coversOf(P) \cdot \pi \overset{\pi}{=} \bigsqcup_{M\in Ms} \pi \setminus M$

Proof: Consider that each abstraction of a trace of P is contained in the union of the abstract traces of the methods in a cover, by Theorem. 8.1.1. Consider that each $\pi \setminus M$ is enough to monitor the traces of M by Definition. 8.2.6 equivalently to π. Then since each trace is a trace of some M the transitions it can use on the way to a violation are present in $\bigsqcup_{M \in Ms} \pi \setminus M$, giving us the result required. □

Consider further the abstraction of another method in Figure. 8.7(b), with $before(A) = after(F) = \Sigma \setminus \{infoLossEvent(u)\}$ and $during(C) = infoLossEvent(u)$, and its simple residual in Figure. 8.8(b). If we consider the union of this residual with the previous method's residual, Figure. 8.8(a) we almost get back to the original property, thus without having made any interprocedural gains.

However, consider that we know that both these abstractions are abstractions of the programs, i.e. they both, separately, cover the whole program. Then we know each residual is enough to monitor for at runtime, and we can choose one or the other. However, we can do better by considering instead their structural intersection, and producing the property in Figure. 8.8(c).

Consider that if the reduction associated with each cover is correct and since each cover of a program P produces a correct residual, then each of the produced residuals is correct, and any pruned transitions and states are not reachable in the program (or can only be reached from states that cannot violate). Thus their structural intersection suffices.

Theorem 8.2.8. $interscoversred\ \pi \stackrel{\pi}{=} \bigsqcap_{Ms \in coversOf(P)} \bigsqcup_{M \in Ms} \pi \setminus M$

Proof Consider that any cover is an over-approximation of the program, and thus any $\bigsqcup_{M \in Ms} \pi \setminus M$ is enough to monitor the program with, by Theorem. 8.2.7. Some over-approximations may be finer than others, leading us identify DEA transitions as useless other over-approximations soundly lead us to include. Then by considering the structural intersection of these residuals we identify exactly those DEA transitions necessary for any program abstracted by $\bigcap_{Ms \in coversof(P)} \bigcup_{M \in Ms} atraces(M)$, which includes program P (a simple corollary of Theorem. 8.1.1). □

We have applied these analyses to a fine-tuned abstraction, i.e. where the chaos at start, end, and call states is limited by some contextual information. This information may be expensive to compute for larger programs, since it requires some intraprocedural analysis, and which may be too expensive for larger programs. Consider also that the success of the analyses we presented depends largely on this fine-tuned context, looking at Figure. 8.7(b) and supposing that $before(A) = after(F) = \Sigma$, i.e. all events, then all the property will be traversed at

the initial state, and thus $\pi \setminus M = \pi$. In this case however we can still do some analysis, that is comparable to one done on a more finely-tuned abstraction.

Recall the condition for compliance, in Theorem. 8.2.3, we specified is based only whether property bad states are induced by a transition local to a method. Consider then that from a single method we only need the DEA transitions that can be triggered by a local transition of the method, ignoring those only triggered by abstract transitions. Note how by collecting the set of such transitions from each method of the program then we get the set of DEA transitions possibly used in each execution of the program, as illustrated in Figure 8.9. In fact, if we consider Figure. 8.7(c), Figure. 8.7(b), and Figure. 8.7(a) in this manner we will immediately produce the residual in Figure. 8.8(c), with less effort than then the previous approach.

Then, from an abstract monitored system we will soundly identify the DEA transitions that can be used by a method M during its executions, namely those that appear on an internal non-abstract transition of M in the composition. By collecting all these concrete updates associated with each method of a program P, including P, we can construct a property that is a sub-structure of the original property.

Definition 8.2.9. *The residual of a DEA π given a program P, denoted by $\pi \setminus\!\!\setminus P$, is the property π but with the transition relation containing only the DEA transitions used from states that can violated in the abstraction of the methods of program P (including P):*

$\rightarrow_{\pi \setminus\!\!\setminus P} \stackrel{\text{def}}{=} \{q \xrightarrow{e|g \mapsto a} q' \mid \exists M \in methodsOf(P) \cdot \exists s, s', s'' \in S_M, q_B \in B_\pi \cdot (s,q) \xrightarrow[e|g\mapsto a]{x} (s',q') \wedge (s,q) \Rightarrow (s'',q_B) \wedge x \neq \square\}$

We assume this property is optimally reduced.

This property is in fact a residual of the original property with respect to the program.

Theorem 8.2.9. $\pi \stackrel{\pi}{=} \pi \setminus\!\!\setminus P$

Proof We need to show that $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi) \Leftrightarrow trace_i(P,\omega)\lfloor_\Sigma \in V(\pi \setminus\!\!\setminus P)$.

Assume $trace_i(P,\omega)\lfloor_\Sigma \in V(\pi)$.

Then by Definition. 7.3.3, we can conclude that $\exists\, q_B \in B, \theta_B \in \Theta \cdot (q_0,\theta_0) \xrightarrow{trace_i(P,\omega)\lfloor_\Sigma}_\pi (q_B,\theta_B)$ by definition of $V(\pi)$, Definition. 7.1.4.

Now, assume for contradiction that $trace_i(P,\omega)\lfloor_\Sigma \notin V(\pi \setminus\!\!\setminus P)$.

Then by definition of $V(\pi \setminus\!\!\setminus P)$, Definition. 7.1.4, we can conclude that $\nexists q_B \in B, \theta_B \in \Theta \cdot (q_0,\theta_0) \xrightarrow{trace_i(P,\omega)\lfloor_\Sigma}_{\pi \setminus\!\!\setminus P} (q_B,\theta_B)$ by definition of $V(\pi)$, Definition. 7.1.4.

Then at some point the two monitors must diverge: $\exists\, q, q' \in Q_\pi, \theta, \theta', \theta_B \in \Theta(q_0, \theta_0) \xRightarrow{ews_1\lfloor_\Sigma}_\pi (q, \theta) \xrightarrow{(e,\omega')}_\pi (q', \theta') \xRightarrow{ews_2\lfloor_\Sigma}_\pi (q_B, \theta_B)$, and $\exists\, q, q', q'' \in Q_\pi \setminus B_\pi, \theta, \theta', \theta_B \in \Theta(q_0, \theta_0 \cdot (q_0, \theta_0) \xRightarrow{ews_1\lfloor_\Sigma}_\pi (q, \theta) \xrightarrow{(e,\omega')}_\pi (q, \theta) \xRightarrow{ews_2\lfloor_\Sigma}_\pi (q'', \theta'')$, where $trace_i(P, \omega)\lfloor_\Sigma = ews_1 +\!\!+ \langle (e, \omega) \rangle +\!\!+ ews_2$.

This difference, by Definition. 7.1.3 then occurs because there is a transition in π from q that is not in $\pi \setminus\!\!\setminus P$, i.e. $q \xrightarrow{e|g\mapsto a}_\pi q' \wedge q \xrightarrow{e|g\mapsto a}\!\!\!\!\!\!/\;\;\; _{\pi\setminus\!\!\setminus P} q'$.

From this, by Definition. 8.2.9 we can conclude that the following condition is not satisfied for this transition: $\nexists M \in methodsOf(P) \cdot \exists\, s, s', s' \in S_M, q_B \in B_\pi \cdot (s, q) \xrightarrow[e|g\mapsto a]{x} (s', q') \wedge (s, q) \Rightarrow (s'' \in q_B \wedge x \neq \square)$.

However, by definition of $trace_i$, Definition. 7.3.3 and the CFA semantics Definition. 7.2.4, either the program trace prefix leads to a state in P: $\exists\, s_1^a, \omega_1 \cdot (s^\downarrow, \omega) \xRightarrow{trace_i(P,\omega)}_P (s_1^a, \omega_1)$; or it stops during some call: $\exists\, s_1^a, \omega_1 \cdot (s^\downarrow, \omega) \xRightarrow{ews}_P (s_1^\downarrow, \omega_1) \xrightarrow{ews' +\!\!+ ews''}_P (s_1^\uparrow, \omega_1') \wedge trace_i(P, \omega) = ews' +\!\!+ ews''$. Both this cases contradict the residual condition above, since in the first case then the required transition is in the abstract monitored system of P, while in the latter it is in the abstract monitored system of some transitively called method at s_1 by the CFA semantics Definition. 7.2.4.

Then we have shown that $trace_i(P, \omega)\lfloor_\Sigma \in V(\pi) \Rightarrow trace_i(P, \omega)\lfloor_\Sigma \in V(\pi \setminus\!\!\setminus P)$

Assume $trace_i(P, \omega)\lfloor_\Sigma \in V(\pi \setminus\!\!\setminus P)$.

Then by definition of $V(\pi \setminus\!\!\setminus P)$, Definition. 7.1.4, we can conclude that $\exists\, q_B \in B, \theta_B \in \Theta \cdot (q_0, \theta_0) \xRightarrow{trace_i(P,\omega)\lfloor_\Sigma}_{\pi\setminus\!\!\setminus P} (q_B, \theta_B)$.

For contradiction assume $trace_i(P, \omega)\lfloor_\Sigma \notin V(\pi)$.

Then by definition of $V(\pi)$, Definition. 7.1.4, we can conclude that $\nexists q_B \in B, \theta_B \in \Theta \cdot (q_0, \theta_0) \xRightarrow{trace_i(P,\omega)\lfloor_\Sigma}_\pi (q_B, \theta_B)$.

Then the two monitors diverge at some property state q, then by Definition. 8.2.9 there is some property transition that is removed, i.e. we have $\nexists M \in methodsOf(P) \cdot \exists\, s, s', s' \in S_M, q_B \in B_\pi \cdot (s, q) \xrightarrow[e|g\mapsto a]{x} (s', q') \wedge (s, q) \Rightarrow (s'' \in q_B \wedge x \neq \square)$. However, projecting this onto the concrete trace by Theorem. 8.2.1 there is a contradiction here because $\pi \setminus\!\!\setminus P$ while at CFA state s and property/monitor state q is reaching a violating state, while in the abstract monitored system (which over-approximates the coupled

program and monitor possible runtime behaviour) we have determined that they cannot violate. □

In fact in the example we considered $\pi \setminus P$ reproduced the original property, illustrating how $\pi \setminus\!\setminus P$ can give better results than simply considering the union of simple residuals, without the need to consider covers of the program.

Here we settle on using $\pi \setminus\!\setminus P$, since it involves a less time- and memory-intensive computation, as opposed to an approach using simple residuals of methods and covers of a program since:

(i) they maintain a larger property for each method (as opposed to only the internally used transitions); and

(ii) they require an interprocedural analysis to identify cover sets of the program.

Moreover, this intersection of simple residuals in our running example produced the same residual as $\pi \setminus\!\setminus P$, as illustrated by Figure. 8.8(c), showing how we can make progress towards a proof without interprocedural analysis.

Lastly, we can consider another reduction, namely a reduction of the guards in a DEA. Consider that we are analysing each possible method participating in an execution at runtime, and we are also removing any non-viable CFA and DEA transition matches by utilising a variable abstraction. Then, we can identify cases where a DEA transition is always activated, i.e. if whenever the DEA transition can be triggered it is triggered. Then we can simply transform such a DEA transition by turning its guard into the *true* guard, and removing other alternate transitions that then cannot be activated.

To characterise this we consider when a guard holds at some program state given some event, i.e. when the negation of the guard is not compatible with any of the possible condition sets after the event occurs at that program state in its method's abstraction. Then, given such a guard on a transition we can check that anywhere this DEA transition is used in the abstract monitoring system it necessarily holds, and then at runtime this transition will always be triggered.

Definition 8.2.10. *Given event e, guard g is said to hold at state s if whenever event e occurs the negation of the guard is not compatible with the program variable state:*
$holds(e,g,s) \stackrel{\text{def}}{=} \forall\, cs, s' \cdot (s \rightsquigarrow cs \wedge s \xrightarrow{c \triangleright st \blacktriangleright e} s') \Rightarrow sat(update(cs,(c,st))\lceil_\Theta \wedge \neg g) = \bot$

We overload this for property states by considering all the possible ways a property state can appear with a program state: $holds(e,g,q) = \forall M \in methodsOf(P), s \in S_M \cdot ((s_{0_M}, q_0) \Rightarrow (s,q)) \Rightarrow holds(e,g,s)$.

Then, if we determine that whenever a transition guard is used in the program it will evaluate to true we can forego checking it.

Definition 8.2.11. *The* guard residual *of a DEA π given a program P is the DEA $\pi \rtimes P$ that is equivalent to π except that transitions whose guards always hold are transformed to transitions with the true guard, and without any alternative transition from the same source state:* $\rightarrow_{\pi \rtimes P} \stackrel{\text{def}}{=} \{q \xrightarrow{e|true \mapsto a} q' \mid q \xrightarrow{e|g \mapsto a}_{\pi} q' \wedge holds(q, e, g)\} \cup \{q \xrightarrow{e|g \mapsto a}_{\pi} q' \mid \nexists q'', g', a \cdot q \xrightarrow{e|g' \mapsto a'}_{\pi} q'' \wedge holds(q, e, g')\}$.

We assume this property is optimally reduced.

We can show this is also a residual of π with respect to program P.

Theorem 8.2.10. $\pi \stackrel{\pi}{=} \pi \rtimes P$

Proof: Consider that by Theorem. 8.1.3 at least one of the condition sets associated with a state s contains conditions that are all true on the program states associated with s at runtime. Then, taking into account the condition of a transition and its statement will produce an approximation of the variable state associated with an event triggered. Then, we can carry the result of *holds* to runtime, and determine that when a transition is triggered it will necessarily activate.

This residual can be another step in reducing a property, and can be combined with any of the other two residuals we defined. In the next chapter we see some instances where this residual is useful. Note that dually we can define a residual that removes transitions that are always used in contexts where their guard evaluates to false. These transitions are however already removed by the reduction we define for the abstract monitored system.

8.3 Conclusions

In this chapter we have described control-flow analyses of a program represented as a CFA. By considering an intraprocedural abstraction, similar to that described by Bodden [2009], and composing it with a DEA we identified how we can reduce program instrumentation and produce property residuals for DEAs. We also defined a simple variable abstraction that propagates conditions on transitions and conditions implied by statements throughout a CFA until a statement that affects is possibly executed, which allows for a less coarse residual analysis. We silence instrumentation that never matches a DEA transition in the abstract monitored system, while we keep only DEA transitions matching concrete local transitions

of a method. This process of silencing instrumentation and reduction of the property can be repeated until a fixed-point is reached, resulting in an optimal result, modulo the analysis chosen. We also exploit the variable abstraction to identify property transitions whose guard always holds at the program states it is associated with at runtime, allowing us to forego some guard evaluation.

The analysis we have defined here is a *sound* one. We take the common position that we prefer to consider all the possible program behaviour, with the addition of some possible non-realisable violating paths, rather than miss some realisable violations. This kind of analysis does not allow us to conclude that any possible path to a violation is actually realisable at runtime, and therefore we cannot detect violating programs using our analysis. However, as discussed, we can use this analysis to detect programs that are satisfying.

9

Evaluation

The residual analysis we presented in the previous chapter is aimed to complement runtime verification by giving some static guarantees and thus reducing the proof obligation at runtime. In this chapter we evaluate this approach by considering several case studies and measuring: (i) how much of the property is proven statically; and (ii) how much runtime computation is avoided through the residual analysis. We consider both case studies in Java and Solidity [1].).

In Section 9.1 we describe briefly the case studies and relevant details of their implementation, while we describe and motivate the measures we use to measure the utility of the residual analysis. In Section 9.2 we present the results of the experiments and discuss their implications.

9.1 Methodology

In this section we describe the context of the experiments, the way we performed residual analysis on them, and describe and motivate the measures we take to evaluate the utility of the analysis.

9.1.1 Context

We applied this work to programs in two languages: Java, and Solidity.

[1] Solidity (`https://solidity.readthedocs.io`) is a language to write smart contracts for the Ethereum (`https://www.ethereum.org/`) blockchain.

The Java case studies presented here are based on a financial transaction system used for benchmarking LARVA and other RV tools in multiple instances of the *Competition on Runtime Verification* (CRV) [Bartocci et al., 2019; Reger et al., 2016]. We made this system larger by introducing different kinds of users and proxies for applications to communicate with the transaction server. We also added privacy and data protection concerns to the system. The specifications we constructed relate to limiting the behaviour of blacklisted users, and to the way the system should treat data associated with a user.

Solidity is a language used to specify smart contracts for the Ethereum blockchain. This is a different context from Java, since Solidity smart contracts tend to be on the smaller side. On Ethereum large and memory- or computation-intensive systems are discouraged through the requirement of the payment of *gas* (i.e. with tokens of real-world value) for the deployment of a smart contract onto the blockchain and for any function call at runtime, resulting in significantly simpler programs. We consider the implementation of smart contracts used to record the ordering and delivery of some objects for a courier service, and two iterations of a token wallet. The specification we consider for the courier service is a contract for the proper use of its interface (e.g. an order cannot be delivered before it is ordered), while for the wallets we consider a specification ensuring that once the balance of a user is reduced it is added to the balance of another user, ensuring the total balance over all the users cannot reduce over time.

9.1.2 Experimental Setup

9.1.2.1 Java

We implemented the control-flow residual analysis for Java programs in a prototype Java tool. We call this CLARVA in reference to the combination of the CLARA and the LARVA approaches. LARVA is then used for the RV part of the experiment.

The generation of CFAs from Java code is performed using the Soot Java bytecode analyser [Vallée-Rai et al., 1999] that is able to generate control-flow graph of the program. We simply further process this to identify relevant program events and to tag the graph with these events. An issue here could be a mismatch between the instrumentation we add here, and the instrumentation performed by the LARVA tool. The process taken in LARVA is to generate aspect specifications that are used by the AspectJ Kiczales et al. [2001] tool to add instrumentation points in matching program points.

The residual analysis implementation reflects closely the analysis as described formally: the control-flow graph of a program function is synchronously com-

posed with a DEA and this is analysed to identify useful program events and property transitions. The tool however does some richer analysis.

The property language of LARVA is not DEAs, but DATEs. These are an extension of DEAs with communication between different DATEs and with timing events. These events are handled by assuming they can be handled at any point. The tool also includes typestate analysis to project the analysis onto objects, using the pointer analyses offered by Soot, but however lacks a variable abstraction algorithm.

The Java experiments were carried out on an Ubuntu 19.04 machine, with an Intel Core i7-4500U 1.80 GHz CPU, and with 8gb of RAM. This tool is not sound for all Java programs, given limitations of Soot and since the presence of dynamic language features does not allow for this [Sui et al., 2018], and is used only as a proof-of-concept. We used this tool to perform both the residual analysis for the Java case studies, verifying its results manually and editing them manually to make the results sound when necessary. LARVA[2] was used to produce the files necessary for monitoring.

9.1.2.2 Solidity

For the Solidity language we developed a counterpart proof-of-concept tool SOLIDCLARVA in Haskell. For the RV part of the experiment the CONTRACTLARVA tool is used.

To generate a CFA of a smart contract we made use of the Solidity component of CONTRACTLARVA, by analysing the intermediate language it uses. Instrumentation in CONTRACTLARVA depends on the notion of modifiers in Solidity (basically function templates). On the other hand statically instrumentation is performed by simply inspecting the uninstrumented CFA. There is less possibility of error in mismatch here than in the Java case, since the event specification language here is more limited than that afforded by aspects.

Unlike CLARVA we implemented the described assertion propagation in SOLIDCLARVA, which enables a more fine-grained analysis. We extended the CONTRACTLARVA Solidity parser to extract from Solidity statements and expressions appropriate counterpart sentences in the SMT-LIB 2.0 language [Barrett et al., 2016], a standard language for SMT solvers. As described in the previous chapter, an SMT solver was used on the fly while constructing the synchronous composition of the program and property automata. This helps avoid exploring parts of the composition that are not viable at runtime. Our SMT solver of choice was the *z3 SMT solver* de Moura and Bjørner [2008].

[2] `http://www.cs.um.edu.mt/svrg/Tools/LARVA/`

The Solidity experiments were carried out using the Remix IDE[3], which allows the simulation of deployment and running of Ethereum smart contracts written with Solidity. CONTRACTLARVA [Azzopardi et al., 2018b; Ellul and Pace, 2018] was then used to produce the instrumented version of the smart contract, with some manual editing to add typestate (which is not yet implemented in the tool).

9.1.3 Measurements

The techniques we presented attempt to tailor a DEA property to a program, by leaving only the parts of the DEA that the program can violate (modulo the static analysis) and the parts of the program instrumentation that can violate. Then in this chapter we want to evaluate the effectiveness of our residual techniques after applying them to different case studies. We measure these static guarantees on two fronts: (i) how much of the property is statically guaranteed; and (ii) how much less monitoring overheads are required when monitoring the residual proof obligation.

Other approaches also measure the loss in precision resulting from using an approximate model of the system (e.g. Grech et al. [2018]). In our case the model is a sound one, and we only use it to attempt to determine satisfaction of the property. In future work it would be relevant to compare our approach with more precise model checking approaches, to determine how useful the tradeoff of our sound approach makes is.

9.1.3.1 Static Guarantees

The kind of analysis we are considered here acts on two objects required for verification: (i) the event instrumentation; and (ii) the property. Unlike Bodden [2009] and Dwyer and Purandare [2007] we do not focus on reducing instrumentation but in inferring what transitions of a property may be activated and which may not. However we are still reducing some instrumentation, while reducing a property may cause the program to be instrumented for less events if they are no longer used in the property.

For a program, we will then be comparing the number of the instrumented points before and after analysis, corresponding to transitions in the program CFA that originally were tagged with a non-ϵ event before the analysis, and with the ϵ event after the analysis. Note how if there are no instrumented points after the analysis then the property is satisfied.

Given a DEA and its residual, we also compare between them with two measures: (i) the difference in the number of transitions; and (ii) the difference

[3] `http://remix.ethereum.org/`

in the number of states. A reduction in transitions and states signals static guarantees that a program cannot trigger the removed transitions or visit the removed states, while if there are none left the property has been proven.

Measuring the difference in instrumentation points and in the property structure may not however give us an accurate view of reduction in effort required for subsequent proof attempts, for example the instrumentation points removed may be in points in the program seldom explored at runtime, requiring experiments to test the difference in runtime computation overheads.

9.1.3.2 Runtime Overheads

Synchronous monitoring causes computation overheads at runtime, i.e. the monitored program will perform more computation steps at runtime. Considering Java programs, from a user perspective there are two aspects relevant to their experience with the program: (i) the time taken for the program to execute; and (ii) the memory consumed by the program. The case studies did not present any significant memory consumption increases, thus here for Java programs we focus on the difference in time taken for the monitored with the original property and the residual property on the same behaviour as the measure for runtime overheads.

Then the Java program we consider, i.e. the financial transaction system, will be instantiated with certain behaviour with a significant amount of computation. The time taken without any monitors is then compared against monitoring with the original property and with monitoring with the residual property, giving a view of how much overheads monitoring causes and how much overheads residual analysis reduces for the considered case studies. The Java programs here then consist of a main method that performs this behaviour. To give a better view of the overheads and how monitoring scales we considering a range of users transacting in the system.

We cannot characterise overheads for Solidity programs (or smart contracts) in the same way, since functions in smart contracts tend to be small and not very intensive. However, instead there is already a notion of the computational expense of computation in the Ethereum blockchain, namely gas. Each kind of possible blockchain instruction (e.g. storage in an array) requires some unit of gas, as pre-defined by Wood [2014]. Moreover, this gas has a real-world value and thus reducing the amount required is preferable.

Smart contracts are programs with multiple public functions that can be called by users or other smart contracts. Then, to measure the overheads associated with a smart contract we do not need to exercise the smart contract as intensely as we do for Java programs. Instead we simply consider the overheads added to

each of these public functions. To get a better view of the monitoring overheads we consider both implementations of the case studies that are compliant and implementations that allow for violations, allowing us a view of the overheads of monitoring when there is no violation to be found and when there are violations.

In our experiments we then characterise overheads associated with monitoring Solidity smart contracts in two ways: (i) the increases gas cost of deploying a smart contract to the blockchain; and (ii) the increased gas cost associated with the instrumented functions of the smart contract.

9.1.4 Threats to Validity

Concisely, we want to measure the effect of adding a static analysis step to runtime verification on runtime overheads. The independent variable, or the input variable to our experiment, here then is a program-property pair. We consider two values for it: (i) the original program-property pair; and (ii) the residual program-property pair. Our dependent variable, or the output, is the time overheads measured during monitoring the input program with the input property.

The major threat here is that of lack of external validity, since we constructed most of the specifications and programs used ourselves for this purpose. Unfortunately existing benchmarks for RV focus on formalisms without symbolic aspects, preventing us from performing our experiments on existing specifications. This means that our results are not necessarily representative of RV with DEA variants in general. However, we attempt to show different kinds of specifications, including specifications for which our analyses fail to give any benefits, to support our argument.

Early in our experiments we also noted a confounding threat to internal validity. The measured time overheads varied in different runs of the experiment. We concluded that this was due to the machine used for experimentation performing other work in the background. To remedy this we ensured that no user-run programs ran in parallel with the program. We also ran the experiments several times and the final results was calculated as the average time taken for each run, adding confidence in the results.

9.2 Results

In this section we present the results of applying our techniques to the described case studies. The effectiveness of the presented techniques are then evaluated according to the collected results in terms of the static guarantees produced

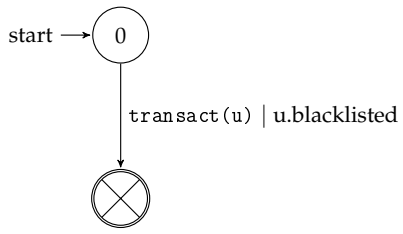


Figure 9.1: A blacklisted user cannot transact.

No. of Users	Unmonitored	Monitored	Monitoring Residual
1000	82.26s	173.31s	-
1050	83.45s	189.85s	-
1100	84.53s	211.32s	-
1150	94.37s	232.27s	-
1200	103.80s	253.45s	-
1250	113.09s	274.27s	-
1300	121.08s	318.05s	-
Average Overheads	0 %	236.07 %	-%

Table 9.2: Overheads associated with monitoring for Figure 9.1.

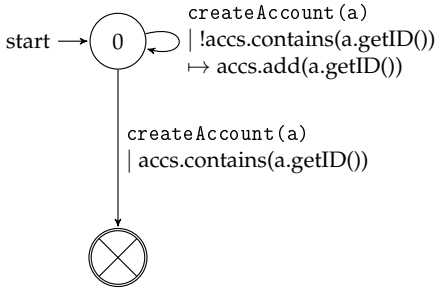


Figure 9.2: Accounts should have distinct account numbers.

No. of Users	Unmonitored	Monitored	Monitoring Residual
1000	82.26s	77.14s	-
1050	83.45s	87.51s	-
1100	84.53s	95.2s	-
1150	94.37s	104.14	-
1200	103.80s	110.37s	-
1250	113.09s	121.01s	-
1300	121.08s	129.47s	-
Average Overheads	0 %	103.55 %	-%

Table 9.3: Overheads associated with monitoring for Figure 9.2.

statically, and the runtime overheads reduced, as described in the previous section.

The experiments were conducted on two lines: (i) analysis with pure control-flow for Java; and (ii) analysis with assertion propagation Solidity. We present the results along with a discussion of the specific case study, and give a general overview of the implications of the results at the end of this section.

9.2.1 Analysis of Java programs

The results here will be presented in two ways. For each case study for illustrative purposes we give a snapshot of the running time of the program under verification for a different range of input values (in this case the number of users transacting with an upper bound of 1300 users, for tractability). Where our method does not manage to prove the property but instead returns a residual problem we created a sample of fifty runs. Here we present estimates about the reduced overheads due to our analysis based on analysing these sample runs.

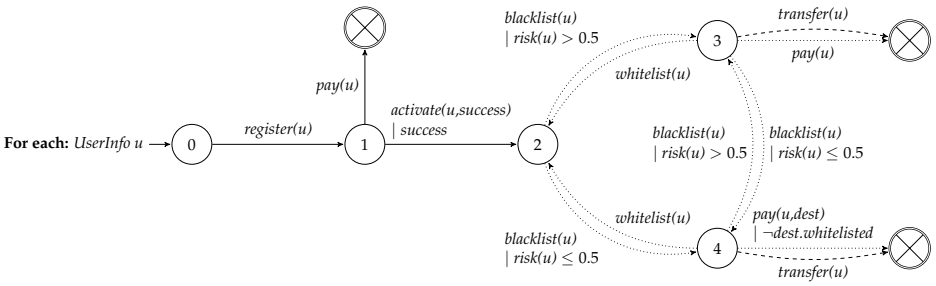


Figure 9.3: Property that regulates for the risk appetite of a client, with dashed transitions removed by the first analysis, and dotted by the third.

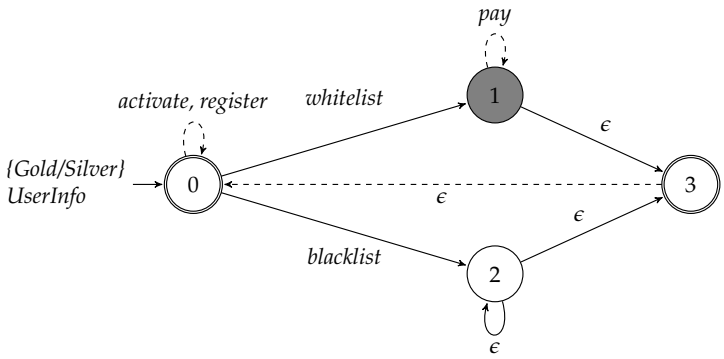


Figure 9.4: FiTs menu CFG lifted to Gold and Silver users, with respect to the property in Figure 9.3.

We look first at the property in Figure 9.1. This is a simple property with only one transition, that however produced significant overheads, as can be seen by in Table 9.2, in fact on average it caused the time taken to run the test traces by around 236%. Using pure control-flow analysis we could not prove this program. Consider that this property can only be reduced if the `transact` event does not occur in a program, which should not be possible in a transaction system. Analysing the data-flow here could be useful, if we are able to determine that the subject of the transaction is never allowed to be blacklisted. However, the check for this in the system is implemented through a function call, hiding the logic that checks whether a user requesting a transaction is blacklisted or not.

Residual analysis on the second property Figure 9.2 fails for the same reason, where a control-flow residual analysis is able to reduce the program only if an account cannot be created, which should never be the case for an appropriate implementation. On the other hand, here data-flow analysis is not useless because of the implementation of the program (where some data-flow is hidden in a function call), but because the property transitions only depend on the property's

No. of Users	Unmonitored	Monitored	Monitoring Residual
1000	82.26s	189.03s	100.12s
1050	83.45s	204.19s	109.18s
1100	84.53s	228.79s	122.32s
1150	94.37s	255.59s	132.01s
1200	103.80s	277.49s	148.53s
1250	113.09s	308.71s	151.12s
1300	121.08s	316.85s	163.20s
Average Overheads	0 %	157.40 %	34.78%

Table 9.4: Overheads(%) for program before, and after monitoring with residuals for data property, Figure 8.6, with residual Figure. 8.8(c).

No. of Users	Unmonitored	Monitored	Monitoring Residual
1000	82.26s	198.5s	178.12s
1050	83.45s	203.71s	191.75s
1100	84.53s	227.16s	205.93s
1150	94.37s	247.06s	225.18s
1200	103.80s	269.06s	242.47s
1250	113.09s	304.02s	275.34s
1300	121.08s	332.78s	300.67s
Average Overheads	0 %	144.15 %	134.39%

Table 9.5: Overheads (%) for program before, and after monitoring with residuals for risk property (Figure 9.3).

	Est. Mean	Est. Sd. Dev.	Confidence Interval Width (95%)	Coefficient of Variation
Monitored	206.39	37.97	18.40	10.63
Monitored (just instrum.)	198.77	39.13	19.69	10.96
Residual	196.53	34.67	17.64	9.71
Residual (just instrum.)	187.47	33.09	17.65	9.27

Figure 9.5: Estimations in terms of percentage of overheads from sample runs for data property, Figure 8.6, with residual Figure. 8.8(c).

	Est. Mean	Est. Sd. Dev.	Confidence Interval Width (95%)	Coefficient of Variation
Monitored	190.46	35.28	9.24	18.52
Monitored (just instrum.)	187.90	38.40	10.1	20.44
Residual	179.55	28.5	7.46	15.87
Residual (just instrum.)	177.39	30.57	8.01	17.23

Figure 9.6: Estimations in terms of percentage of overheads from sample runs for risk property (Figure 9.3).

variable state, which we do not abstract. This is a limitation of the techniques we present. However the overheads associated with this property are minimal, as can be seen in Table 9.3.

We consider two other properties with more sophisticated control-flow. One of these is the running example used in Chapter 8. This property contains both transitions with guards and without. Analysed against the transaction system (as described in Chapter 8) the dashed transitions can be removed, and state 4 transformed into an accepting state. As discussed in Chapter 8, the implementation of the system here allowed us to determine that certain events will not occur in certain contexts (e.g. an information loss event after authentication)

and that certain events will occur in certain contexts (e.g. data will always be sanitised after de-registration is requested). In this case data-flow analysis would not allow us to prune the property further, giving us a minimal residual (modulo the limitations of our analyses).

As can be seen in Table 9.4, monitoring this residual instead of the original property gives us a significant reduction in overheads, reducing the time taken from around 157% of the unmonitored version, to around 34%. Looking at a larger sample, Figure 9.5, we can determine a trend towards a reduction in overheads but at a smaller scale. The sample used here involved iterations with smaller numbers of users than in the snapshot Table 9.4, pushing the mean value down. Looking at the data we determined that the trend towards reduction in overheads is more pronounced the larger the number of users (and thus transactions) involved. From Figure 9.5 we can also determine that the majority of overheads are due to instrumentation, as opposed to the business logic of the property.

The last property we considered is Figure 9.3. This deals with a notion of risk, expecting certain certain behaviour when a user is blacklisted depending on their risk level, which is computed by the monitor using a computationally-intensive implementation. In the particular implementation, abstracted by automaton in Figure 9.4, we are able to determine that transfers are not allowed and that the possibility for payments is disabled when the user is blacklisted, allowing us to ignore a large part of the property for the particular implementation we consider. This only results in moderate overhead reduction here as ilustrated in Table 9.5, where the time taken in the residual version is only better by around 10%. In [Azzopardi et al., 2017c], we presented another version of this case study, with test traces that were less intensive, where they performed less transactions. In that case the overheads for this property almost reduced to 0%. The results of the larger sample here, shown in Figure 9.6 reflect those for the previous property.

With these case studies we saw how purely control-flow analysis can be useful in reducing a property when the both the program and the property make explicit some control-flow, instead of wholly encoding it in or making it dependent on data (note how Figure 9.3 monitors the action of blacklisting, as opposed to Figure 9.1 that checks the variable state of the user). More specifically we can identify two cases where the described control-flow analysis gave results: (i) when a DEA contains a sub-graph without guards (e.g. the required behaviour of the `deRegister` function in Figure 8.6); and (ii) when the program satisfies a DEA in a stricter way than required (e.g. the application does not perform information loss events after authentication or that a blacklisted user is not allowed any transactions after being blacklisted, although the specifications in Figure 8.6 and Figure 9.3 allow for this in a restricted manner). We can conclue

that pure control-flow analysis of DEAs is then useful either when parts of the DEA or relevant parts of the program lack branching based on data state, which is a severe limitation. In the next section we consider the effect of adding an abstraction of this data state.

9.2.2 Analysis of Smart Contracts

Here we consider several smart contracts and specifications for them, and illustrate when assertion propagation in the program can be useful, focusing on Ethereum blockchain use cases. In this context the specification is not just being monitored but also enforced, in that upon reaching a bad trace the execution is stopped and its effects on the variable state of the smart contract reverted, as illustrated and discussed in [Azzopardi et al., 2018b]. Here then we consider different versions of the smart contract under verification, namely a version that satisfies the property and a version that does not, to also measure the overheads of monitoring with reverting. Also, in a real-world context before runtime verification we do not know whether a program is compliant with the property or not, and then a sound evaluation of our techniques should consider their effect on both programs that satisfy and programs that violate the property. The expected result here is that our techniques should be able to reduce more of a property for compliant programs than for violating programs, if our analysis is able to determine at least partial compliance of the program.

We present the results of the analysis here differently from the Java case. Determining the gas cost of deploying a smart contract is straightforward and deterministic. Determining the gas cost of transactions is simply a matter of executing a number of transactions that exercise each path in the transaction. We then present the results for transaction cost in terms of a lower and upper bound gas cost. Here we do not need to use a confidence interval as with time measurements for the Java case, since the gas costs are predictable and deterministic. Loops and recursion could be issues for such an approach, however they are discouraged in Ethereum and thus do not appear in our case studies.

9.2.2.1 Courier Service

Consider the specification of a courier service in Figure 9.7. The specification requires that once a customer places an order (i.e. the function `order(no,eta)` is called for an order with identifier *no*) then it is either delivered to the proper customer or a refund is given to the customer by the owner of the smart contract. This property expects an implementation with three functions `order`, `delivered`, and `refund`, and with a global variable `mapping(int => address) customer`.

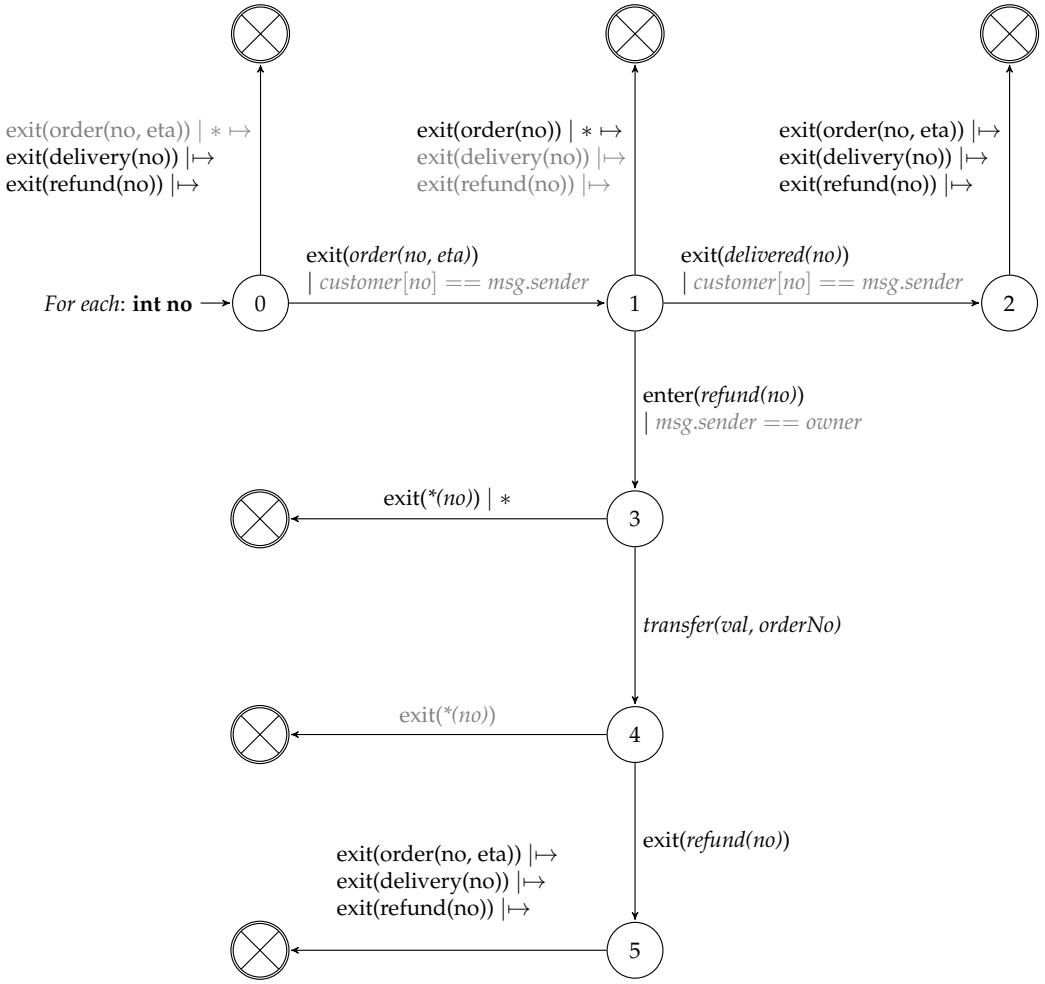


Figure 9.7: Courier service behavioural interface specification.

Consider an implementation of the expected behaviour in the smart contract with the functions defined in Listing. 9.1. Proceeding with the intraprocedural analysis with pure control-flow we cannot reduce the property, since all these functions act as the entry-points to the smart contract, and can be called in any order, however if we consider the simple assertion propagation we described (propagating an assertion until a statement that could affect its truth value) we can reduce the property significantly.

Since we are interested in only one order (note how Figure 9.7 is a typestate property on each order number), we can then fix an integer `no` when analysing the variable state of the smart contract. If we start by analysing the `order` function we can easily determine that the condition between state 0 and state 1 in the property is always satisfied by the order, since any successful call to the `order`

Listing 9.1: Courier Service smart contract.

```
function addOrder(uint orderNo, uint eta) payable public{
    require(msg.value >= cost);
    require(!ordered[orderNo]);

    customer[orderNo] = msg.sender;
    orderETA[orderNo] = eta;
    ordered[orderNo] = true;
}

function deliverySignature(uint orderNo) public{
    require(msg.sender == customer[orderNo]);
    require(ordered[orderNo] && !delivered[orderNo]);

    delivered[orderNo] = true;

    orderDeliveryTime[orderNo] = now;
}

function giveRefund(uint orderNo) public payable{
    require(msg.sender == owner);
    require(ordered[orderNo] && !delivered[orderNo] && !cancelled[orderNo]);

    (customer[orderNo]).transfer(cost);
    cancelled[orderNo] = false;
}
```

function executes the assign statement `customer[orderNo]` $=$ `msg.sender`, leaving the corresponding assertion as a post-condition of the function. Then, a call to `order` while at monitor state 0 will always match the transition to state 1, and the transition from 0 to a bad state with condition `customer[no]` $\neq$ `msg.sender`(here hidden with the $*$ syntactic sugar) can also be removed.

We can perform a similar analysis for the other functions, determining that a residual monitor with respect to the implementation is the part of Figure 9.7 without the transitions with the gray tags. Note how the rest of the DEA is also ensured by the smart contract, which however we cannot prove since we are limiting our analysis to be largely intraprocedural. Adding interprocedurality, through a flatter CFA abstraction would allow us to prove the rest of the DEA. In the context of Solidity smart contracts such interprocedural analysis is also more viable, given their size and complexity is limited. The smart contract used here is in fact very simple, since it does not have any loops or any dynamic calls to other smart contracts. However, it is representative of many desirable smart contracts since calls and loops that depend on a dynamic variable are respectively

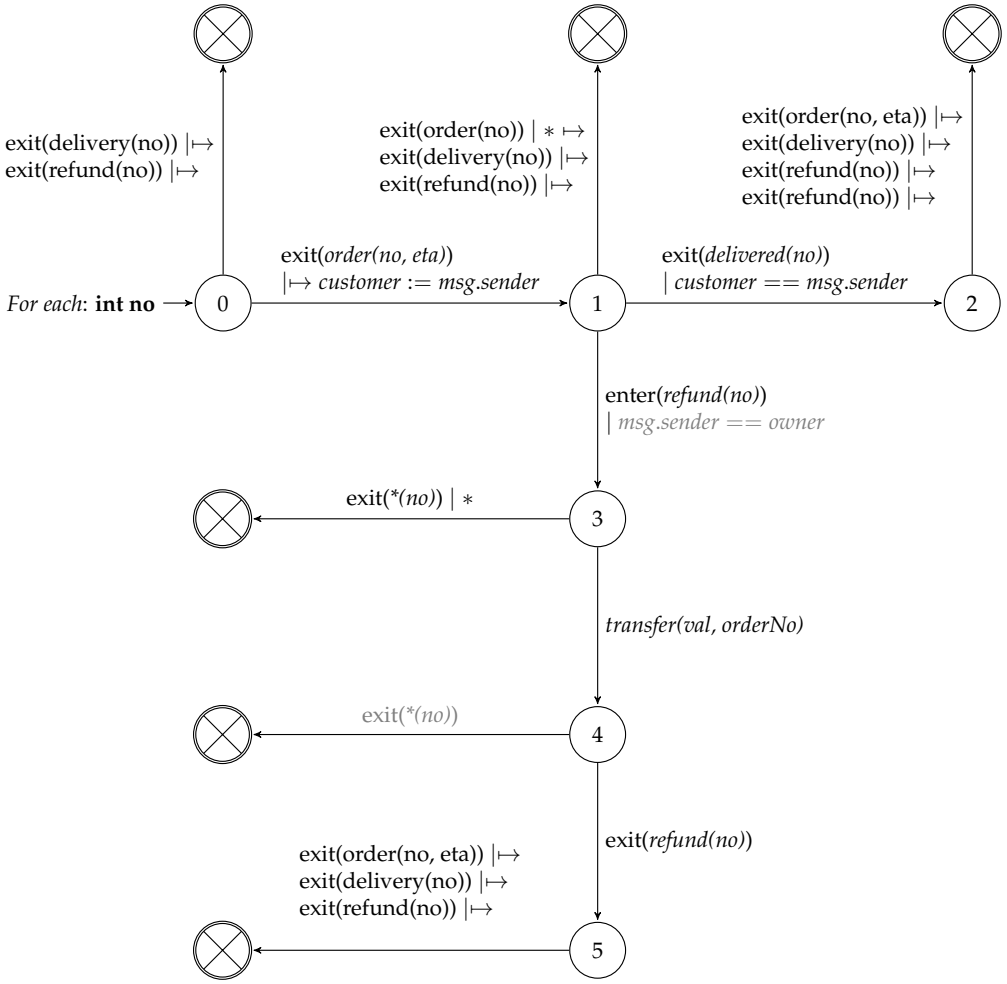


Figure 9.8: Courier service specification using property local state.

discouraged by security concerns and by gas constraints.

The results of performing this residual analysis can be seen in the first row of Table 9.6 and Table 9.7. The deployment costs overheads are significantly reduced, by around half in both compliant version of the implementation (shown in Listing. 9.1) and a violating version of the case study that does not check the message sender on `delivery` being called. However the difference in transaction costs is only minimal. Although the increased costs with monitoring are proportionally significant (around 26% at most) the cost in gas, they are not very expensive in real-world terms given the trend in real-world value associated with gas units.

Another pertinent aspect of this use case is the way the DEA is formulated. An equivalent formulation is the DEA in Figure 9.8, where instead of monitoring

Specification	Program Version	Original Specification		Residual Specification	
		Added Deployment Costs		Added Deployment Costs	
		Gas	% of Original	Gas	% of Original
Courier Interface Well-Use v1	Compliant	845465	197.00	380036	88.55
	Violating	845330	202.42	459295	109.98
Courier Interface Well-Use v2	Compliant	770050	179.46	445107	103.73
	Violating	782856	187.46	537377	128.68
Coupled Token Additions and Reductions	Compliant	287083	52.18	0	0
	Violating	287019	52.28	214390	39.05
Coupled Token and Ether State	Compliant	744011	113.03	0	0
	Violating	744075	113.26	296279	45.10

Table 9.6: Solidity case studies added deployment costs of original specification versus the residual specification.

for the value of the smart contract variable `customer[no]`, the customer identifier is saved to the monitoring state upon order (in a variable *customer*), and reused during the monitor's lifetime. Using intraprocedural analysis we cannot create a residual for this alternate formulation that is as small as for the original formulation, compare Figure 9.7 and Figure 9.8. The issue here is that the *customer* variable becomes divorced from the smart contract's variable state once it is saved to the property state, preventing intraprocedural analysis to exploit knowledge about the smart contract variable state to prove monitor guards. The residual analysis performed still reduces to an extent deployment costs, as can be seen in the second row of Table 9.6, but not transaction costs as can be seen in Table 9.7.

A possible way to reconnect these is to perform the same variable state abstraction for DEAs, and performing interprocedural analysis. The former requires applying the variable abstraction to DEAs, while the latter requires creating a flat whole-program CFA. By performing the analysis on this CFA with the DEA states tagged with assertions, we would be able to prune more of the DEA, potentially even proving it. Note an interprocedural would be required here to allow us to conclude that `customer`in the DEA corresponds to `customer[no]`in the smart contract, and that the value of `customer[no]`is invariant and thus that checking for `customer[no] == msg.sender`in the smart contract is equivalent to checking for `customer == msg.sender`. Here we did not attempt this, preferring to focus on intraprocedural analysis that is agnostic of the rest of the program.

9.2.2.2 Wallets

A common use case for the Ethereum blockchain is token wallets. These associate with each user an amount of tokens, allowing the users to transfer these tokens between them. Since these tokens can have real-life value then the well-behaviour

Specification	Program Version	Original Spec				Residual Spec			
		Added Transaction Costs				Added Transaction Costs			
		Gas		% of Original		Gas		% of Original	
		Least	Most	Least	Most	Least	Most	Least	Most
Courier Interface Well-Use v1	Compliant	7453	22107	11.80	26.53	6387	20977	10.12	25.17
	Violating	7453	22107	11.80	42.21	6387	21041	10.12	39.60
Courier Interface Well-Use v2	Compliant	7453	26357	11.64	31.65	6715	26357	10.63	31.65
	Violating	7453	41385	11.80	49.70	6743	41385	10.68	49.70
Coupled Token Additions and Reductions	Compliant	0	4108	0	8.82	0	0	0	0
	Violating	0	4108	0	8.82	0	3249	0	6.97
Coupled Token and Ether State	Compliant	0	8013	0	13.50	0	0	0	0
	Violating	0	8013	0	13.50	0	4657	0	7.85

Table 9.7: Solidity case studies added transaction costs of original specification versus the residual specification.

of these tokens is essential. One important property is that once an amount of tokens is removed from the balance of a user it is added to another user's balance, that is the reduction of an amount of tokens is always coupled with an addition of tokens. If this property does not hold we may have the case that a user may have an increasing amount of tokens, causing a reduction in value of the token, or that tokens may disappear from circulation causing scarcity of the token. This property is captured by the property in Figure 9.9, which requires that tokens removed from a user must be given to another user before the execution ends, and vice-versa that tokens given to a user must come from another user.

Our control-flow analysis here is not be able to prove the whole property, but it negates a possible permutation of the coupled events, reducing the property accordingly. In our case the implementation first adds the tokens to a user's balance and then removes it from the sender's balance, allowing us to remove state 2 in Figure 9.9, and transitions using it. Using variable abstraction we can also prove the rest of the property, since we can detect that the transaction is only successful when the sender has enough balance. We also consider a different version of such a smart contract, one that maintains a token balance associated with some users, corresponding to the amount of *ether* (the native currency used by Ethereum) held by the smart contract for them, for which we are also able to prove the result when compliant.

We also consider violating versions of these case studies where the token value exchanged may not be reduced upon a transfer. Our residual analysis thus still can remove one of the permutations, and remove state 2 from Figure 9.9, however it is not able to prove the rest of the property (since it is false).

The results associated with each of these wallet case studies can be seen in the bottom two rows of Table 9.6 and Table 9.7. As can be seen, in the compliant versions we are able to prove the whole property statically, while in the violating

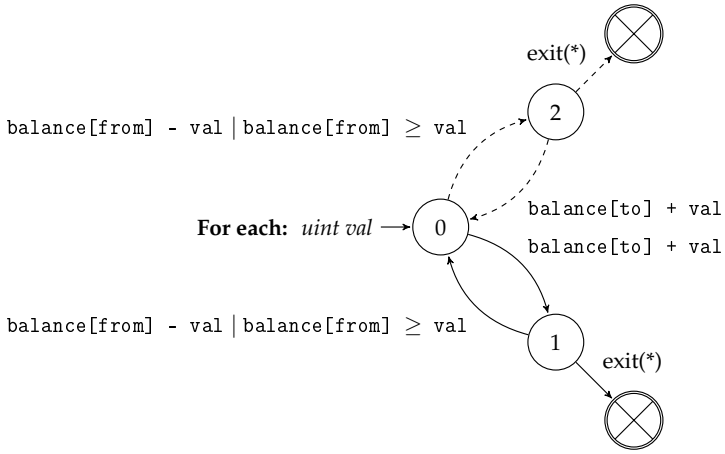


Figure 9.9: If a user is given an amount of tokens then the same amount must be reduced from another user, and vice-versa.

versions we are able to reduce the overheads significantly.

With these case studies we see how adding the notion of data to the residual analysis of DEAs can be useful, in that we can evaluate some guards statically which allows us to avoid some expensive computation at runtime. We also so how the lack of interprocedural analysis can hamper the effectiveness of the approach when DATEs use their local state.

9.3 Conclusions

In this chapter we have evaluated the presented residual analysis techniques against several case studies in both the Java and the Solidity languages. We have presented the results of case studies in relation to the difference in runtime overheads (time taken for Java, and gas for Solidity) between monitoring with the original and the residual property.

We can conclude that there are cases that control-flow analysis can be useful without any data abstraction, however these cases are limited since not all aspects of interest are encoded in the control-flow of a program. We saw how adding the notion of data to the analysis can help make inroads in places where pure control-flow analysis fails by adding the possibility of evaluating statically some property transition guards. There are however evident limitations with the intraprocedural approach we took, including that it does not allow us to properly treat DEA variable state.

10

Discussion

We have presented techniques to partially verify DEAs against programs that can be abstracted by CFAs. In this chapter we discuss several aspects of this work.

10.1 Partial Order of Verdicts

In this part we have presented techniques that consider the verification problem statically by pruning transitions from a property. Ignoring event reductions, in effect, here instead of a verification technique that produces a verdict from the set $\{\top, \bot, ?\}$ we are creating a technique that produces a verdict in terms of a partial order of properties.

Consider a program P and a property π, and an attempt to show $P \vdash \pi$. In the case that we cannot determine satisfaction we are instead producing a residual of π. In this part we have been using lockstep equivalence between properties (see Definition. 7.3.5) as the correctness condition for residuals. This condition induces an equivalence class of properties, however it does not capture the general idea that monitoring for a property reduction is 'easier' (i.e. requires less computation at runtime). For this we can instead consider the notion of the property sub-structure relation (see Definition. 7.1.6). These two relations together induce a partial order of properties, which can be used as the verdict set for our residual static analysis: $\mathit{Verdicts}_{\pi}^{P} \stackrel{\text{def}}{=} \{\pi' \mid \pi \equiv_P \pi' \wedge \pi' \sqsubseteq \pi\}$. Note that this sub-structure relation does not capture the guard reduction we defined in Definition. 8.2.11, however it should be clear that we can extend the sub-structure relation to include it, which we do not do here for simplicity.

Note then how showing compliance reduces to showing that $\pi_\top$ (the DEA that only contains the initial state and no transitions, and thus accepts every program as satisfying) is in $\mathit{Verdicts}^P_\pi$, while showing violation reduces to showing that $\pi_\top$ is not in this set.

10.2 Property Variable State

Although our main motivation was to deal with automaton-based properties with guarded events, we have also considered the analyses in the presence of a property maintaining some variable state. We have seen that some properties expressible with using property variable state (Figure 9.8) can be expressed without (Figure 9.7). In this case a specification without variable state is preferable, since it does not require using further memory. However there are properties that require this variable state, for example consider that a property may want to compare the size of a list at different points in time, which requires storing the earliest value. Maintaining variable state then allows us to specify properties about the program that can relate together different variable states of the program at different points in time.

10.3 Comparison with Existing Work

In Chapter 6 we discussed approaches with a similar motivation: performing some static analysis to reduce what must be proven at runtime. Here we compare them with our contributions, and consider how they are complementary to our work.

Bodden [2009] tool, CLARA, considers properties as finite-state automata, with the main novelty being the reduction of instrumentation that does not affect when and what verdict is given at runtime. This is similar to our own conditions for an appropriate residual analysis. Here we have slight differences in that the finite-state monitors considered by CLARA do not have explicit accepting states and they allow for multiple violations (i.e. a bad state is not a sink state). However the approach can still be applied to DEAs by analysing the abstract monitored system in much of the same way. Bodden [2009] do consider reducing a property by removing transitions with events that do not occur in a program, however our approach is much richer in that we consider more sophisticated reductions that consider the full control-flow of a method. A similarity is that CLARA apply their analyses at the intraprocedural level, also with some interprocedural information about which events can occur in which methods.

Dwyer and Purandare [2007] summarise deterministic regions in the program that cannot violate and that have one possible property state entry-point and one possible property state exit-point. Unlike our approach this is performed at the interprocedural level, performing a top-down traversal of the program source code to identify these regions. This is different from our approach in both purpose and the approach used, since we use intraprocedural analysis and attempt to prove a property rather than summarise instrumentation.

Here we have chosen to focus on reducing properties rather than instrumentation, although we do illustrate how abstract monitored systems can be used to also reduce instrumentation in Definition. 8.2.7. These approaches can both be applied to DEAs in a similar manner, however it is not clear these approaches do in the presence of guarded events, although variable abstractions can possibly be applied to make the analysis more precise.

The work of Chimento et al. [2015] is more in line with our work, where the residual corresponds to a reduction of the original ppDATE, where the Hoare triples associated with states of a DATE are removed or reduced. This is also complementary to our own work, where our techniques can be used to attack the control-flow of DATEs, and StaRVOOrs used to reduce the triples. Moreover, the theorem prover used by StaRVOOrs, `KeY`, can be further exploited to attempt to prove or reduced guards associated with entry or exit into a method.

Looking at the symbolic aspect of our work, in literature we find related approaches. Francalanza [2017] symbolically analyse monitors represented as transition systems with labels as guarded events to detect for *contrallability*. In other work similar approaches are used by Aceto et al. [2018b] to synthesize enforcing monitors for μHML properties.

10.4 DEA Extensions

The DEAs we have been using here are a core version of richer specification languages. Here we consider how our analysis extends to these richer languages. We consider Colombo et al. [2008]'s *dynamic automata with timers and events* (DATEs), Barringer et al. [2012]'s *quantified event automata* (QEAs), and Chimento et al. [2015]'s *pre- and post-condition DATEs* (ppDATEs).

One extension of DEAs is to add *typestate*. Throughout our examples have used this notion, but we did not formalise it for simplicity (although we implement typestate analysis in Clarva). In effect typestate entails that the events at runtime are also associated with a certain typestate object (or sets of objects). This is implemented by parameterising a DEA by some free variables and binding them at runtime, possibly replicating a DEA for each possible binding. These variables may be quantified over universally (as done by DATEs and QEAs) and

also existentially (as done by QEAs). Following the approach of Bodden [2009] a pointer analysis can be used to abstract the possible variable bindings statically, producing relations that relate instrumentation points in the program that may be associated with the same monitor instance at runtime. Our analysis techniques can be easily projected onto each such static approximation of a typestate object, producing a residual for each such abstract object. A monitor at runtime can then monitor for the union of these residuals safely. Typestate static analysis for QEAs has been proposed by Reger [2016], in attempt to apply Bodden [2009]'s approach, while they consider briefly that analysing guarded events requires some program variable abstraction. Here we have considered this formally and proposed concrete techniques to this end.

Events in this work have been limited to points in the program. DATEs consider *timing* events, where a clock may signal a certain time by triggering an appropriate event. In our approach this can be handled by extending our CFA abstraction with a looping transition for each state for each relevant timing event, modeling the possibility that a timing event occurs at any point during the execution of a program. DATEs also allow for multiple monitors that can communicate with each other across channels, modeled through channel *receive* events. This can be handled by synchronously composing the communicating DATEs and applying the analysis on the composition, projecting results onto the original single DATEs.

A syntactic extension of DATEs are ppDATEs, that augment DATEs by associating sets of Hoare triples with explicit states. These can be reduced into DATEs, as shown by Chimento et al. [2015], and thus our approach extended to DATEs as described above projects onto ppDATEs easily.

10.5 Analysis is Harder than Verification

Consider that analysis is a different activity from verification, where analysis attempts to identify facts about the program while verification attempts to confirm a fact about the program. Cousot et al. [2018] contrast verification and analysis in terms of computational complexity, showing they are equivalent when the property space is finite, while for an infinite property space analysis is harder than verification. This is shown by showing reductions between verifiers and analysers in the finite case, and showing there is no reduction from a non-trivial verifier to an analyser in the infinite-case.

Cousot et al. [2018]'s results have no bearing on the work here, since we are dealing with transformations from analysers to verifiers (i.e. we attempt to use analysis to show a verification problem, and when failing we produce a

residual problem), Cousot et al. [2018] instead presents an impossibility result about translations of certain verifiers to analysers.

10.6 Limitations and Future Work

The static analysis we present is limited in at least two ways: (i) it is performed intraprocedurally; and (ii) the data-flow techniques we apply to create state invariants are quite shallow (since we discard conditions upon meeting a statement that can effect their truth value), both limiting the precision of the results. Our purpose here was not to create precise analyses however, instead our purpose here was to both create inexpensive analyses that can be used as part of the instrumentation process and to create a formal framework which can be used to reason about residual analysis of dynamic event automata and variants thereof.

Here we have not attempted to evaluate the appropriateness of performing these techniques as part of the instrumentation process, but instead focused on validating the applicability of the analyses to reduce properties. To evaluate this in the future our prototype tools must be merged with the instrumentation process, allowing for the time taken to instrument a program without the analysis to be compared to the time taken to analyse and instrument the program. We expect that the control-flow analysis will not present substantial addition of time taken, however we expect that querying an SMT solver to prune an abstract monitored system to be relatively expensive.

The benefits of an intraprocedural analysis is that analysis can be performed piecemeal and independently for each method, while the results can be reused for any high-level use of the procedures since in effect they create function summaries. However we have not compared the expense of interprocedural analysis against intraprocedural analysis. One way to implement interprocedural analysis here would be to use a depth-first traversal of a CFA while using function summaries, in a similar way to Dwyer and Purandare [2007].

Moreover, the techniques we presented are only sound and do not attempt to show that a program violates a property, since our main purpose was to produce residuals. However the artifacts we produce can be analysed for this purpose, where if we manage to show that there is a path in the abstract monitored system that only uses concrete transitions towards a violation then we can conclude that there may be a concrete violation at runtime modulo the branching conditions in the path.

A reduction in instrumentation, and a reduction in a property can have an effect on the overheads at runtime. Both entail that less property transitions are checked for triggering at runtime. This checking can be a significant overhead, since given a triggered event e the e-transitions outgoing from the current monitor

state must be iterated over and their guards evaluated. In the worst-case scenario all the outgoing transitions are evaluated, while in the best-case scenario the first transition checked matches. It is not clear if the ordering of this checking can benefit from some type of static analysis, however we do not consider that here.

Here we have been considering only monitorable properties. There are however other interesting properties. Future work can focus on trying to prove the non-monitorable parts of a property using static analysis, while leaving the monitorable parts for runtime. The issue here is how to identify and extract any non-monitorable parts of a property. There already is work towards this, for example, Alpern and Schneider [1987] show how to decompose Büchi automata into safety and liveness properties.

11

Conclusions

In this part we have motivated the use of static analysis techniques to perform residual analysis of dynamic event automata, i.e. a form of extended finite-state machines with a symbolic variable state and transitions with events guarded by predicates on the program and property variable state, and actions on the property variable state.

We further have given a formal framework for residual analysis of DEAs, by considering when program and property reductions are equivalent with respect to each other both at a high-level of general program satisfaction, and at a lower level of being equivalent in lockstep at the low-level of the verdicts given to program states. We have presented approaches to residual analysis that are used to reduce the required program event instrumentation and to produce a reduced property that is however equivalent to the original with respect to the program. The analyses we presented are intraprocedural and act piecewise on the program, while they may exploit an SMT solver to identify when property transitions cannot match a certain program execution step.

We have evaluated this approach finding moderate overhead reductions both in the case of Java and Solidity programs. Future works to improve these results will focus on evaluating the tradeoff between scalability and precision of interprocedural analysis, and the further development and improvement of prototypes that currently only partially implement the presented analyses.

Conclusions

Verification attempts can fail without leaving a result. In this thesis we have identified a trend in formal verification communities to deal with such failure not by simply failing with an unknown verdict, but with a transformation of the input problem to another simpler problem. In this manner different verifiers can work in concert to partially or gradually verify compliance, each making steps useful towards solving the problem. We have formalised this by giving an abstract semantics to properties in terms of the programs they satisfy. This allows us to characterise the notion of a residual property in terms of whether it is equivalent to the original property in the context of what is known about the program under verification. Through an appropriate property quotienting operator, as we showed for both state- and event-based property formalisms, we can then reduce a property modulo a certain level of precision, allowing subsequent techniques to tackle the remaining proof obligation. The work we surveyed also considers program transformations which are not captured by this abstract framework. In future we can consider the effectiveness of including a similar abstract semantics for programs in terms of the properties they satisfy, and to characterise a quotienting operator that allows us to transform a program with respect to the property that remains to be proven.

A specific interesting application to this is in combining static and runtime verification. This combination allows one to prove as much as possible pre-deployment whilst leaving any residuals for runtime. Such an approach allows us to resolve known parts of the program statically, while focusing runtime verification on more dynamic aspects of the program that cannot be dealt easily using static analysis. In the process we then are able to provide some static

guarantees (increasing confidence in the program) and reduce overheads due to both instrumentation and monitor computations.

We validated this approach more concretely for state-based properties with a case study involving an industrial project that necessitated the use of the partial approach to verification motivated here. The context was of a payments ecosystem, serving as a server with which external applications could communicate with to provide payment services to their customers. In this case study we showed how in a context where runtime verification is a necessity one can exploit knowledge about what is being enforced at runtime to provide static guarantees of a program. In fact, given a developer-provided assured model of the behaviour of an application being enforced post-deployment, we showed how a regulation property (as a universally quantified proposition specified in a controlled natural language) can be verified or partially evaluated from this model.

For event-based properties we have considered the residual analysis of dynamic event automata against a control-flow representation of a program (control-flow automata). Both of the representations we use maintain some symbolic state representing a variable state and allow branching based on guards that also transform the variable state. The difference is that DEAs act as event listeners and CFAs act as event producers, with the latter then being appropriately instrumented to produce events during execution for the DEA to give its verdict. We have defined techniques to both reduce event instrumentation associated with monitoring for DEAs, and reductions to a DEA itself, allowing any subsequent analyses to exploit the simpler framing of the input verification problem. Our treatment of this approach was technology-agnostic (with respect to a programming language), and in fact we have implemented this work for both Java and Solidity, and validated its utility through a number of case studies. Concretely, we have used an intraprocedural analysis of a program to abstract runtime executions in terms of the behaviour explicit in methods. The approach we take is limited, since intraprocedural information is over-approximated very coarsely, while the data-flow analysis we consider only propagates assertions up to statements that can affect them. Subsequent work in this direction can focus on introducing static analyses that produce a finer representation of a program, perhaps using a form of interprocedural analysis and introducing some context-sensitivity with respect to assertions flowing through a program. Nevertheless, the less precise static analysis we use can be used as a first pass in a workflow of such analyses, limiting the state space that more precise and resource-intensive analyses have to explore.

References

Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. Monitoring for silent actions. In Satya V. Lokam and R. Ramanujam, editors, *37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2017, December 11-15, 2017, Kanpur, India*, volume 93 of *LIPIcs*, pages 7:1–7:14. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.FSTTCS.2017.7. URL `https://doi.org/10.4230/LIPIcs.FSTTCS.2017.7`.

Luca Aceto, Antonis Achilleos, Adrian Francalanza, and Anna Ingólfsdóttir. A framework for parameterized monitorability. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 203–220, Cham, 2018a. Springer International Publishing. ISBN 978-3-319-89366-2.

Luca Aceto, Ian Cassar, Adrian Francalanza, and Anna Ingólfsdóttir. On runtime enforcement via suppressions. In Sven Schewe and Lijun Zhang, editors, *29th International Conference on Concurrency Theory, CONCUR 2018, September 4-7, 2018, Beijing, China*, volume 118 of *LIPIcs*, pages 34:1–34:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2018b. doi: 10.4230/LIPIcs.CONCUR.2018.34. URL `https://doi.org/10.4230/LIPIcs.CONCUR.2018.34`.

Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. Adventures in monitorability: From branching to linear time and back again. *Proc. ACM Program. Lang.*, 3(POPL), January 2019a. doi: 10.1145/3290365. URL `https://doi.org/10.1145/3290365`.

Luca Aceto, Antonis Achilleos, Adrian Francalanza, Anna Ingólfsdóttir, and Karoliina Lehtinen. An operational guide to monitorability. In Peter Csaba Ölveczky and Gwen Salaün, editors, *Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings*, volume 11724 of *Lecture Notes in Computer Science*, pages 433–453. Springer, 2019b. doi: 10.1007/978-3-030-30446-1_23. URL `https://doi.org/10.1007/978-3-030-30446-1_23`.

Wolfgang Ahrendt, Bernhard Beckert, Richard Bubel, Reiner Hähnle, Peter H. Schmitt, and Mattias Ulbrich, editors. *Deductive Software Verification - The KeY Book - From Theory to Practice*,

volume 10001 of *Lecture Notes in Computer Science*. Springer, 2016. ISBN 978-3-319-49811-9. doi: 10.1007/978-3-319-49812-6. URL http://dx.doi.org/10.1007/978-3-319-49812-6.

Irem Aktug and Katsiaryna Naliuka. Conspec – a formal language for policy specification. *Electron. Notes Theor. Comput. Sci.*, 197(1):45–58, February 2008. ISSN 1571-0661. doi: 10.1016/j.entcs.2007.10.013. URL http://dx.doi.org/10.1016/j.entcs.2007.10.013.

V. S. Alagar and K. Periyasamy. *Extended Finite State Machine*, pages 105–128. Springer London, London, 2011. ISBN 978-0-85729-277-3. doi: 10.1007/978-0-85729-277-3_7. URL https://doi.org/10.1007/978-0-85729-277-3_7.

Bowen Alpern and Fred B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2 (3):117–126, Sep 1987. ISSN 1432-0452. doi: 10.1007/BF01782772. URL https://doi.org/10.1007/BF01782772.

H. R. Andersen. Partial model checking. In *Proceedings of Tenth Annual IEEE Symposium on Logic in Computer Science*, pages 398–407, Jun 1995. doi: 10.1109/LICS.1995.523274.

Henrik Reif Andersen. Model checking and boolean graphs. In Bernd Krieg-Brückner, editor, *ESOP '92*, pages 1–19, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg. ISBN 978-3-540-46803-5.

A. Arnold, A. Vincent, and I. Walukiewicz. Games for synthesis of controllers with partial observation. *Theor. Comput. Sci.*, 303(1):7–34, June 2003. ISSN 0304-3975. doi: 10.1016/S0304-3975(02)00442-5. URL http://dx.doi.org/10.1016/S0304-3975(02)00442-5.

Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. A model-based approach to combining static and dynamic verification techniques. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*, volume 9952 of *Lecture Notes in Computer Science*, pages 416–430, 2016a. ISBN 978-3-319-47165-5. doi: 10.1007/978-3-319-47166-2_29. URL https://doi.org/10.1007/978-3-319-47166-2_29.

Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. Residual control-flow static analysis with symbolic automata. In *CSAW 2016: Computer Science Annual Workshop 2016, University of Malta, Malta*, 2016b.

Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. Regulation specification and automatic static and dynamic checks generation in the ope. In *CSAW 2016: Computer Science Annual Workshop 2016, University of Malta, Malta*, 2016c.

Shaun Azzopardi, Christian Colombo, Gordon J. Pace, and Brian Vella. Compliance checking in the open payments ecosystem. In Rocco De Nicola and Eva Kühn, editors, *Software Engineering and Formal Methods*, pages 337–343, Cham, 2016d. Springer International Publishing. ISBN 978-3-319-41591-8.

Shaun Azzopardi, Christian Colombo, Jean-Paul Ebejer, Edward Mallia, and Gordon J. Pace. Runtime verification using VALOUR. In *RV-CuBES 2017. An International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools, September 15, 2017, Seattle, WA, USA*, pages 10–18, 2017a. URL http://www.easychair.org/publications/paper/wbSB.

Shaun Azzopardi, Christian Colombo, and Gordon Pace. Control-Flow Residual Analysis for Symbolic Automata. Technical report, 2017b. URL `https://www.um.edu.mt/ict/cs/research/technical_reports`.

Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. Control-flow residual analysis for symbolic automata. In Adrian Francalanza and Gordon J. Pace, editors, *Proceedings Second International Workshop on Pre- and Post-Deployment Verification Techniques, Torino, Italy, 19 September 2017*, volume 254 of *Electronic Proceedings in Theoretical Computer Science*, pages 29–43. Open Publishing Association, 2017c. doi: 10.4204/EPTCS.254.3.

Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. A controlled natural language for financial services compliance checking. In Brian Davis, C. Maria Keet, and Adam Wyner, editors, *Controlled Natural Language - Proceedings of the Sixth International Workshop, CNL 2018, Maynooth, Co. Kildare, Ireland, August 27-28, 2018*, volume 304 of *Frontiers in Artificial Intelligence and Applications*, pages 11–20. IOS Press, 2018a. ISBN 978-1-61499-903-4. doi: 10.3233/978-1-61499-904-1-11. URL `https://doi.org/10.3233/978-1-61499-904-1-11`.

Shaun Azzopardi, Joshua Ellul, and Gordon J. Pace. Monitoring smart contracts: CONTRACTLARVA and open challenges beyond. In *The 18th International Conference on Runtime Verification*, 2018b.

Shaun Azzopardi, Christian Colombo, and Gordon Pace. A technique for automata-based verification with residual reasoning. Technical Report CS-2019-02, Department of Computer Science, University of Malta, 2019. URL `https://www.um.edu.mt/ict/cs/ourresearch/technicalreports`.

Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. CLARVA: Model-based residual verification of java programs. In *Model-Driven Engineering and Software Development - 8th International Conference, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, 2020a.

Shaun Azzopardi, Christian Colombo, and Gordon J. Pace. A technique for automata-based verification with residual reasoning. In *Model-Driven Engineering and Software Development - 8th International Conference, MODELSWARD 2020, Valletta, Malta, February 25-27, 2020*, 2020b.

Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Comput. Surv.*, 51(3), May 2018. ISSN 0360-0300. doi: 10.1145/3182657. URL `https://doi.org/10.1145/3182657`.

Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018. ISBN 978-3-319-10575-8. doi: 10.1007/978-3-319-10575-8_11. URL `https://doi.org/10.1007/978-3-319-10575-8_11`.

Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

Howard Barringer, David Rydeheard, and Klaus Havelund. Rule systems for run-time monitoring: From eagle to ruler. In Oleg Sokolsky and Serdar Taşıran, editors, *Runtime Verification*, pages 111–125, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-77395-5.

Howard Barringer, Yliès Falcone, Klaus Havelund, Giles Reger, and David Rydeheard. Quantified event automata: Towards expressive and efficient runtime monitors. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 68–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32759-9.

Ezio Bartocci, Radu Grosu, Atul Karmarkar, Scott A. Smolka, Scott D. Stoller, Erez Zadok, and Justin Seyster. Adaptive runtime verification. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, pages 168–182, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-35632-2.

Ezio Bartocci, Yliès Falcone, Borzoo Bonakdarpour, Christian Colombo, Normann Decker, Klaus Havelund, Yogi Joshi, Felix Klaedtke, Reed Milewicz, Giles Reger, Grigore Rosu, Julien Signoles, Daniel Thoma, Eugen Zalinescu, and Yi Zhang. First international competition on runtime verification: rules, benchmarks, tools, and final results of crv 2014. *International Journal on Software Tools for Technology Transfer*, 21(1):31–70, Feb 2019. ISSN 1433-2787. doi: 10.1007/s10009-017-0454-5. URL https://doi.org/10.1007/s10009-017-0454-5.

P. Baudin, J.C. Filliâtre, T. Hubert, C. Marché, B. Monate, Y. Moy, and V. Prevosto. Acsl :ansi/iso c specification language. http://frama-c.cea.fr/acsl.html, 2011. [Online; accessed 10-April-2019].

Andreas Bauer, Martin Leucker, and Christian Schallhart. Comparing ltl semantics for runtime verification. *J. Log. and Comput.*, 20(3):651–674, June 2010. ISSN 0955-792X. doi: 10.1093/logcom/exn075. URL http://dx.doi.org/10.1093/logcom/exn075.

Andreas Bauer, Martin Leucker, and Christian Schallhart. Runtime verification for ltl and tltl. *ACM Trans. Softw. Eng. Methodol.*, 20(4):14:1–14:64, September 2011. ISSN 1049-331X. doi: 10.1145/2000799.2000800. URL http://doi.acm.org/10.1145/2000799.2000800.

Dirk Beyer. Partial verification and intermediate results as a solution to combine automatic and interactive verification techniques. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, pages 874–880, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47166-2.

Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification*, pages 504–518, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73368-3.

Dirk Beyer, Thomas A. Henzinger, M. Erkan Keremoglu, and Philipp Wendler. Conditional model checking: A technique to pass information between verifiers. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 57:1–57:11, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1614-9. doi: 10.1145/2393596.2393664. URL http://doi.acm.org/10.1145/2393596.2393664.

Dirk Beyer, Sumit Gulwani, and David A. Schmidt. *Combining Model Checking and Data-Flow Analysis*, pages 493–540. Springer International Publishing, Cham, 2018a. ISBN 978-3-319-10575-8. doi: 10.1007/978-3-319-10575-8_16. URL https://doi.org/10.1007/978-3-319-10575-8_16.

Dirk Beyer, Marie-Christine Jakobs, Thomas Lemberger, and Heike Wehrheim. Reducer-based construction of conditional verifiers. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 1182–1193, New York, NY, USA, 2018b. ACM. ISBN 978-1-4503-5638-1. doi: 10.1145/3180155.3180259. URL `http://doi.acm.org/10.1145/3180155.3180259`.

Eric Bodden. *Verifying finite-state properties of large-scale programs*. PhD thesis, McGill University, 2009.

Eric Bodden, Patrick Lam, and Laurie Hendren. Clara: A framework for partially evaluating finite-state runtime monitors ahead of time. In Howard Barringer, Ylies Falcone, Bernd Finkbeiner, Klaus Havelund, Insup Lee, Gordon Pace, Grigore Roşu, Oleg Sokolsky, and Nikolai Tillmann, editors, *Runtime Verification*, pages 183–197, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-16612-9.

Borzoo Bonakdarpour, Cesar Sanchez, and Gerardo Schneider. Monitoring hyperproperties by combining static analysis and runtime verification. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Verification*, pages 8–27, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03421-4.

Muffy Calder and Carron Shankland. A symbolic semantics and bisimulation for full lotos. In Myungchul Kim, Byoungmoon Chin, Sungwon Kang, and Danhyung Lee, editors, *Formal Techniques for Networked and Distributed Systems*, pages 185–200, Boston, MA, 2001. Springer US. ISBN 978-0-306-47003-5.

Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, PLDI '91, pages 278–292, New York, NY, USA, 1991. ACM. ISBN 0-89791-428-7. doi: 10.1145/113445.113469. URL `http://doi.acm.org/10.1145/113445.113469`.

Omar Chebaro, Nikolai Kosmatov, Alain Giorgetti, and Jacques Julliand. Program slicing enhances a verification technique combining static and dynamic analysis. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1284–1291, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0857-1. doi: 10.1145/2245276.2231980. URL `http://doi.acm.org/10.1145/2245276.2231980`.

Jesús Mauricio Chimento, Wolfgang Ahrendt, Gordon J. Pace, and Gerardo Schneider. Starvoors : A tool for combined static and runtime verification of java. In Ezio Bartocci and Rupak Majumdar, editors, *Runtime Verification*, pages 297–305, Cham, 2015. Springer International Publishing. ISBN 978-3-319-23820-3.

Jong-Deok Choi, Keunwoo Lee, Alexey Loginov, Robert O'Callahan, Vivek Sarkar, and Manu Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. *SIGPLAN Not.*, 37(5):258–269, May 2002. ISSN 0362-1340. doi: 10.1145/543552.512560. URL `http://doi.acm.org/10.1145/543552.512560`.

Maria Christakis and Valentin Wüstholz. Bounded abstract interpretation. In Xavier Rival, editor, *Static Analysis*, pages 105–125, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. ISBN 978-3-662-53413-7.

Maria Christakis, Peter Müller, and Valentin Wüstholz. Collaborative verification and testing with explicit assumptions. In Dimitra Giannakopoulou and Dominique Méry, editors, *FM 2012: Formal Methods*, pages 132–146, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32759-9.

Thomas Colcombet and Pascal Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, page 54–66, New York, NY, USA, 2000. Association for Computing Machinery. ISBN 1581131259. doi: 10.1145/325694.325703. URL https://doi.org/10.1145/325694.325703.

Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Dynamic event-based runtime monitoring of real-time and contextual properties. In *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, pages 135–149, 2008. doi: 10.1007/978-3-642-03240-0_13. URL https://doi.org/10.1007/978-3-642-03240-0_13.

Christian Colombo, Gordon J. Pace, and Gerardo Schneider. Larva — safer monitoring of real-time java programs (tool paper). In *Proceedings of the 2009 Seventh IEEE International Conference on Software Engineering and Formal Methods*, SEFM '09, pages 33–37, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3870-9. doi: 10.1109/SEFM.2009.13. URL https://doi.org/10.1109/SEFM.2009.13.

Loïc Correnson and Julien Signoles. Combining analyses for c program verification. In Mariëlle Stoelinga and Ralf Pinger, editors, *Formal Methods for Industrial Critical Systems*, pages 108–130, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-32469-7.

Gabriele Costa, David Basin, Chiara Bodei, Pierpaolo Degano, and Letterio Galletta. From natural projection to partial model checking and back. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 344–361, Cham, 2018. Springer International Publishing. ISBN 978-3-319-89960-2.

P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

Patrick Cousot, Roberto Giacobazzi, and Francesco Ranzato. Program analysis is harder than verification: A computability perspective. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification*, pages 75–95, Cham, 2018. Springer International Publishing. ISBN 978-3-319-96142-2.

Mike Czech, Marie-Christine Jakobs, and Heike Wehrheim. Just test what you cannot verify! In Alexander Egyed and Ina Schaefer, editors, *Fundamental Approaches to Software Engineering*, pages 100–114, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-46675-9.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

N. Dragoni, F. Massacci, K. Naliuka, and I. Siahaan. Security-by-contract: Toward a semantics for digital signatures on mobile code. In Javier Lopez, Pierangela Samarati, and Josep L. Ferrer, editors, *Public Key Infrastructure*, pages 297–312, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-73408-6.

Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis exploiting static analysis: Results to reformulate and reduce the cost of dynamic analysis. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 124–133, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: 10.1145/1321631.1321651. URL http://doi.acm.org/10.1145/1321631.1321651.

Matthew B. Dwyer and Rahul Purandare. Residual checking of safety properties. In Klaus Havelund, Rupak Majumdar, and Jens Palsberg, editors, *Model Checking Software*, pages 1–2, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-85114-1.

Arvind Easwaran, Sampath Kannan, and Oleg Sokolsky. Steering of discrete event systems: Control theory approach. *Electronic Notes in Theoretical Computer Science*, 144(4): 21 – 39, 2006. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2005.02.066. URL http://www.sciencedirect.com/science/article/pii/S157106610600301X. Proceedings of the Fifth Workshop on Runtime Verification (RV 2005).

Joshua Ellul and Gordon J. Pace. Runtime verification of ethereum smart contracts. In *14th European Dependable Computing Conference, EDCC 2018, Iaşi, Romania, September 10-14, 2018*, pages 158–163. IEEE Computer Society, 2018. ISBN 978-1-5386-8060-5. doi: 10.1109/EDCC.2018.00036. URL https://doi.org/10.1109/EDCC.2018.00036.

Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In Bernhard Beckert and Claude Marché, editors, *Formal Verification of Object-Oriented Software*, pages 10–30, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-18070-5.

Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 2103–2110, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-639-7. doi: 10.1145/1774088.1774531. URL http://doi.acm.org/10.1145/1774088.1774531.

Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *STTT*, 14(3):349–382, 2012a. doi: 10.1007/s10009-011-0196-8. URL https://doi.org/10.1007/s10009-011-0196-8.

Yliès Falcone, Jean-Claude Fernandez, and Laurent Mounier. What can you verify and enforce at runtime? *International Journal on Software Tools for Technology Transfer*, 14(3):349–382, Jun 2012b. ISSN 1433-2787. doi: 10.1007/s10009-011-0196-8. URL https://doi.org/10.1007/s10009-011-0196-8.

Stephen J. Fink, Eran Yahav, Nurit Dor, G. , and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2):9:1–9:34, May 2008. ISSN 1049-331X. doi: 10.1145/1348250.1348255. URL http://doi.acm.org/10.1145/1348250.1348255.

Maria João Frade and Jorge Sousa Pinto. Verification conditions for source-level imperative programs. *Computer Science Review*, 5(3):252 – 277, 2011. ISSN 1574-0137. doi: https://doi.org/10.1016/j.cosrev.2011.02.002. URL http://www.sciencedirect.com/science/article/pii/S1574013711000037.

Adrian Francalanza. Consistently-detecting monitors. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 8:1–8:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.CONCUR.2017.8. URL https://doi.org/10.4230/LIPIcs.CONCUR.2017.8.

Adrian Francalanza, Luca Aceto, Antonis Achilleos, Duncan Paul Attard, Ian Cassar, Dario Della Monica, and Anna Ingólfsdóttir. A foundation for runtime monitoring. In Shuvendu Lahiri and Giles Reger, editors, *Runtime Verification*, pages 8–29, Cham, 2017a. Springer International Publishing. ISBN 978-3-319-67531-2.

Adrian Francalanza, Luca Aceto, and Anna Ingólfsdóttir. Monitorability for the hennessy-milner logic with recursion. *Formal Methods in System Design*, 51(1):87–116, 2017b. doi: 10.1007/s10703-017-0273-z. URL https://doi.org/10.1007/s10703-017-0273-z.

Carlo A. Furia, Bertrand Meyer, and Sergey Velder. A survey of loop invariants. *CoRR*, abs/1211.4470, 2012. URL http://arxiv.org/abs/1211.4470.

Mihaela Gheorghiu Bobaru, Corina S. Păsăreanu, and Dimitra Giannakopoulou. Automated assume-guarantee reasoning by abstraction refinement. In Aarti Gupta and Sharad Malik, editors, *Computer Aided Verification*, pages 135–148, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70545-1.

Dimitra Giannakopoulou and Klaus Havelund. Runtime analysis of linear temporal logic specifications. Technical report, 2001.

Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Heaps don't lie: Countering unsoundness with heap snapshots. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017. doi: 10.1145/3133892. URL https://doi.org/10.1145/3133892.

Neville Grech, George Fourtounis, Adrian Francalanza, and Yannis Smaragdakis. Shooting from the heap: Ultra-scalable static analysis with heap snapshots. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 198–208, New York, NY, USA, 2018. Association for Computing Machinery. ISBN 9781450356992. doi: 10.1145/3213846.3213860. URL https://doi.org/10.1145/3213846.3213860.

Simon Jantsch, David Müller, Christel Baier, and Joachim Klein. From ltl to unambiguous büchi automata via disambiguation of alternating automata. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods – The Next 30 Years*, pages 262–279, Cham, 2019. Springer International Publishing. ISBN 978-3-030-30942-8.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993. ISBN 0-13-020249-5.

Johannes Kanig, Rod Chapman, Cyrille Comar, Jerôme Guitton, Yannick Moy, and Emyr Rees. Explicit assumptions - a prenup for marrying static and dynamic program verification. In Martina Seidl and Nikolai Tillmann, editors, *Tests and Proofs*, pages 142–157, Cham, 2014. Springer International Publishing. ISBN 978-3-319-09099-3.

Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-42206-4. URL http://dl.acm.org/citation.cfm?id=646158.680006.

Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c: A software analysis perspective. *Formal Aspects of Computing*, 27(3):573–609, May 2015. ISSN 1433-299X. doi: 10.1007/s00165-014-0326-7. URL https://doi.org/10.1007/s00165-014-0326-7.

S. Kleene. Introduction to Metamathematics, 1952.

Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2):6:1–6:34, February 2010. ISSN 0164-0925. doi: 10.1145/1667048.1667051. URL http://doi.acm.org/10.1145/1667048.1667051.

Tobias Kuhn. A survey and classification of controlled natural languages. *Comput. Linguist.*, 40 (1):121–170, March 2014. ISSN 0891-2017. doi: 10.1162/COLI_a_00168. URL http://dx.doi.org/10.1162/COLI_a_00168.

Akash Lal, Nicholas Kidd, Thomas Reps, and Tayssir Touili. Abstract error projection. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis*, pages 200–217, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-74061-2.

Leslie Lamport and Martin Abadi. Decomposing specifications of concurrent systems. pages 327–340, August 1994. URL https://www.microsoft.com/en-us/research/publication/decomposing-specifications-concurrent-systems/.

Martin Leucker. Sliding between model checking and runtime verification. In Shaz Qadeer and Serdar Tasiran, editors, *Runtime Verification*, pages 82–87, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-35632-2.

Lars Luthmann, Stephan Mennicke, and Malte Lochau. Compositionality, decompositionality and refinement in input/output conformance testing. In Olga Kouchnarenko and Ramtin Khosravi, editors, *Formal Aspects of Component Software*, pages 54–72, Cham, 2017. Springer International Publishing. ISBN 978-3-319-57666-4.

Fabio Martinelli. Symbolic partial model checking for security analysis. In Vladimir Gorodetsky, Leonard Popyack, and Victor Skormin, editors, *Computer Network Security*, pages 122–134, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. ISBN 978-3-540-45215-7.

Erik Meijer and Peter Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages. In *OOPSLA'04 Workshop on Revival of Dynamic Languages*, 01 2004.

Samaneh Navabpour, Chun Wah Wallace Wu, Borzoo Bonakdarpour, and Sebastian Fischmeister. Efficient techniques for near-optimal instrumentation in time-triggered runtime verification. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 208–222, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-29859-2. doi: 10.1007/978-3-642-29860-8_16. URL http://dx.doi.org/10.1007/978-3-642-29860-8_16.

Samaneh Navabpour, Yogi Joshi, Wallace Wu, Shay Berkovich, Ramy Medhat, Borzoo Bonakdarpour, and Sebastian Fischmeister. Rithm: A tool for enabling time-triggered runtime verification for c programs. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, page 603–606, New York, NY, USA, 2013. Association for Computing Machinery. ISBN 9781450322379. doi: 10.1145/2491411.2494596. URL https://doi.org/10.1145/2491411.2494596.

Neda Noroozi, Mohammad Reza Mousavi, and Tim A. C. Willemse. Decomposability in input output conformance testing. In *Proceedings Eighth Workshop on Model-Based Testing, MBT 2013, Rome, Italy, 17th March 2013.*, pages 51–66, 2013. doi: 10.4204/EPTCS.111.5. URL https://doi.org/10.4204/EPTCS.111.5.

Doron Peled. Partial order reduction: Model-checking using representatives. In Wojciech Penczek and Andrzej Szałas, editors, *Mathematical Foundations of Computer Science 1996*, pages 93–112, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-70597-0.

A. Pnueli and A. Zaks. Psl model checking and run-time verification via testers. In Jayadev Misra, Tobias Nipkow, and Emil Sekerinski, editors, *FM 2006: Formal Methods*, pages 573–586, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-37216-5.

Rahul Purandare, Matthew B. Dwyer, and Sebastian Elbaum. Monitoring finite state properties: Algorithmic approaches and their relative strengths. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 381–395, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-29860-8.

Jean-Baptiste Raclet. Residual for component specifications. *Electronic Notes in Theoretical Computer Science*, 215:93 – 110, 2008. ISSN 1571-0661. doi: https://doi.org/10.1016/j.entcs.2008.06.023. URL http://www.sciencedirect.com/science/article/pii/S1571066108003666. Proceedings of the 4th International Workshop on Formal Aspects of Component Software (FACS 2007).

P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, 77(1):81–98, Jan 1989. ISSN 0018-9219. doi: 10.1109/5.21072.

Giles Reger. Considering typestate verification for quantified event automata. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques*, pages 479–495, Cham, 2016. Springer International Publishing. ISBN 978-3-319-47166-2.

Giles Reger, Sylvain Hallé, and Yliès Falcone. Third International Competition on Runtime Verification CRV 2016. In *Sixteenth International Conference on Runtime Verification*, Madrid, Spain, September 2016. URL https://hal.inria.fr/hal-01428834.

A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *2010 23rd IEEE Computer Security Foundations Symposium*, pages 186–199, July 2010. doi: 10.1109/CSF.2010.20.

R. Sekar, V.N. Venkatakrishnan, Samik Basu, Sandeep Bhatkar, and Daniel C. DuVarney. Model-carrying code: A practical approach for safe execution of untrusted applications. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 15–28, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945448. URL http://doi.acm.org/10.1145/945445.945448.

Jeremy G. Siek. Gradual typing for functional languages. In *In Scheme and Functional Programming Workshop*, pages 81–92, 2006.

Scott D. Stoller, Ezio Bartocci, Justin Seyster, Radu Grosu, Klaus Havelund, Scott A. Smolka, and Erez Zadok. Runtime verification with state estimation. In Sarfraz Khurshid and Koushik Sen, editors, *Runtime Verification*, pages 193–207, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. ISBN 978-3-642-29860-8.

Sandro Stucki, César Sánchez, Gerardo Schneider, and Borzoo Bonakdarpour. Gray-box monitoring of hyperproperties. In Maurice H. ter Beek, Annabelle McIver, and José N. Oliveira, editors, *Formal Methods - The Next 30 Years - Third World Congress, FM 2019, Porto, Portugal, October 7-11, 2019, Proceedings*, volume 11800 of *Lecture Notes in Computer Science*, pages 406–424. Springer, 2019. doi: 10.1007/978-3-030-30942-8_25. URL https://doi.org/10.1007/978-3-030-30942-8_25.

Li Sui, Jens Dietrich, Michael Emery, Shawn Rasheed, and Amjed Tahir. On the soundness of call graph construction in the presence of dynamic language features - a benchmark and tool evaluation. In Sukyoung Ryu, editor, *Programming Languages and Systems*, pages 69–88, Cham, 2018. Springer International Publishing. ISBN 978-3-030-02768-1.

Satish Thatte. Quasi-static typing. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 367–381, New York, NY, USA, 1990. ACM. ISBN 0-89791-343-4. doi: 10.1145/96709.96747. URL http://doi.acm.org/10.1145/96709.96747.

Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3:121–189, 1995.

Tomás E. Uribe. Combinations of model checking and theorem proving. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems*, pages 151–170, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. ISBN 978-3-540-46421-1.

Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999. URL http://dl.acm.org/citation.cfm?id=781995.782008.

Moshe Y. Vardi. Automata-theoretic model checking revisited. In Byron Cook and Andreas Podelski, editors, *Verification, Model Checking, and Abstract Interpretation*, pages 137–150, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-69738-1.

Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, Apr 2003. ISSN 1573-7535. doi: 10.1023/A:1022920129859. URL https://doi.org/10.1023/A:1022920129859.

Gavin Wood. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper*, 151:1–32, 2014.

Chun Wah Wallace Wu. *Methods for Reducing Monitoring Overhead in Runtime Verification*. Master thesis, University of Waterloo, Waterloo, Ontario, Canada, 2013.

Y. Zhou and Y. Zhang. A logical study of partial entailment. *J. Artif. Intell. Res.*, 40:25–56, 2011. doi: 10.1613/jair.3117. URL https://doi.org/10.1613/jair.3117.