Department of Computer Science

Symbolic Methods for Transducers and Testing

Olli Saarikivi

Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 255/2017

Symbolic Methods for Transducers and Testing

Olli Saarikivi

A doctoral dissertation completed for the degree of Doctor of Science (Technology) to be defended, with the permission of the Aalto University School of Science, at a public examination held at the lecture hall T2 of the school on 16th February 2018 at 12:00.

Aalto University
School of Science
Department of Computer Science

Supervising professor
Assoc. Prof. Keijo Heljanko, Aalto University, Finland

Thesis advisor
Assoc. Prof. Keijo Heljanko, Aalto University, Finland

Preliminary examiners
Prof. Parosh Aziz Abdulla, Uppsala University, Sweden
Prof. Dr. Jaco van de Pol, University of Twente, The Netherlands

Opponent
Adj. Prof. Rupak Majumdar, University of California, Los Angeles, USA; Scientific Director, Max Planck Institute for Software Systems, Germany

Aalto University publication series
DOCTORAL DISSERTATIONS 255/2017

© 2017 Olli Saarikivi

ISBN 978-952-60-7786-4 (printed)
ISBN 978-952-60-7787-1 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)
http://urn.fi/URN:ISBN:978-952-60-7787-1

Unigrafia Oy
Helsinki 2017

Finland

NORDIC ECOLABEL
441 697
Printed matter

A! Aalto University

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Abstract

Author
Olli Saarikivi

Name of the doctoral dissertation
Symbolic Methods for Transducers and Testing

Publisher School of Science

Unit Department of Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 255/2017

Field of research Computer Science

Manuscript submitted 9 October 2017 **Date of the defence** 16 February 2018

Permission to publish granted (date) 5 December 2017 **Language** English

☐ **Monograph** ☒ **Article dissertation** ☐ **Essay dissertation**

Abstract

Symbolic methods reason about groups of values. The evolution of modern satisfiability modulo theories (SMT) solvers has enabled an increasing variety of symbolic applications that require efficient reasoning in rich logics, such as bit vectors and arrays. SMT solvers remove barriers for employing solver technology in the development of symbolic methods. This dissertation contributes applications of symbolic methods in the areas of stream processing, test generation and software verification.

Symbolic automata and transducers are generalizations of their respective classical models that shift reasoning about inputs to an SMT solver, which allows efficient use of very large alphabets. In the areas of stream processing and Big Data analytics, it is desirable to be able to represent computations as pipelines of modular processing stages. Pipelines represented as symbolic transducers can be efficiently fused into a single stage, which reduces communication overhead and enables further reductions.

This dissertation presents a tool that applies fusion to pipelines of symbolic transducers specified in a variety of frontend languages. Fusion is supported by further reductions based on reachability analysis and automata minimization. The tool generates efficient fused code, which is shown to provide comparable performance to a sample of real-world, hand-optimized, monolithic code, as well as greater performance than alternative methods of composing modular pipelines of stream computations.

Concolic testing is a popular approach for implementing symbolic execution, which commonly uses SMT solvers to efficiently generate test inputs. The Dash algorithm combines concolic testing with a predicate abstraction. This dissertation presents an implementation of Dash for automatically verifying sequential C programs on the LLVM compiler framework. A refinement strategy for the LLVM intermediate representation is presented and design considerations for reducing memory usage and improving verification performance are discussed. The resulting tool LCTD shows competitive results on a set of benchmarks drawn from SV-COMP.

Partial order reduction is an approach for exploring reduced sets of interleavings between threads in a multi-threaded program. This dissertation presents an improvement to the dynamic partial order reduction (DPOR) algorithm, which exploits the commutativity of read operations to enable more reduction. The new algorithm DPOR-CR is shown to explore significantly fewer tests. Furthermore, a proof is presented that data races can be cheaply checked for during DPOR.

Keywords symbolic methods, symbolic transducers, stream processing, stream fusion, test generation, software verification, partial order reduction, race detection

ISBN (printed) 978-952-60-7786-4 **ISBN (pdf)** 978-952-60-7787-1

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Helsinki **Location of printing** Helsinki **Year** 2017

Pages 187 **urn** http://urn.fi/URN:ISBN:978-952-60-7787-1

Aalto-yliopisto

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tiivistelmä

Tekijä
Olli Saarikivi

Väitöskirjan nimi
Symboliset metodit muuntimille ja testaukseen

Julkaisija Perustieteiden korkeakoulu

Yksikkö Tietotekniikan laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 255/2017

Tutkimusala Tietotekniikka

Käsikirjoituksen pvm 09.10.2017 **Väitöspäivä** 16.02.2018

Julkaisuluvan myöntämispäivä 05.12.2017 **Kieli** Englanti

☐ **Monografia** ☒ **Artikkeliväitöskirja** ☐ **Esseeväitöskirja**

Tiivistelmä

Symboliset metodit päättelevät joukoilla arvoja. Modernien satisfiability modulo theories (SMT) -ratkaisimien evoluutio on mahdollistanut kasvavan valikoiman symbolisia sovelluksia, jotka vaativat päättelyä rikkailla logiikoilla, kuten bittivektoreilla ja taulukoilla. SMT-ratkaisimet helpottavat ratkaisinteknologian käyttöä symbolisten metodien kehityksessä. Tämä väitöskirja edistää symbolisten metodien sovelluksia tietovirtojen käsittelyssä, testien generoinnissa ja ohjelmistojen verifioinnissa.

Symboliset automaatit ja muuntimet ovat vastaavien klassisten mallien yleistyksiä, jotka siirtävät syötteisiin liittyvää päättelyä SMT-ratkaisimelle, mikä mahdollistaa erittäin suurten aakkostojen tehokkaan käytön. Tietovirtojen käsittelyssä ja suuren datan analytiikassa on hyödyllistä pystyä esittämään laskentaa sarjana modulaarisia vaiheita. Sarjana symbolisia muuntimia esitetty ohjelma on mahdollista tehokkaasti yhdistää yhdeksi vaiheeksi, mikä vähentää ylimääräistä kommunikaatiota. Tätä kutsutaan fuusioksi. Lisäksi näin yhdistettyjä muuntimia voidaan edelleen supistaa.

Tämä väitöskirja esittelee työkalun, joka soveltaa fuusiota erinäisillä ohjelmointikielillä esitettyjen symbolisien muuntimien sarjoista koostuviin ohjelmiin. Fuusiota täydentävät saavutettavuusanalyysiin ja automaattien minimointiin perustuvat supistusmenetelmät. Työkalu tuottaa tehokasta koodia, jonka näytetään olevan suorituskyvyltään vertailukelpoista näytteeseen käsin optimoitua ei-modulaarista koodia, sekä tehokkaampaa kuin vaihtoehtoisilla menetelmillä fuusioidut ohjelmat.

Dynaaminen symbolinen suoritus käyttää SMT-ratkaisimia generoidakseen tehokkaasti syötteitä testaukseen. Dash-algoritmi yhdistää dynaamisen symbolisen suorituksen ja predikaattiabstraktion. Tämä väitöskirja esittelee LLVM kääntäjäkirjastoa käyttävän toteutuksen Dash-algoritmista C-ohjelmien automaattista verifiointia varten. Tämä sisältää selityksen siitä, miten Dash-algoritmi toteutetaan LLVM:n välikielelle, sekä tavan parantaa verifioinnin suorituskykyä. Toteutettu työkalu LCTD on kilpailukykyinen vertailtuna joukolla SV-COMP-kilpailusta haettuja ohjelmia.

Osittaisjärjestysreduktio on lähestymistapa, jolla voidaan käydä läpi supistettuja joukkoja rinnakkaisten ohjelmien testisuorituksia menettämättä virheiden saavutettavuutta. Tämä väitöskirja esittelee uuden version dynaamisesta osittaisjärjestysreduktiosta (DPOR), joka mahdollistaa lukuoperaatioiden kommutatiivisuuden hyödyntämisen. Uuden algoritmin DPOR-CR:n näytetään käyvän läpi huomattavasti vähemmän testisuorituksia kuin alkuperäinen. Lisäksi todistetaan, että synkronoimaton jaetun muistin käyttö voidaan tarkistaa edullisesti DPOR:n aikana.

Avainsanat symboliset metodit, symboliset muuntimet, tietovirtojen prosessointi, tietovirtojen fuusio, testigenerointi, ohjelmistojen verifiointi, osittaisjärjestysreduktio

ISBN (painettu) 978-952-60-7786-4 **ISBN (pdf)** 978-952-60-7787-1

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Helsinki **Painopaikka** Helsinki **Vuosi** 2017

Sivumäärä 187 **urn** http://urn.fi/URN:ISBN:978-952-60-7787-1

Preface

I started out as a research assistant at Aalto University after Assoc. Prof. Keijo Heljanko invited me to a late interview despite a spam filtering mishap on my part. Enchanted by research, I soon switched to Aalto University for my Master's studies. Since the completion of my Master's thesis under Assoc. Prof. Keijo Heljanko's supervision I've been working with him as a doctoral student at the Department of Computer Science. During my doctoral studies I also interned twice at the Microsoft Research Redmond lab under the mentorship of Margus Veanes, Todd Mytkowicz and Madan Musuvathi.

This dissertation presents research I have done both at the Department of Computer Science at Aalto University and at the Microsoft Research Redmond lab. The research has been funded by the LIME project joint with Tekes (Finnish Funding Agency for Technology and Innovation), the SARANA project in the SAFIR 2014 program, the Academy of Finland projects 139402 and 277522, the Nokia Foundation, the Emil Aaltonen Foundation and the industrial partner Conformiq.

I am grateful to Assoc. Prof. Keijo Heljanko for his invaluable guidance and continuous support during my doctoral research. I would also like to thank everyone I collaborated with at Aalto University. I am especially grateful to Kari Kähkönen for his guidance (including instructing my Master's thesis) and collaboration in the development of test generation tools.

I am also grateful to Margus Veanes, Todd Mytkowicz and Madan Musuvathi for providing me amazing internship experiences at Microsoft Research, which contributed greatly to my development as a researcher. The continued collaboration with Margus Veanes has been invaluable for completing this dissertation.

I want to thank Prof. Parosh Aziz Abdulla (Uppsala University, Sweden) and Prof. Dr. Jaco van de Pol (University of Twente, The Netherlands) for agreeing to act as pre-examiners for this dissertation and Adj. Prof. Rupak Majumdar (University of California, Los Angeles, USA; Scientific Director, Max Planck

Institute for Software Systems, Germany) for agreeing to act as an opponent in the defense of this dissertation.

Finally, I want to thank my family for all their support during my doctoral studies and Natasha for seeing me through the home stretch.

Helsinki, December 15, 2017,

Olli Saarikivi

Contents

Preface i

Contents iii

List of Publications v

Author's Contribution vii

List of Abbreviations ix

1. Introduction 1

1.1 Contributions 7

1.2 The Structure of the Dissertation 8

2. Symbolic Transducers for Stream Processing 11

2.1 Symbolic Transducers and Automata 13

2.2 Fusion 17

2.3 Reductions 22

2.3.1 Reachability Based Branch Elimination 22

2.3.2 Control State Reduction 27

2.4 Frontends 32

2.5 Implementation 34

2.6 Evaluations 35

2.7 Related Work 36

3. Automated Testing and Verification 39

3.1 Concolic Testing 40

3.2 Abstraction Refinement with Concolic Testing 42

3.2.1 Evaluations 44

3.3 Dynamic Partial Order Reduction with Concolic Testing 45

3.3.1 Evaluation 48

3.4 Related Work 48

4. Conclusions **51**

References **55**

Errata **63**

Publications **65**

List of Publications

This dissertation consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Olli Saarikivi, Margus Veanes, Todd Mytkowicz, Madan Musuvathi. Fusing effectful comprehensions. In *Proceedings of the 38th Conference on Programming Language Design and Implementation (PLDI 2017)*, pages 17–32, https://doi.org/10.1145/3062341.3062362, June 2017.

II Olli Saarikivi, Margus Veanes. Minimization of symbolic transducers. In *Proceedings of the 29th International Conference on Computer Aided Verification (CAV 2017), Part II*, LNCS 10427, pages 176–196, https://doi.org/10.1007/978-3-319-63390-9_10, July 2017.

III Olli Saarikivi, Margus Veanes. Translating C# to branching symbolic transducers. In *21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR-21), Short Presentations*, pages 86–99, May 2017.

IV Olli Saarikivi, Keijo Heljanko. LCTD: test-guided proofs for C programs on LLVM. *Journal of Logic and Algebraic Methods in Programming (JLAMP)*, 85(6), pages 1292–1317, https://doi.org/10.1016/j.jlamp.2015.10.010, October 2016.

V Olli Saarikivi, Kari Kähkönen, Keijo Heljanko. Improving dynamic partial order reductions for concolic testing. In *Proceedings of the 12th International Conference on Application of Concurrency to System Design (ACSD 2012)*, pages 132–141, https://doi.org/10.1109/ACSD.2012.18, June 2012.

VI Olli Saarikivi, Keijo Heljanko. Reporting races in dynamic partial order reduction. In *Proceedings of NASA Formal Methods - 7th International*

Symposium (NFM 2015), LNCS 9058, pages 450–456, https://doi.org/10.1007/978-3-319-17524-9_35, April 2015.

The author of this dissertation has also co-authored the following publications, which are not included in this dissertation.

Olli Saarikivi, Hernán Ponce de León, Kari Kähkönen, Keijo Heljanko, Javier Esparza. Minimizing test suites with unfoldings of multithreaded programs. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(2), https://doi.org/10.1145/3012281, February 2017.

Olli Saarikivi, Keijo Heljanko. LCTD: Tests-guided proofs for C programs on LLVM (competition contribution). In *Proceedings of the Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*, LNCS 9636, pages 927–929, https://doi.org/10.1007/978-3-662-49674-9_62, April 2016.

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko. Unfolding based automated testing of multithreaded programs. *Automated Software Engineering*, 22(4), pages 475–515, https://doi.org/10.1007/s10515-014-0150-6, December 2015.

Hernán Ponce de León, Olli Saarikivi, Kari Kähkönen, Keijo Heljanko, Javier Esparza. Unfolding based minimal test suites for testing multithreaded programs. In *Proceedings of the 15th International Conference on Application of Concurrency to System Design (ACSD 2015)*, pages 40–49, https://doi.org/10.1109/ACSD.2015.12, June 2015.

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko. LCT: A parallel distributed testing tool for multithreaded Java programs. In *Proceedings of the 11th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2012)*, ENTCS 296, pages 253–259, https://doi.org/10.1016/j.entcs.2013.09.002, August 2013.

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko. Using unfoldings in automated testing of multithreaded programs. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, pages 150–159, https://doi.org/10.1145/2351676.2351698, September 2012.

Kari Kähkönen, Tuomas Launiainen, Olli Saarikivi, Janne Kauttio, Keijo Heljanko, Ilkka Niemelä. LCT: An open source concolic testing tool for Java programs. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2011)*, pages 75–80, March 2011.

Author's Contribution

Publication I: "Fusing effectful comprehensions"

The ideas for this publication are joint work between the authors. The theory concerning fusion of symbolic transducers and the ideas for reachability based branch elimination are joint work between the author of this dissertation and Margus Veanes. The author of this dissertation is responsible for the design and implementation of the frontend and backend of the tool developed, and the evaluation. The implementations of fusion and reachability based branch elimination were contributed by Margus Veanes. The manuscript was written as joint work between the authors.

Publication II: "Minimization of symbolic transducers"

The author of this dissertation is responsible for the idea that minimizing an overapproximation of a symbolic transducer allows safe control state reduction, the idea of strengthening transitions with state invariants to allow more reduction, and the construction of finite symbolic transducers for prefix codes. The initial idea of generalizing quasi-determinization to the symbolic setting was suggested by Margus Veanes, the subsequent algorithm was designed by both authors in collaboration, and the final concrete algorithm was implemented by the thesis author. The author of this dissertation is responsible for the implementation and experiments, while the manuscript was written as joint work between the authors.

Publication III: "Translating C# to branching symbolic transducers"

The author of this dissertation is responsible for the design and implementation of the translation from C# to branching symbolic transducers. Margus Veanes designed and implemented the state exploration algorithm. The manuscript was written as joint work between the authors, with the authors taking responsibility for the sections describing their respective contributions.

Publication IV: "LCTD: test-guided proofs for C programs on LLVM"

The author of this dissertation is the main contributor for this publication. Keijo Heljanko provided guidance, supervision and comments to the manuscript.

Publication V: "Improving dynamic partial order reductions for concolic testing"

The idea of modifying dynamic partial order reduction (DPOR) to exploit commutativity of reads is joint work. Otherwise the author of this dissertation is responsible for the contributions of this publication. Kari Kähkönen provided assistance in the implementation, which is a modification of an existing tool by him, as well as guidance and comments to the manuscript. Keijo Heljanko provided guidance, supervision and comments to the manuscript.

Publication VI: "Reporting races in dynamic partial order reduction"

The author of this dissertation is the main contributor for this publication. Keijo Heljanko provided guidance, supervision and comments to the manuscript.

List of Abbreviations

BDD binary decision diagram. 1, 31

BST branching symbolic transducer. 7–9, 12, 13, 15, 17, 18, 20–24, 32–34, 37, 52, 53

CSR control state reduction. 12, 27–29, 31, 32, 36, 37, 51

Dash is a software verification algorithm that combines test generation with abstraction refinement. 6, 8, 9, 24, 39, 40, 42–44, 52, 53

DPOR dynamic partial order reduction. 7–9, 39, 40, 45–48, 52, 53

DPOR-CR DPOR with commutative reads. 46–48, 52

LCT is a concolic testing tool for Java and C programs. 9, 39, 43, 47, 52

LCTD is a verification tool for sequential C programs on LLVM. 9, 39, 43, 44, 49, 52

LLVM is a collection of compiler and toolchain technologies. ix, 43, 52, 53

RBBE reachability based branch elimination. 7–9, 12, 13, 22, 24, 26, 27, 36, 44, 51–53

SAT the Boolean satisfiability problem. 1, 2, 49

SFA symbolic finite automaton. 7, 13, 14, 16, 27, 28, 31

SFT symbolic finite transducer. 7, 14, 17, 27–29, 31, 33, 34, 37

SMT satisfiability modulo theories. 2, 4–6, 9, 11, 13, 37, 41, 44, 49, 51

ST symbolic transducer. 7, 12, 14–17, 22, 23, 27–29, 37, 41, 52

1. Introduction

Symbolic methods reason about groups of values. For example, in software testing, techniques based on symbolic execution can reliably reach code that would be unlikely to be covered by a random test, such as tests generated by *blackbox fuzzing* [15]. In the realm of software verification, algorithms employ symbolic abstractions to provide proofs of safety, which would be tedious to produce by hand. Symbolic generalizations of classical automata and transducers result in more compact representations, while many classical algorithms, such as automata minimization, generalize to the symbolic case [31].

Symbolic execution for automatic testing has been described as early as 1976 in [53]. However, at that time an automated implementation was not feasible due to available constraint solvers and hardware not being sufficiently powerful. Subsequent research produced testing tools based on symbolic execution using various constraint solving methods [37]. The imprecision arising from the various abstractions employed in these tools was partly addressed by DART [41], which combines a linear integer programming based symbolic execution engine with concrete test execution to ensure that all reported errors are actual errors. This approach has since been referred to as both *dynamic symbolic execution* and *concolic testing*.

For hardware verification, binary decision diagram (BDD) based *symbolic model checking* tools, such as SMV [65], achieved a lot of success [21]. BDD based model checking has also been found to be amenable to parallelization [84]. In the search for scalability, *bounded model checking* trades completeness for bug finding ability [14]. A primary benefit of bounded model checking is that it encodes reachability queries as instances of the Boolean satisfiability (SAT) problem, for which modern solvers can often solve instances with hundreds of thousands of variables and millions of constraints.

SAT solvers are a good fit for verifying circuits, as the encodings into Boolean satisfiability are relatively straightforward. SAT based bounded model check-

```
output = input.Decode().Deserialize().SelectPrice()
  .FindPriceDips().Serialize().Encode();
```

Figure 1.1. A high-level view of a pipeline of data transformations

ers have also been developed for software [27, 51, 89], by encoding the richer datatypes found in software into SAT instances through *bit-blasting* [54].

Satisfiability modulo theories (SMT) solvers build on the efficiency of SAT solvers to provide built-in support for more expressive theories, such as arrays and bit vectors, which greatly simplifies the otherwise complex task of designing encodings. The expressiveness and efficiency offered by SMT solvers have made an increasing variety of applications for symbolic methods feasible. This dissertation contributes applications of symbolic methods in the areas of stream processing, test generation and software verification. In stream processing, the theory of symbolic transducers is applied to optimize pipelines of computation. Test generation and software verification use symbolic execution to systematically explore program behavior. The rest of this introduction further motivates these problems, starting from how symbolic transducers can be applied to the optimization of stream computations.

Optimizations and User Code

As a running example, consider the pipeline of data transformations in Figure 1.1. One would like to maximize the throughput of such pipelines without losing the modularity of the implementation. The input is a stream of bytes that is first decoded and deserialized into higher level objects, then a projection `SelectPrice` produces a stream of prices to which a query `FindPriceDips` is applied, and finally the results are serialized and encoded to produce the output as a stream of bytes. This style of processing is similar to the processing inside a single node of a data processing system [34, 91, 88, 6].

The core functionality of this pipeline is in the query `FindPriceDips`, which would use some form of pattern matching to find price drops followed by price increases. While `SelectPrice` and `FindPriceDips` could be combined, the modularity gained from keeping them separate may be desirable, as now `FindPriceDips` can be used unchanged in another pipeline.

In a Big Data analysis context where the amount of data exceeds main memory, `input` and `output` are necessarily streams of bytes from/to disk or network. For reasons of flexibility, robustness and compression, the wire format of `input` typically does not match the in-memory format used by `SelectPrice`, which

is, therefore, preceded by `Decode` and `Deserialize` for bridging this gap. Similarly, the outputs from `FindPriceDips` are serialized and encoded to form `output`.

Popular data processing frameworks [34, 91, 88, 6] implement transformations to and from these wire level formats, thus freeing users from having to implement them. However, these transformations can constitute a significant portion of all computation to the point that, contrary to popular belief, data analytics workloads often end up being CPU rather than I/O bound [73]. Two major sources of overhead contribute to the problem:

Communication Data to and from user code must be communicated using some streaming mechanism, which incurs overhead.

Unspecialized computation The transformations to and from the wire level format (serialization/deserialization) may be too general in the context of assumptions made and guarantees given by user code.

Various optimizations have been implemented in data processing frameworks to reduce these overheads, including: *predicate/projection pushdown* to drop data that will not be used as early as possible, *code generation* to specialize serialization [90], and *columnar data formats* to allow only relevant parts of data to be read [59, 66, 60]. These optimizations have preconditions that must be satisfied for them to be applicable. For example, a predicate can only be pushed over operations that are invariant under that predicate.

Both code generation of serializers and usage of columnar data formats require a structured data model, where input/output formats are declaratively specified. The three different user facing APIs in Spark [7, 30] provide concrete insight into the tradeoffs between unstructured and structured data models:

RDDs provide an unstructured data model and users can write arbitrary code to manipulate it.

DataFrames provide a semi-structured data model of untyped rows and users can write code that maps to queries understood by Spark's query optimizer.

DataSets extend DataFrames to a structured data model of typed rows.

While RDDs provide the most flexibility, programs using them are not analyzable by Spark's query optimizer and thus have limited applicable optimizations. The table based model imposed by DataFrames allows writing relational queries to which traditional query optimizations can be applied. When DataFrames are further limited to typed data in DataSets, Spark will use code generation techniques to lower the serialization overhead [90].

SCOPE is a Big Data analytics language that allows flexible use of user defined operations [23]. The SCOPE query optimizer does support projection pushdown over user defined operations, but requires users to annotate their code to inform which columns are actually used [74]. This is both a potential source of error, if users annotate a column used by the code as not required, and of inefficiency, if users fail to annotate a column, that is not used by the code, as such.

When optimizations either impose a rigid programming model, require user annotation or are only narrowly applicable, one general strategy to relax these limitations is to apply program analysis techniques to check the required preconditions.

Fusion

The individual stages of the running example pipeline in Figure 1.1 are *transducers*, by virtue of having one input stream and one output stream. Transducers are an automata theoretic model of computation for representing transformations over streams of input. While classical transducers assume finite and typically small alphabets, advances in SMT solving have made symbolic generalizations of transducers practical. For example, in the context of string sanitizers[1], symbolic transducers have been successfully used for verification of real world sanitizers [49].

Symbolic transducers have the attractive property that they can be composed [85] (Publication I). In the context of stream processing, composition is referred to as *fusion*, which is the name that is used in the rest of this dissertation. Looking from a different perspective, code generation in query compilation can be seen to perform a light-weight form of fusion [71]. In functional programming, *deforestation* performs a similar transformation that eliminates intermediate data structures [87]. Please refer to Section 2.7 for a detailed discussion of related work.

To illustrate fusion, if the stages `Deserialize` and `SelectPrice` in Figure 1.1 were represented as symbolic transducers, then fusing them would create a single transducer `DeserializeThenSelectPrice`. The resulting transducer has advantages over `Deserialize().SelectPrice()`, namely:

(1) omitting the intermediate data stream reduces communication overhead, and

(2) deserialization of columns that `SelectPrice` drops can be omitted, thus reducing overhead from unspecialized computation.

[1] String sanitizers are programs that remove unsafe patterns in untrusted input.

Advantage 2 is of particular interest, as it corresponds to the relational optimization of projection pushdown. While Advantage 1 is a direct consequence of fusion, Advantage 2, however, is only *enabled* by fusion and to realize it the fused transducer must be transformed with an appropriate reduction to remove the unnecessary computation. This situation is analogous to function inlining in compilers, where the inlining itself removes the function call overhead, but also enables further optimization in the context of the call site. Three forms of reduction have been explored in this dissertation:

Pruning removes transitions with unsatisfiable guards, which naturally arise during fusion (Publication I).

Reachability analysis can be used to remove transitions that are not triggered by any sequence of inputs (Publication I).

Control state reduction can be used to merge states that have equivalent behavior (Publication II).

These reductions were designed to eliminate or at least alleviate the increase in transducer size associated with fusion. All of these reductions use an SMT solver for efficiently reasoning about sets of values.

Testing and Verification

Reducing fused symbolic transducers using reachability analysis was inspired by similar analyses used in *software verification*, where the objective is to prove that no transition violates an *error property* (e.g., that no assertions fail).

Verification algorithms commonly use *abstraction* to efficiently handle large or even infinite state spaces. The abstraction must be some representation for the program in which checking the error property is cheap. For example, in *predicate abstraction* the state space of the original program is split using a set of predicates (Boolean valued formulas) into a smaller number of *regions*, inside of which all states are assumed to behave equivalently. Regardless of its type, an abstraction must have the property that any execution of the program must be found in the abstraction, i.e., the abstraction must be an *overapproximation*. This is in contrast with automated test generation, where an *underapproximation* of a program is explored in the form of a set of tests. Both under- and overapproximation are useful in program analysis due to different guarantees given on the program safety depending on whether there exists a counterexample, i.e., a behavior that violates the error property present in the model:

Model	No counterexample	Counterexample
Abstraction	Safe	Maybe unsafe
Tests	Maybe safe	Unsafe

Verification methods may benefit from combining abstraction and testing, which will in effect constrain the behavior of the program from "both sides" until either the reachability or unreachability of the error property is shown. Dash is one such algorithm, which combines a predicate abstraction with an automated test generation approach called *concolic testing* [9] (Publication IV).

Concolic testing [41, 78, 77] combines *conc*rete execution with symb*olic* execution. During a test execution with some specific, concrete inputs for every operation executed the symbolic execution records a formula over input variables that captures the semantics of the operation. These formulas are subsequently used to construct constraints for following previously unexplored paths. An SMT solver is used to solve these path constraints, which yields concrete inputs for subsequent tests. One advantage of concolic testers is that they can tolerate code for which symbolic execution is not supported. This can happen, for example, if the symbolic execution is implemented through instrumentation and the program-under-test calls into uninstrumented code. In these situations the concolic tester can gracefully degrade into for example random testing [41].

Concolic testing has been successful for finding bugs in real-world software [42]. However, concolic testing only addresses generating tests for single-threaded programs. Adding basic support for testing multi-threaded programs is straightforward: concolic testing explores a tree of executions in which branches correspond to input-dependent branches in the program, and the interleavings of a multi-threaded program also form a tree where branches correspond to scheduling decisions. These trees can be interleaved during execution.

The complexity in test generation for multi-threaded programs, however, is in handling the combinatorial explosion. In a multi-threaded program, the execution of local operations in one thread does not interfere with that of local operations in another thread. These kinds of relationships between operations in different threads can be formalized into an *independence relation*. When testing multi-threaded programs, exploring multiple interleavings of operations that only differ on the ordering of independent operations is typically redundant, as these interleavings will encounter the same deadlocks and violate the same assertions [38]. As exploring all interleavings is typically infeasible, the challenge then is to minimize the number of redundant interleavings explored.

Test generation methods that try to avoid exploring such equivalent interleavings are called *partial order reduction* methods. A class of these methods that

only require the histories of operations that have been executed are known as *dynamic* partial order reduction methods. This dissertation improves on the original dynamic partial order reduction (DPOR) algorithm [38] with a more accurate independence relation for read operations (Publication V). While DPOR was originally designed to find all deadlocks and assertion errors, this dissertation also shows that DPOR can be retrofitted for race detection with minimal modifications (Publication VI).

1.1 Contributions

The contributions made in the publications in this dissertation are as follows. For the specific contributions made by the author of this dissertation please see the section "Author's Contribution" in the front matter.

Publication I: "Fusing effectful comprehensions" An algorithm and tool for fusing compositions of branching symbolic transducers (BSTs) is presented. To complement the satisfiability based branch elimination built into the fusion algorithm, a branch elimination algorithm based on reachability analysis is presented. For specifying BSTs, frontends from C#, regex and XPath are presented. The code generated from the fused BSTs shows significant speedups over reasonable hand-written code.

Publication II: "Minimization of symbolic transducers" A theory of control state reduction of symbolic transducers (STs) through an over-approximating encoding to symbolic finite automatons (SFAs) is presented. Quasi-determinization is generalized to the symbolic setting and it is shown that for the special case of deterministic symbolic finite transducers (SFTs) quasi-determinization followed by control state reduction provides a minimal SFT. The control state reduction approach is shown effective on a varied set of STs obtained as fusions of stream processing pipelines, especially ones including Huffman decoders, for which an SFT construction is provided.

Publication III: "Translating C# to branching symbolic transducers" A tool and method for translating imperative C# code to BSTs is presented. The tool supports native C# datatypes and arbitrary control flow, allowing for convenient and succinct specification of transducers as C# functions. To support subsequent algorithms that leverage control state, such as reachability based branch elimination (RBBE), an algorithm for exploring Boolean registers into control state is presented.

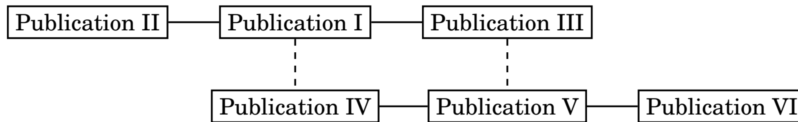


Figure 1.2. Topics in this dissertation and their relations.

Publication IV: "LCTD: test-guided proofs for C programs on LLVM" LCTD, an open source tool implementing the Dash algorithm [9] for verifying sequential C programs is presented. An extension for array support is described. An implementation strategy that allows execution traces to only store pointer values instead of whole program state and avoids evaluation of abstract regions' predicates is presented.

Publication V: "Improving dynamic partial order reductions for concolic testing" An extension of the DPOR algorithm to exploit the commutativity of read operations for further partial order reduction is presented and implemented in an open source tool that combines DPOR with concolic testing for automated testing of multi-threaded Java programs. Implementation strategies for DPOR and sleep sets in a distributed setting are discussed.

Publication VI: "Reporting races in dynamic partial order reduction" A way to retrofit the DPOR algorithm for race detection is presented, and the resulting algorithm is shown to be precise and of negligible overhead over that of DPOR itself. The algorithm is shown to be sound for the C++11 memory model and a large subset of Java.

1.2 The Structure of the Dissertation

Figure 1.2 presents the connections between the publications in this dissertation. Direct dependencies are shown as solid lines, while dashed lines represent publications sharing similar techniques. These connections are discussed below.

The work in Publication II was directly inspired by the work in Publication I, with the control state reduction presented in Publication II helping reduce the control state explosion resulting from the product construction inherent in the fusion described in Publication I.

Publication III describes a translation from C# to BSTs, which is used as a frontend in Publication I to specify individual transducers for the fusion backend.

RBBE in Publication I uses a reachability analysis combining underapproximation and overapproximation that was inspired by the work on the Dash algorithm in Publication IV. Dash integrates expanding the underapproxima-

tion and refining the overapproximation in an abstraction refinement loop, while RBBE only produces an underapproximation once as a pre-processing step.

The Dash algorithm in Publication IV generates tests using similar techniques as concolic testing in Publication V, and the LCTD verification tool that implements Dash is based on the LCT concolic testing tool that Publication V extends. The translation from C# into a transition rule for a BST in Publication III also performs a similar exploration of a program's execution tree as in concolic testing in Publication V, with both algorithms using an SMT solver to avoid exploring infeasible execution paths.

The work in Publication VI repurposes the extended version of the DPOR algorithm in Publication V for race detection. The tool is implemented as a modification to the LCT tool in Publication V.

Publications I–III are covered by Chapter 2. The contributions of Publications IV – VI are covered in Chapter 3. Chapter 4 concludes and provides directions for future work.

2. Symbolic Transducers for Stream Processing

This chapter introduces work on applying the theory of symbolic transducers to compiling pipelines of transducers into efficient code. The main objective is to allow users to write modular code without sacrificing the performance that can be gained from a monolithic implementation. The following is a real-world motivating example that is used as a case study for the work presented in this chapter.

Example 1. HTML encoding replaces multi-byte UTF8 characters with string encodings composed of single-byte characters. For reasons of robustness, HTML encoder implementations often first repair the input string by replacing invalid multi-byte UTF8 characters with the replacement character `0xFFFD`. UTF8 repairing and HTML encoding can be implemented as one fused pass, which does both transformations at the same time. This has the advantage of omitting the construction of the intermediate repaired-but-not-encoded string, as well as that of being able to share work to parse the UTF8 encoding. However, writing this hand-fused code results in greater complexity compared to implementing the steps separately and prevents the UTF8 repair implementation from being reused as-is in other anti-XSS encoders. ☒

The main optimization studied in this chapter for enabling performant modular code is *fusion*, which takes two transducers and produces a fused transducer that implements their serial composition. By repeatedly applying fusion, a pipeline of transducers can be reduced into a single transducer. In doing so, fusion directly addresses *communication overhead* by removing potentially costly communication between stages in a pipeline. Additionally, fusion exposes the context of each transducer, thus enabling further reductions that rely on inter-stage dependencies.

The computational model for the work in this chapter is that of symbolic transducers, which generalize the finite and typically small alphabets used in classical transducers to a decidable theory, such as one offered by a satisfiability modulo

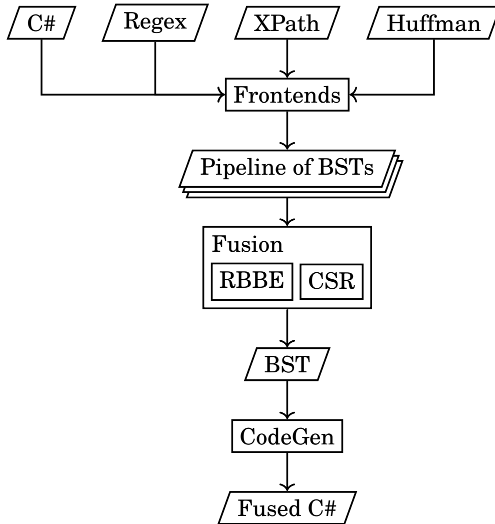


Figure 2.1. Architecture of the fusion engine

theories (SMT) solver. A new branching model of symbolic transducers called branching symbolic transducers (BSTs) is used for the work in this dissertation, which enables users to impart more control over the final generated code, thus preserving user intent and taking advantage of their domain knowledge. While implementations of all algorithms use BSTs, some work is presented for the sake of simplicity using the flat representation of symbolic transducers (STs) [85]. Section 2.1 provides definitions for the variants of symbolic transducers used.

The approach and techniques described in this chapter have been implemented in a tool that consumes pipelines of transducers specified using a variety of frontends and generates efficient fused code for the pipelines. Figure 2.1 presents a high-level breakdown of the steps used to compile a pipeline of comprehensions:

Frontends Pipelines written in the frontend languages are translated into BSTs. In addition to a general-purpose C# frontend, more specialized frontends for parsing and variable-length coding scenarios are provided. See Section 2.4 for details.

Fusion Given a pipeline of BSTs, adjacent pairs of BSTs will be fused until a single BST remains. Any order could in principle be used as fusion is semantically associative, but a user defined order is used as some evaluation orders will produce larger intermediate BSTs than others. The fusion process employs two reductions, reachability based branch elimination (RBBE) and control state reduction (CSR), that are applied after every pairwise fusion step. Section 2.2 illustrates the fusion algorithm, while Section 2.3 gives an overview of the

reductions.

CodeGen The final BST is translated into C# code, providing an efficient fused implementation of the pipeline. See Section 2.5 for a brief description of the code generation process.

The tool is evaluated using benchmarks on a variety of modular pipelines, as well as a comparison with a hand-fused monolithic pipeline corresponding to Example 1. Section 2.6 provides an overview of these evaluations.

This introduction presents the fusion and reduction algorithms on a high level with flow charts and a functional programming style. For lower level descriptions with more detail (including imperative pseudocode for fusion and RBBE), please see Publications I, II and III.

2.1 Symbolic Transducers and Automata

This section introduces the models of computation used in this chapter. The intuition behind symbolic transducers and automata along with the graphical notation used for them in this dissertation are presented first, followed by central definitions.

Instead of concrete alphabets, symbolic transducers and automata use formulas from a *background theory*, for which the satisfiability of predicates (i.e., Boolean valued formulas) must be decidable. In addition, the algorithms presented in this chapter will require more specific features, including Boolean operations, substitution and evaluating formulas against a model. In practice, such background theories are offered by SMT solvers, with the implementations in this dissertation using Z3 [33].

Some basic notation used for the examples in this section is presented here, while the rest of the notation used in this chapter is given in the Definitions subsection below. Lists are written $[e_1,\dots,e_n]$ and the empty list is ϵ. The examples in this dissertation use theories involving Unicode characters, which would be modeled as 32-bit bit vectors, and unbounded integers. A character literal 'a' is a constant matching the displayed character's Unicode codepoint. A literal string "ab" is shorthand for ['a','b'].

Intuitively, a symbolic finite automaton (SFA) [48] is like a classical automaton whose transitions are labeled with predicates instead of concrete characters. These predicates are symbolic representations of sets of inputs. A transition is taken from a given state if the predicate evaluates to true for the current input. This dissertation uses a convention of referring to the current input with the

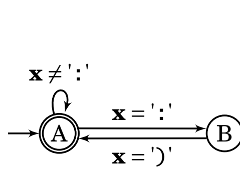


Figure 2.2. *EvenOdd*

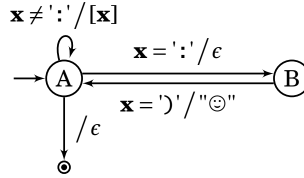


Figure 2.3. *EvenOddSubst*

variable **x**.

Example 2. Figure 2.2 presents an SFA that accepts all strings, where every ':' is followed by a ')'. It has an accepting state (A) and a rejecting state (B). The transition (A) $\xrightarrow{\mathbf{x}\neq\text{':'}}$ (A) accepts anything that does not contain ':', while the transitions (A) $\xrightarrow{\mathbf{x}=\text{':'}}$ (B) and (B) $\xrightarrow{\mathbf{x}=\text{')'}}$ (A) accept the pattern ":)". ☒

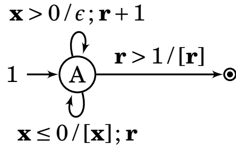
In addition to an input predicate, transitions in a symbolic finite transducer (SFT) [85] include a list of output formulas, which may depend on the input. The labels are written φ / w, where φ is the guard and w is the list of output formulas. When a transition is taken, these formulas are evaluated with the current input to produce a list of outputs. Instead of having accepting states, SFTs use a special class of transitions called *finalizers*, which are taken at the end of the input stream to produce additional final output.

Example 3. Figure 2.3 presents an SFT that accepts the same language as the SFA in Example 2, and that replaces each occurrence of ":)" in the input with a smiley "☺". The transition (A) $\xrightarrow{\mathbf{x}\neq\text{':'}/[\mathbf{x}]}$ (A) copies anything that does not contain ':' into the output. The transitions (A) $\xrightarrow{\mathbf{x}=\text{':'}/\epsilon}$ (B) and (B) $\xrightarrow{\mathbf{x}=\text{')'}/\text{"☺"}}$ (A) replace the pattern ":)" with the smiley character ☺.

(A) $\xrightarrow{/\epsilon}$ ⦿ is a finalizer that accepts input streams ending in (A) while producing no additional output. ☒

The state of a symbolic transducer (ST) [85] further includes a *register* from the background theory for storing additional state, which can be used in guards and output formulas. The labels $\varphi / w\,; g$ also include a *register update* formula g, which is evaluated when a transition is taken to set the new value of the register. As a convention, the variable **r** is used to refer to the current value of the register.

Example 4. Figure 2.4 presents an ST that for each strictly positive input increments the register with the transition (A) $\xrightarrow{\mathbf{x}>0/\epsilon;\mathbf{r}+1}$ (A), and simply copies all other inputs into the output with the transition (A) $\xrightarrow{\mathbf{x}\leq 0/[\mathbf{x}];\mathbf{r}}$ (A). Finally, a string of input is accepted if the register has been incremented at least once, in



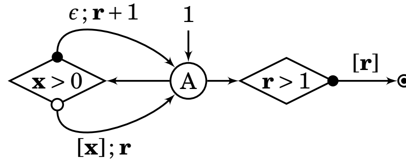
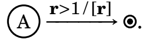


Figure 2.4. *IncOrOutput* **Figure 2.5.** *IncOrOutputBranching*

which case the final register value is appended to the output with the finalizer
. ☒

As a new variation of symbolic transducers, this dissertation presents branching symbolic transducers (BSTs), which replace the flat transitions of STs with branching transition rules.

Example 5. Figure 2.5 presents a BST that implements the same function as the ST in Example 4. The diamond shaped node left of (A) is an *if-then-else* rule, which evaluates the guard inside it to continue to either the true case marked with the black dot, or the false case marked with the empty white dot. The true and false cases are *base rules* that specify the outputs (nothing or the input, respectively) and register updates (increment or do nothing, respectively), and move to (A). In text, the whole transition rule is written $\mathbf{ite}(\mathbf{x} > 0, \xrightarrow{\epsilon;\mathbf{r}+1} (A), \xrightarrow{[\mathbf{x}];\mathbf{r}} (A))$.

The finalizer to the right of (A) is similarly a branching rule. The if-then-else rule does not have a false case, which means that the input is rejected if $\mathbf{r} \leq 1$. In text the finalizer rule is written $\mathbf{ite}(\mathbf{r} > 1, \xrightarrow{[\mathbf{r}]} \odot, \mathbf{undef})$. ☒

The motivation for branching is that it makes the evaluation order of guards explicit, which helps maintain user intent from the frontend languages used to specify symbolic transducers to the code generated in the backend. This also sidesteps the problem of having to choose an evaluation order during code generation.

Definitions

Before giving definitions of symbolic automata and transducers some notations for the background theories are introduced. The Boolean type is written $\mathbb{B} \stackrel{\text{def}}{=} \{\top, \bot\}$ and the type of integers is written $\mathbb{Z}$. The type with a single value is Unit. Given types τ and σ, the set of formulas denoting functions from τ to σ is written $\mathcal{F}(\tau \to \sigma)$. Formulas in $\mathcal{P}(\tau) \stackrel{\text{def}}{=} \mathcal{F}(\tau \to \mathbb{B})$ are called (τ-)predicates. Given a predicate φ its satisfiability is checked with $\mathrm{SAT}(\varphi)$.

This dissertation uses a convention of writing formulas that represent func-

tions using two variables: **x** for the input and **r** for the register. This results in lighter notation compared to using lambda terms. Theory level variables and built-in functions are written in a bold font, e.g., **x** is a variable and **head** is a built-in function.

Given types $\tau_1,\ldots,\tau_n$, the n-fold Cartesian product $\tau_1 \times \ldots \times \tau_n$ is the type for tuples of the form $(t_1,\ldots,t_n)$, where $t_i \in \tau_i$. Given a tuple t its i:th element is accessed with t_i. Named accessors will also be defined as required for clarity.

The type of finite-length lists of elements of type τ is written τ^*. A literal list of n elements is written $[a_1,\ldots,a_n]$ or $[a_i]_{i=1}^n$, and the empty list is ϵ. Concatenation of two lists v and w is written $v +\!\!+ w$. Given a list a, the first element of a is accessed with $\mathbf{head}(a)$, and the rest of a excluding the first element is $\mathbf{tail}(a)$.

Given a formula ψ and formulas $a_1,\ldots,a_n$ and $b_1,\ldots,b_n$, the formula where for each $i = 1\ldots n$ every occurrence of b_i has been substituted with a_i is written $\psi\{a_1/b_1,\ldots,a_n/b_n\}$. The substitutions happen simultaneously.

A mapping $M = \{a_1 \mapsto b_1,\ldots,a_n \mapsto b_n\}$ is a set of entries $(a_i \mapsto b_i)$ that denotes a function such that $\forall(a \mapsto b) \in M : M(a) = b$. A mapping can be updated using a new mapping N as $M \uplus N \stackrel{\text{def}}{=} N \cup \{a \mapsto b \mid (a \mapsto b) \in M \wedge \nexists b' : (a \mapsto b') \in N\}$, which overwrites any shared parts with entries from the new mapping.

Symbolic automata and transducers will be defined next.

Definition 1. A symbolic finite automaton (SFA) is a tuple $(\iota, Q, Q^0, \Delta, F)$, where ι is the *input type* from some decidable theory, Q is a finite set of *states*, $Q^0 \subseteq Q$ is the set of initial states, $\Delta \subseteq Q \times \mathcal{P}(\iota) \times Q$ is the transition relation, and $F \subseteq Q$ is the set of accepting states.

Given an SFA A, a transition $(q,\varphi,q') \in \Delta_A$ is written $q \xrightarrow[A]{\varphi} q'$ (as seen in Example 2). Subscripts are used to differentiate components of different automata and transducers. When the context is clear the name is omitted in the transitions and as subscripts. For the definition of the language of an SFA please see Publication II, Section 2.1.

Definition 2. A symbolic transducer (ST) is a tuple $(\iota, \rho, o, Q, q^0, r^0, \delta, \$)$, where ι, ρ and o are the *input*, *register* and *output types*, respectively, from some decidable theory; Q is a finite set of *control states*; $q^0 \in Q$ is the initial control state; $r^0 \in \rho$ is the initial register value; $\delta \subseteq Q \times \mathcal{P}(\iota \times \rho) \times \mathcal{F}(\iota \times \rho \to o)^* \times \mathcal{F}(\iota \times \rho \to \rho) \times Q$ is the transition relation; and $\$ \subseteq Q \times \mathcal{P}(\rho) \times \mathcal{F}(\rho \to o)^*$ is the finalizer relation.

A transition $(q,\varphi,w,g,q') \in \delta$ is written $q \xrightarrow{\varphi/w;g} q'$. A finalizer $(q,\varphi,w) \in \$$ is written $q \xrightarrow{\varphi/w} \odot$. Note that due to how δ and $\$$ were defined, for a single transition the number of outputs cannot depend on the input or the register. Please

see Publication II, Sections 2.2 and 2.3 for the definition of the transduction of an ST.

The introduction above introduced STs as an extension of SFTs, but the definition of SFTs is given here as a special case of STs.

Definition 3. A ST A is a symbolic finite transducer (SFT) if $\rho_A = \text{Unit}$.

For SFTs the notation for transitions is simplified by omitting the superfluous register update. Thus, a transition is written $q \xrightarrow{\varphi/w} q'$. For finalizers the guard is also omitted as any formula in $\mathcal{P}(\text{Unit})$ is necessarily equivalent to $\top$ or $\bot$, and for the $\bot$ case the finalizer can simply be omitted altogether. Thus, a finalizer is written $q \xrightarrow{/w} \circledcirc$.

For BSTs the sets of the tree-structured rules are defined first.

Definition 4. Given types τ, o and ρ, and a set of control states Q, the set of rules $\mathcal{R}(\tau,o,Q,\rho)$ is the maximal set such that for each $R \in \mathcal{R}(\tau,o,Q,\rho)$ one of the following holds:

- R is a base rule $\xrightarrow{w;g} q$, where $w \in \mathcal{F}(\tau \to o)^*$, $g \in \mathcal{F}(\tau \to \rho)$, and $q \in Q$.
- R is an undefined rule **undef**.
- R is an if-then-else rule $\textbf{ite}(\varphi,R_t,R_f)$, where $\varphi \in \mathcal{P}(\tau)$, and $R_t,R_f \in \mathcal{R}(\tau,o,Q,\rho)$.

The rest of this chapter omits handling of **undef** for simplicity of presentation. Please see Publications I, II and III for how **undef** is handled in the algorithms presented in this chapter.

Definition 5. A branching symbolic transducer (BST) is a tuple $(\iota,\rho,o,Q,q^0,r^0,\boldsymbol{\delta},\boldsymbol{\$})$, where ι, ρ and o are the *input*, *register* and *output types*, respectively, from some decidable theory; Q is a finite set of *control states*; $q^0 \in Q$ is the initial control state; $r^0 \in \rho$ is the initial register value; $\boldsymbol{\delta} : Q \to \mathcal{R}(\iota \times \rho,o,Q,\rho)$ is a function giving each control state a transition rule; and $\boldsymbol{\$} : Q \to \mathcal{R}(\rho,o,\{\circledcirc\},\text{Unit})$ is a function giving each control state a finalizer rule.

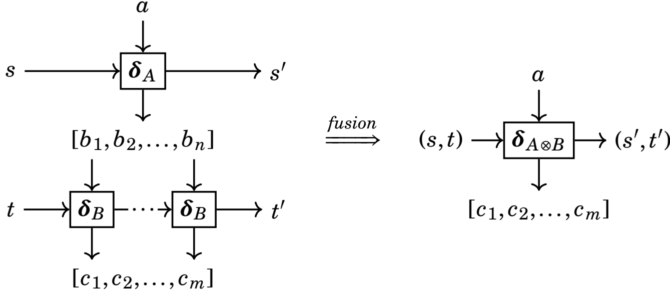
For a formal definition of the semantics of rules and the transductions of a BST, please see Publication I, Section 2.

2.2 Fusion

This section will introduce an algorithm for *fusing* pipelines of symbolic transducers. When a transducer outputs values of the same type that another transducer reads as input, they can be *serially composed* into a pipeline of two transducers

such that each output from the first is read as an input by the second. The objective of fusion is then to compute a single transducer that implements the same function as the serial composition.

The fusion operator $\otimes$ combines the serial composition of two BSTs into a single BST such that the result implements the transduction semantics of the composition. Given BSTs A and B such that $o_A = \iota_B$, their *fusion* is $A \otimes B$. The following diagram illustrates the relationship between $\boldsymbol{\delta}_A$, $\boldsymbol{\delta}_B$ and $\boldsymbol{\delta}_{A\otimes B}$:



The left side depicts how the pipeline consisting of A followed by B consumes one input a, moving from states s and t to states s' and t'. One transition in A outputs a list of values $[b_1, b_2, \ldots, b_n]$, which triggers n transitions in B producing the concatenated outputs $[c_1, c_2, \ldots, c_m]$.

On the right side the fused transition rule $\boldsymbol{\delta}_{A\otimes B}$ consumes a, moving in one step from the product state (s,t) to state (s',t'). The outputs $[c_1, c_2, \ldots, c_m]$ are produced without the intermediate list $[b_1, b_2, \ldots, b_n]$.

The core of the fusion algorithm is the construction of the fused transition rules $\boldsymbol{\delta}_{A\otimes B}$ and the fused finalizer rules $\boldsymbol{\$}_{A\otimes B}$. This section presents an algorithm for constructing $\boldsymbol{\delta}_{A\otimes B}$ in a functional style. For an imperative presentation of the algorithm, please see Publication I, Section 3.2. The construction of $\boldsymbol{\$}_{A\otimes B}$ is similar to that of $\boldsymbol{\delta}_{A\otimes B}$ and has been relegated to a remark at the end of this section.

The following helper functions are used for rewriting rules:

$$\begin{aligned}
\mathrm{rewriteIte}(h, \mathbf{ite}(\varphi, R_t, R_e)) &\stackrel{\text{def}}{=} h(\mathbf{ite}(\varphi, \mathrm{rewriteIte}(h, R_t), \mathrm{rewriteIte}(h, R_e))) \\
\mathrm{rewriteIte}(h, \xrightarrow{w;g} q) &\stackrel{\text{def}}{=} \xrightarrow{w;g} q
\end{aligned}$$

$$\begin{aligned}
\mathrm{rewriteBase}(h, \mathbf{ite}(\varphi, R_t, R_e)) &\stackrel{\text{def}}{=} \mathbf{ite}(\varphi, \mathrm{rewriteBase}(h, R_t), \mathrm{rewriteBase}(h, R_e)) \\
\mathrm{rewriteBase}(h, \xrightarrow{w;g} q) &\stackrel{\text{def}}{=} h(\xrightarrow{w;g} q)
\end{aligned}$$

The algorithms in this chapter also use the higher-order *left fold* function $\mathrm{foldL}(f, v, w)$, which recursively combines the initial value v with elements of the list w using the function f. Here the left associative variant is used. Now given $(p,q) \in Q_{A\otimes B}$, the rule $\boldsymbol{\delta}_{A\otimes B}(p,q)$ could be constructed by calling the

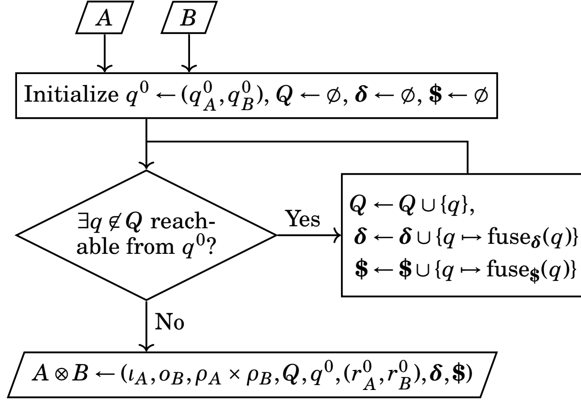


Figure 2.6. The incremental fusion algorithm

following function $\mathrm{fuse}_{\boldsymbol{\delta}}(p,q)$:

$$\begin{aligned}
\mathrm{fuse}_{\boldsymbol{\delta}}(p,q) &\stackrel{\text{def}}{=} \mathrm{rewriteBase}(\mathrm{fuseBase}_q, \boldsymbol{\delta}_A(p)) \\
\mathrm{fuseBase}_q(\xrightarrow{u\,;g} p') &\stackrel{\text{def}}{=} \mathrm{foldL}(\mathrm{step}, \xrightarrow{\epsilon\,;(g,r_2)} (p',q), u) \\
\mathrm{step}(R,v) &\stackrel{\text{def}}{=} \mathrm{rewriteBase}(\mathrm{stepBase}_v, R) \\
\mathrm{stepBase}_v(\xrightarrow{w\,;g} (p',q')) &\stackrel{\text{def}}{=} \mathrm{rewriteBase}(\mathrm{evalBase}_{v,w,g,p'}, \\
&\qquad \mathrm{rewriteIte}(\mathrm{evalIte}_{v,g}, \boldsymbol{\delta}_B(q'))) \\
\mathrm{evalIte}_{v,g}(\mathbf{ite}(\varphi, R_t, R_e)) &\stackrel{\text{def}}{=} \mathbf{ite}(\varphi\{v/\mathbf{x}, g_2/\mathbf{r}\}, R_t, R_e) \\
\mathrm{evalBase}_{v,w,g,p}(\xrightarrow{[f_i]_{i=1}^n\,;g'} q'') &\stackrel{\text{def}}{=} \xrightarrow{w+[f_i\{v/\mathbf{x}, g_2/\mathbf{r}\}]_{i=1}^n\,;(g_1, g'\{g_2/\mathbf{r}\})} (p,q'')
\end{aligned}$$

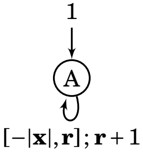
Each base rule $\xrightarrow{u\,;g} p'$ in $\boldsymbol{\delta}_A(p)$ is rewritten with $\mathrm{fuseBase}_q$, which iterates over the outputs u in a base rule to build an aggregate rule that corresponds to the execution of B from q using u as the inputs.

The aggregation is performed by step, which uses $\mathrm{stepBase}_v$ to extend the aggregate rule for each input term v. $\mathrm{stepBase}_v$ in turn uses $\mathrm{evalIte}_{v,g}$ and $\mathrm{evalBase}_{v,w,g,p}$ to rewrite the transition rule $\boldsymbol{\delta}_B(q')$ to use the aggregated register update $g = (g_1, g_2)$ and the current input term v, to output an aggregated prefix w and to move to a product control state (p,q''), where q'' is determined by the base rules of $\boldsymbol{\delta}_B(q')$.

The fusion could now be constructed as:

$$\begin{aligned}
A \otimes B &= (\iota_A, o_B, \rho_A \times \rho_B, \boldsymbol{Q}_A \times \boldsymbol{Q}_B, (q_A^0, q_B^0), (r_A^0, r_B^0), \boldsymbol{\delta}_{A\otimes B}, \boldsymbol{\$}_{A\otimes B}), \text{ where} \\
\boldsymbol{\delta}_{A\otimes B} &= \{(p,q) \mapsto \mathrm{fuse}_{\boldsymbol{\delta}}(p,q) \mid (p,q) \in \boldsymbol{Q}_{A\otimes B}\}
\end{aligned}$$

However, not all states in $\boldsymbol{Q}_A \times \boldsymbol{Q}_B$ are necessarily reachable. For example, if A only produces output in pairs, then states of B that require an odd number



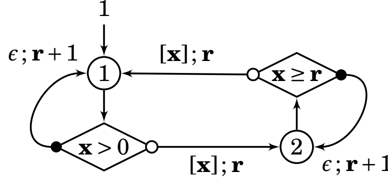


Figure 2.7. *AlwaysInc* **Figure 2.8.** *MaybeInc*

of inputs to reach are unreachable. Therefore, $A \otimes B$ should be constructed incrementally by starting from (q_A^0, q_B^0) and only fusing rules for reachable control states (see Publication I, Figure 6). This will result in a $Q_{A\otimes B}$ such that $Q_{A\otimes B} \subseteq Q_A \times Q_B$. Figure 2.6 presents the flow of the incremental fusion algorithm.

A second problem to be solved is that rules in $\boldsymbol{\delta}_{A\otimes B}$ may have unsatisfiable branches. Omitting register updates and control states for now, consider the rules $\xrightarrow{[|x|];\dots} \dots$ and $\mathbf{ite}(x < 0, \xrightarrow{\epsilon;\dots} \dots, \xrightarrow{[x];\dots} \dots)$, and their fusion $\mathbf{ite}(|x| < 0, \xrightarrow{\epsilon;\dots} \dots, \xrightarrow{[|x|];\dots} \dots)$. The condition $x < 0$ is satisfiable, but $|x| < 0$ is not and the fused rule can be *pruned* into just $\xrightarrow{[|x|];\dots} \dots$. The following function performs this kind of pruning:

$$\mathrm{prune}(\gamma, \mathbf{ite}(\varphi, R_t, R_e)) \stackrel{\text{def}}{=} \begin{cases} \mathrm{prune}(\gamma, R_t) & \text{if } \neg\,\mathrm{SAT}(\gamma \wedge \neg\varphi) \\ \mathrm{prune}(\gamma, R_e) & \text{if } \neg\,\mathrm{SAT}(\gamma \wedge \varphi) \\ \mathbf{ite}(\varphi, \mathrm{prune}(\gamma \wedge \varphi, R_t), & \text{otherwise} \\ \qquad \mathrm{prune}(\gamma \wedge \neg\varphi, R_e)) & \end{cases}$$

$$\mathrm{prune}(\gamma, \xrightarrow{w;g} q) \stackrel{\text{def}}{=} \xrightarrow{w;g} q$$

Given a rule R calling $\mathrm{prune}(\top, R)$ returns a rule where unsatisfiable branches have been removed. Pruning could be integrated into fusion by modifying $\mathrm{fuse}_{\boldsymbol{\delta}}$ to be:

$$\mathrm{fuse}_{\boldsymbol{\delta}}(p,q) \stackrel{\text{def}}{=} \mathrm{prune}(\top, \mathrm{rewriteBase}(\mathrm{fuseBase}_q, \boldsymbol{\delta}_A(p)))$$

Control states that have no path to them should be subsequently removed. However, pruning can also be more tightly integrated with fusion, which allows direct construction of the reduced set of control states. See Publication I, Figures 6 and 7 for a more algorithmic presentation of fusion, that performs pruning and fusion at the same time. The following example illustrates pruning between calls to step.

Example 6. Consider the BSTs *AlwaysInc* in Figure 2.7 and *MaybeInc* in Figure 2.8. This example will illustrate the construction of *AlwaysInc* $\otimes$ *MaybeInc*, which has input type $\iota_{AlwaysInc} = \mathbb{Z}$, register type $\rho_{AlwaysInc} \times \rho_{MaybeInc} = \mathbb{Z} \times \mathbb{Z}$

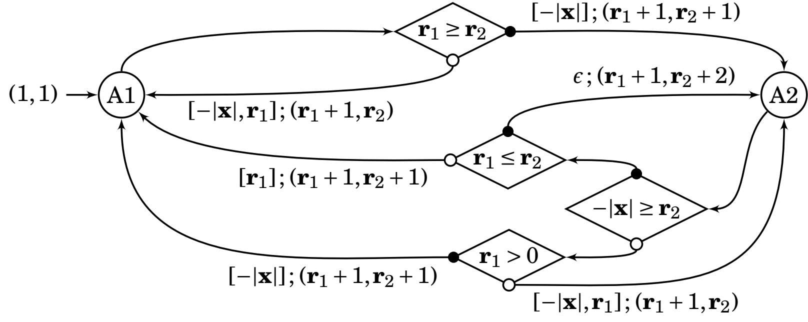


Figure 2.9. *AlwaysInc* ⊗ *MaybeInc*

and output type $\iota_{MaybeInc} = \mathbb{Z}$. Note that the example BSTs omit finalizer rules for simplicity.

The initial state is $\xrightarrow{(1,1)}$ (A1). To create $\boldsymbol{\delta}$((A1)) the rule $\boldsymbol{\delta}$((A)) $= \xrightarrow{[-|x|,r];r+1}$ (A) is fused with the execution of *MaybeInc* from (1). Since $\boldsymbol{\delta}$((A)) is a base rule, fusion expands the rule $\boldsymbol{\delta}$((1)) $= \mathbf{ite}(x > 0, \xrightarrow{\epsilon;r+1}$ (1), $\xrightarrow{[x];r}$ (2)) using the first output formula $-|x|$ as the input, producing:

$$\mathbf{ite}(-|x| > 0, \xrightarrow{\epsilon;(r_1+1,r_2+1)} \text{(A1)}, \xrightarrow{[-|x|];(r_1+1,r_2)} \text{(A2)})$$

However, the condition $-|x| > 0$ is unsatisfiable and the rule is simplified to:

$$\xrightarrow{[-|x|];(r_1+1,r_2)} \text{(A2)}$$

This rule is further expanded with the second output formula r_1 using the corresponding rule $\boldsymbol{\delta}$((2)) $= \mathbf{ite}(x \geq r, \xrightarrow{\epsilon;r+1}$ (1), $\xrightarrow{[x];r}$ (2)), giving the fused rule:

$$\boldsymbol{\delta}(\text{(A1)}) = \mathbf{ite}(r_1 \geq r_2, \xrightarrow{[-|x|];(r_1+1,r_2+1)} \text{(A2)}, \xrightarrow{[-|x|,r_1];(r_1+1,r_2)} \text{(A1)})$$

Since (A2) had not been previously seen, it is added to *AlwaysInc*⊗*MaybeInc*. The rule $\boldsymbol{\delta}$((A2)) will be constructed next, concluding the fusion process and producing the BST shown in Figure 2.9. ☒

The construction of $\$_{A\otimes B}(p,q)$ for some $(p,q) \in Q_{A\otimes B}$ is similar to that of $\boldsymbol{\delta}_{A\otimes B}$, except that any outputs from $\$_A(p)$ are first processed by rules from $\boldsymbol{\delta}_B$ before $\$_B$ is applied.

The fusion algorithm presented in this section can be applied to adjacent pairs of BSTs in a pipeline until a single BST remains. The next section introduces further reductions, which exploit opportunities exposed by fusion.

2.3 Reductions

Fusion is a worst-case quadratic operation both in the number of states and the number of transitions. While the pruning integrated into the procedure already provides reduction, more sophisticated program analyses can provide additional reduction. Reductions based on reachability analysis and automata minimization are introduced in Section 2.3.1 and Section 2.3.2 respectively.

2.3.1 Reachability Based Branch Elimination

The pruning performed during fusion eliminates branches that are unsatisfiable under all register values. However, not all register values are necessarily reachable. For example, the rule $\mathbf{ite}(-|x| \geq r_2, \ldots, \ldots)$ from (A2) in Figure 2.9 will always take its false branch, as r_2 is initialized to 1 and is only ever incremented. It is safe to replace this rule with its false branch, thus eliminating the true branch. The result of this transformation can be seen in Figure 2.11. This section presents the reachability based branch elimination (RBBE) algorithm, that performs this kind of analysis to eliminate unreachable branches.

RBBE uses a reachability analysis for STs as a subroutine. This analysis is presented first, followed by how it is used for branch elimination in BSTs.

Given an ST A, the following function isReachable$(A, q_{tgt}, \varphi_{tgt})$ attempts to check whether state q_{tgt} of A can be reached such that φ_{tgt} holds:

$$\text{isReachable}(A, q_{tgt}, \varphi_{tgt}) \stackrel{\text{def}}{=} \text{let } \Psi = \text{saturate}_A(\{q \mapsto \bot \mid q \in Q_A\} \uplus \{q_{tgt} \mapsto \varphi_{tgt}\}) \text{ in}$$
$$\text{SAT}(\Psi(q^0) \wedge (\mathbf{r} = r_A^0))$$
$$\text{saturate}_A(\Psi) \stackrel{\text{def}}{=} \begin{cases} \Psi & \text{if } \Psi = \text{next}_A(\Psi) \\ \text{saturate}_A(\text{next}_A(\Psi)) & \text{otherwise} \end{cases}$$
$$\text{next}_A(\Psi) \stackrel{\text{def}}{=} \text{foldL}(\text{propagate}, \Psi, \{(q, \varphi, g, q') \mid q \xrightarrow{\varphi/u;g} q' \in \delta_A\})$$
$$\text{propagate}(\Psi, (q, \varphi, g, q')) \stackrel{\text{def}}{=} \text{let } \psi = \varphi \wedge \Psi(q')\{g/\mathbf{r}\} \text{ in}$$
$$\text{subsume}(\Psi, q, (\mathbf{w} \neq \epsilon) \wedge \psi\{\mathbf{tail(w)}/\mathbf{w}, \mathbf{head(w)}/\mathbf{x}\})$$
$$\text{subsume}(\Psi, q, \gamma) \stackrel{\text{def}}{=} \begin{cases} \Psi \uplus \{q \mapsto \Psi(q) \vee \gamma\} & \text{if SAT}(\gamma \wedge \neg\Psi(q)) \\ \Psi & \text{otherwise} \end{cases}$$

The function isReachable$(A, q_{tgt}, \varphi_{tgt})$ first obtains a mapping Ψ from states in Q to reachability conditions and then checks whether the initial register value is included in $\Psi(q_A^0)$. The reachability conditions are created from an initial candidate Ψ_0, that just maps q_{tgt} to φ_{tgt}, with saturate(Ψ), which creates new candidates Ψ_{i+1} by calling next(Ψ_i) until a fixpoint with $\Psi_{i+1} = \Psi_i$ is reached.

The main reachability analysis is performed by next(Ψ), which returns a

Ψ' such that $\forall(q \xrightarrow{\varphi/w;g} q') \in \delta_A : (\varphi \wedge \Psi(q')\{g/\mathbf{r}\}) \implies \Psi'(q)$. In other words, next($\Psi$) expands Ψ one transition *backwards*. To do this it calls propagate() to include the backwards reachable states for each transition in turn.

To reason about strings of input, propagate($\Psi,(q,\varphi,g,q')$) uses a free variable $\mathbf{w}$ of type ι^*. In the predicate it creates for the backwards reachable states the conjunct $\mathbf{w} \neq \epsilon$ ensures that there is an input available. In the other conjuncts $\mathbf{w}$ is rewritten to $\mathbf{tail}(\mathbf{w})$ to make space for a new input at the beginning of the input list and $\mathbf{x}$ is rewritten to $\mathbf{head}(\mathbf{w})$ to evaluate them on the new first input.

To add a reachability condition γ for control state q into the mapping Ψ, the function subsume(Ψ,q,γ) is called, which performs a subsumption check to make sure adding the condition is necessary. If the existing $\Psi(q)$ implies the new condition γ, then Ψ is returned unchanged. Otherwise γ is included as a disjunct in $\Psi(q)$. This subsumption check is what allows the analysis to potentially reach a fixpoint.

As checking reachability properties in programs is undecidable in general, isReachable as it is presented here is not guaranteed to terminate, i.e., no fixpoint is necessarily reached. In practice, this problem is worked around by bounding the number of iterations. If the bound is reached, then the target is assumed to be reachable. Please see Publication I, Figure 8 for a version of the algorithm that does bound the number of iterations.

Now that the reachability analysis for STs has been introduced, the following will show how it is used for branch elimination in BSTs. For this purpose, BSTs are *flattened* into STs.

Given a rule R, paths(R) returns a mapping from base rules to their *path constraints*:

$$\begin{aligned}
\text{paths}(\xrightarrow{w;g} q) &\stackrel{\text{def}}{=} \{\left(\xrightarrow{w;g} q\right) \mapsto \top\} \\
\text{paths}(\mathbf{ite}(\varphi,R_t,R_f)) &\stackrel{\text{def}}{=} \text{extend}(\text{paths}(R_t),\varphi) \cup \text{extend}(\text{paths}(R_f),\neg\varphi) \\
\text{extend}(P,\xi) &\stackrel{\text{def}}{=} \bigcup_{(R\mapsto\psi)\in P} \{R \mapsto \xi \wedge \psi\}
\end{aligned}$$

For a given base rule R' appearing in R, paths(R)(R') captures the set of inputs and register values under which R' will be executed. These path constraints are used in the following construction:

Definition 6. Given a BST A, its *flattening* is the following ST:

$$\begin{aligned}
\text{st}(A) \stackrel{\text{def}}{=} (&\iota_A, o_A, \rho_A, Q_A, q_A^0, r_A^0, \\
&\{q \xrightarrow{\varphi/w;g} q' \mid q \in Q \wedge (\left(\xrightarrow{w;g} q'\right) \mapsto \varphi) \in \text{paths}(\boldsymbol{\delta}_A(q))\}, \\
&\{q \xrightarrow{\varphi/w} q' \mid q \in Q \wedge (\left(\xrightarrow{w} q'\right) \mapsto \varphi) \in \text{paths}(\boldsymbol{\$}_A(q))\})
\end{aligned}$$

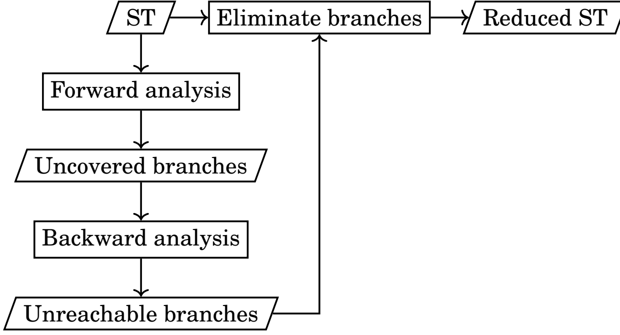


Figure 2.10. The reachability based branch elimination (RBBE) algorithm

For a given BST A and a control state $q \in Q_A$, the reachability of a specific rule that has the reachability condition φ_{tgt} can be checked with:

$$\text{isReachable}(\text{st}(A), q, \varphi_{tgt})$$

For example, a base rule R would have a reachability condition $\varphi_{tgt} = (\mathbf{w} \neq \epsilon) \wedge (\text{paths}(\boldsymbol{\delta}_A(q))(R))\{\mathbf{head}(\mathbf{w})/\mathbf{x}\}$. Now unreachable branches can be eliminated in a fashion similar to how unsatisfiable branches are pruned during fusion in Section 2.2.

The analysis performed by isReachable is relatively expensive and as such RBBE first performs a cheaper forward reachability analysis to avoid calling isReachable for base rules that are easily coverable. The flow of RBBE with the initial forward reachability analysis can be seen in Figure 2.10. This combination was inspired by the work on the Dash [9] algorithm in Section 3.2, and the forward analysis is also a form of symbolic execution. However, unlike the depth-first concolic execution that Dash employs, in RBBE the execution tree is expanded in a breadth-first fashion. Furthermore, any transitions into a control state that has previously been covered are not explored. This makes the analysis very cheap at the cost of reducing coverage. To regain some coverage, paths that enter a control state on the same iteration of the breadth first search are combined. Now in the backwards reachability phase, RBBE will not call isReachable for any base rules already covered.

The following example illustrates how RBBE can be applied to further reduce the fused BST produced in Example 6. Note that the variable $\mathbf{w}$ is not seen in this example, because the guards and register updates do not include $\mathbf{x}$.

Example 7. Consider *SimpleInc* in Figure 2.11, which is a simplified version of the BST in Figure 2.9. The unreachability of *Target* will be shown in the flattened st(*SimpleInc*) (Figure 2.12) by propagating the reachability condition

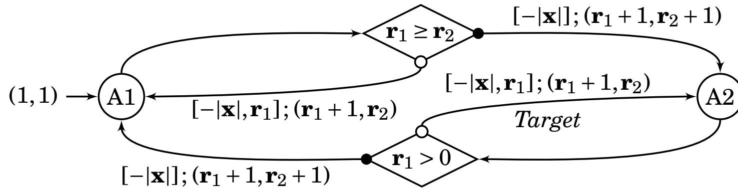


Figure 2.11. *SimpleInc*

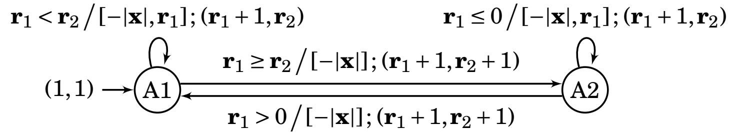


Figure 2.12. st(*SimpleInc*)

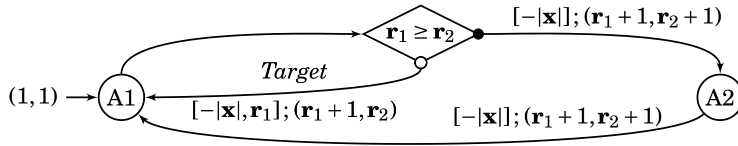


Figure 2.13. *SimpleInc*$'$

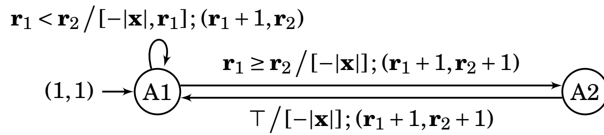


Figure 2.14. st(*SimpleInc*$'$)

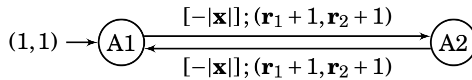


Figure 2.15. *SimpleInc*$''$

$\mathbf{r}_1 \le 0$ backwards from (A1) until a fixpoint is reached and by verifying that the initial state is not covered. The reachability condition for the first iteration is initialized as follows:

$$\Psi_0 \equiv \{(A1) \mapsto \bot, (A2) \mapsto \mathbf{r}_1 \le 0\}$$

Since $\Psi_0((A1))$ does not cover any states, all the conjuncts it might contribute to Ψ_1 are subsumed. The transitions to (A2) have the following contributions:

Transition	Target	Contribution
$(A2) \xrightarrow{\mathbf{r}_1 \le 0 / [-\lvert\mathbf{x}\rvert, \mathbf{r}_1]; (\mathbf{r}_1+1, \mathbf{r}_2)} (A2)$	$\Psi_1((A2))$	$\mathbf{r}_1 \le 0 \land \mathbf{r}_1 + 1 \le 0 \equiv$ $\mathbf{r}_1 \le -1$
$(A1) \xrightarrow{\mathbf{r}_1 > 0 / [-\lvert\mathbf{x}\rvert]; (\mathbf{r}_1+1, \mathbf{r}_2+1)} (A2)$	$\Psi_1((A1))$	$\mathbf{r}_1 > 0 \land \mathbf{r}_1 + 1 \le 0 \equiv$ $\bot$

Because $\mathbf{r}_1 \le 0$ subsumes $\mathbf{r}_1 \le -1$, the reachability condition for (A2) will not change. The reachability condition for the next iteration is:

$$\Psi_1 \equiv \{(A1) \mapsto \bot, (A2) \mapsto \mathbf{r}_1 \le 0\}$$

Since $\Psi_0 \equiv \Psi_1$ a fixpoint has been reached, and because $\Psi_1((A1)) \land \mathbf{r} = (1,1)$ is unsatisfiable *Target* has been proven unreachable. The *Target* branch is eliminated from *SimpleInc* to produce *SimpleInc′* shown in Figure 2.13, which also shows the next *Target* branch. Figure 2.14 shows the corresponding flattened st(*SimpleInc′*). The second round of RBBE is initialized with:

$$\Psi_0 \equiv \{(A1) \mapsto \mathbf{r}_1 < \mathbf{r}_2, (A2) \mapsto \bot\}$$

The transitions to (A1) have the following contributions:

Transition	Target	Contribution
$(A1) \xrightarrow{\mathbf{r}_1 < \mathbf{r}_2 / [-\lvert\mathbf{x}\rvert, \mathbf{r}_1]; (\mathbf{r}_1+1, \mathbf{r}_2)} (A1)$	$\Psi_1((A1))$	$\mathbf{r}_1 < \mathbf{r}_2 \land \mathbf{r}_1 + 1 < \mathbf{r}_2 \equiv$ $\mathbf{r}_1 + 1 < \mathbf{r}_2$
$(A2) \xrightarrow{\top / [-\lvert\mathbf{x}\rvert]; (\mathbf{r}_1+1, \mathbf{r}_2+1)} (A1)$	$\Psi_1((A2))$	$\mathbf{r}_1 + 1 < \mathbf{r}_2 + 1$

Because $\mathbf{r}_1 < \mathbf{r}_2$ subsumes $\mathbf{r}_1 + 1 < \mathbf{r}_2$, the reachability condition for (A1) will not change. The reachability condition for the next iteration is:

$$\Psi_1 \equiv \{(A1) \mapsto \mathbf{r}_1 < \mathbf{r}_2, (A2) \mapsto \mathbf{r}_1 + 1 < \mathbf{r}_2 + 1\}$$

Since $\Psi_1((A1)) \equiv \Psi_0((A1))$, all contributions from (A1) will be subsumed. The transition to (A2) has the following contribution:

Transition	Target	Contribution
$(A1) \xrightarrow{\top / [-\lvert x\rvert]; (\mathbf{r}_1+1, \mathbf{r}_2+1)} (A2)$	$\Psi_2((A1))$	$\mathbf{r}_1 + 2 < \mathbf{r}_2 + 2$

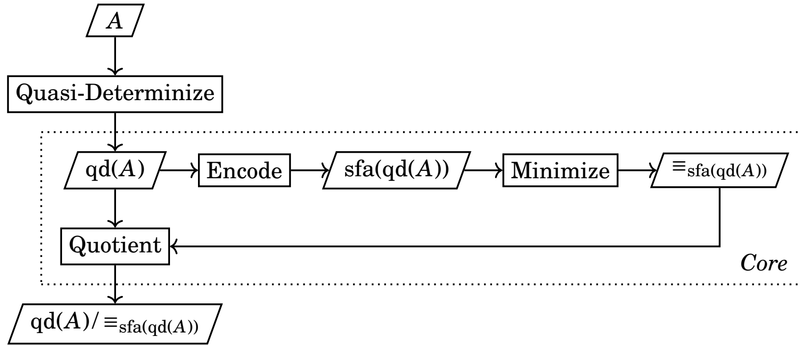


Figure 2.16. The CSR algorithm

Because $\mathbf{r}_1 < \mathbf{r}_2$ subsumes $\mathbf{r}_1 + 2 < \mathbf{r}_2 + 2$, it will hold that $\Psi_2 \equiv \Psi_1$ and thus a fixpoint has been reached. Now because $\Psi_2(\text{(A1)}) \wedge r = (1,1)$ is unsatisfiable, *Target* has been proven unreachable and is eliminated to produce *SimpleInc''* in Figure 2.15. ☒

RBBE is agnostic to the specific reachability analysis being employed, i.e., the isReachable function could be replaced with any reachability analysis for STs. Please see Chapter 4 for further discussion.

While RBBE provides reduction by removing transitions that are redundant due to being unreachable, the next section introduces a reduction that merges control states that are redundant due to having equivalent behavior.

2.3.2 Control State Reduction

Fusion commonly creates *indistinguishable* control states, i.e., states that have equivalent behavior for all inputs. The original inspiration for the work in this section came from fusions of the form *Huffman* ⊗ *Ignore*, where *Huffman* is an SFT implementing Huffman decoding [50] (Publication II, Section 7.1) and *Ignore* is any ST that ignores a part of the output from *Huffman*. In these kinds of fusions, a subgraph of the transducer will resemble an SFA due to producing no output and having no register updates. This insight led to a control state reduction (CSR) algorithm that uses an encoding of STs into SFAs.

The flow of the CSR algorithm is presented in Figure 2.16. The steps labeled *Core* will be introduced first, followed by *Quasi-Determinize*, which is used as a preprocessing step.

Encode The ST to be reduced is encoded into an SFA that operates on tuples representing transitions in the ST. For an ST with input type ι, output type o and register type ρ, its SFA encoding has the input type $\mathbf{T}(\iota \times \rho \times o^* \times \rho) \cup \mathbf{F}(\rho \times o^*)$,

which is a tagged sum type with the associated recognizer functions is**T** and is**F**. An instance t of $\mathbf{T}(\iota \times \rho \times o^* \times \rho)$ is used to represent a transition on input $t_i \in \iota$ that outputs $t_o \in o^*$ and updates the register from $t_r \in \rho$ to $t_{r'} \in \rho$. Finalizations are represented by instances of $\mathbf{F}(\rho \times o^*)$, which do not include the input or register update part, since these are not present in finalizers. The encoding is defined as follows:

Definition 7. Given an ST $A = (\iota, o, \rho, Q, q^0, r^0, \delta, \$)$ the *SFA encoding of A* is $\mathrm{sfa}(A) = (\mathbf{T}(\iota \times \rho \times o^* \times \rho) \cup \mathbf{F}(\rho \times o^*), Q \cup \{q^f\}, \{q^0\}, \Delta_{\mathrm{sfa}(A)}, \{q^f\})$, where q^f is a fresh state and $\Delta_{\mathrm{sfa}(A)}$ is the following transition relation:

$$\Delta_{\mathrm{sfa}(A)} \stackrel{\mathrm{def}}{=} \left\{ q \xrightarrow[\mathrm{sfa}(A)]{\substack{\mathrm{is}\mathbf{T}(\mathbf{x}) \wedge \varphi\{\mathbf{x}_i/\mathbf{x}, \mathbf{x}_r/\mathbf{r}\} \wedge \\ \mathbf{x}_o = [f_j\{\mathbf{x}_i/\mathbf{x}, \mathbf{x}_r/\mathbf{r}\}]_{j=1}^n \wedge \mathbf{x}_{r'} = g\{\mathbf{x}_i/\mathbf{x}, \mathbf{x}_r/\mathbf{r}\}}} q' \;\middle|\; q \xrightarrow[A]{\varphi/[f_j]_{j=1}^n ; g} q' \in \delta \right\} \cup$$
$$\left\{ q \xrightarrow[\mathrm{sfa}(A)]{\mathrm{is}\mathbf{F}(\mathbf{x}) \wedge \varphi\{\mathbf{x}_r/\mathbf{r}\} \wedge \mathbf{x}_o = [f_j\{\mathbf{x}_r/\mathbf{r}\}]_{j=1}^n} q^f \;\middle|\; q \xrightarrow[A]{\varphi/[f_j]_{j=1}^n} \odot \in \$ \right\}$$

Minimize Existing algorithms [31] for minimizing an SFA B internally calculate an equivalence relation $\equiv_B$ of states that accept the same language. CSR uses the equivalence relation of its encoding $\equiv_{\mathrm{sfa}(A)}$ obtained from the minimization of $\mathrm{sfa}(A)$.

Quotient The equivalence relation $\equiv_{\mathrm{sfa}(A)}$ can be used to merge $\equiv_{\mathrm{sfa}(A)}$-equivalent states in A, by selecting a representative state for each equivalence class and redirecting incoming transitions from other states to the representatives. The result is called the $\equiv_{\mathrm{sfa}(A)}$-quotient of A, which is written $A/\equiv_{\mathrm{sfa}(A)}$.

The core of the CSR algorithm as described above is enough to provide good reduction. However, the following preprocessing step can enable further reduction and provides, in conjunction with the steps above, a full minimization algorithm for deterministic SFTs.

Quasi-Determinize Sometimes transducers have states where all outgoing transitions have a shared prefix of output. If a state has such a prefix, then these outputs could be produced earlier by moving them to incoming transitions. This transformation, called quasi-determinization, can enable additional reduction (see Example 8).

For classical transducers, quasi-determinization can be implemented as follows [69]:

(1) For each state find the longest prefix of outputs in outgoing transitions.

(2) Remove any non-empty prefixes and append them onto each incoming transition.

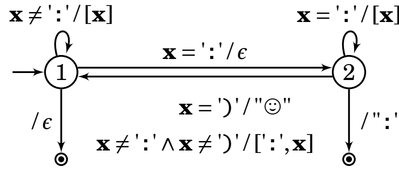


Figure 2.17. *Smileyfy*

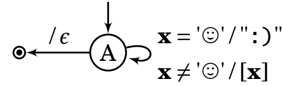


Figure 2.18. *Unsmileyfy*

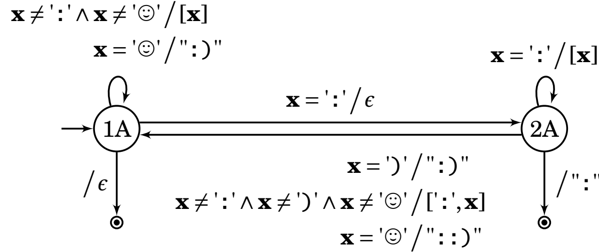


Figure 2.19. *SmileysFused*

(3) Repeat from step 1 until no more non-empty prefixes are found.

To generalize this algorithm to SFTs, a constant value analysis is performed first as follows. For each transition $\varphi / [f_i]_{i=1}^n$ and each $i = 1 \dots n$, if $\varphi \wedge \varphi \{\mathbf{x}'/\mathbf{x}\} \wedge (f_i \neq f_i \{\mathbf{x}'/\mathbf{x}\})$ is unsatisfiable, then that output formula has a constant value in the context of φ. Any such outputs are replaced with constant value obtained by evaluating f_i in the model of φ.

Now given an SFT A, the quasi-determinization $\mathrm{qd}(A)$ is produced by first performing the constant value analysis, and then running a version of the classical quasi-determinization algorithm, where non-constant outputs are blocked from being moved. Note that using syntactic equivalence of output formulas in the classical quasi-determinization algorithm would not be sound, as input dependent outputs might be moved resulting in them being evaluated for a different input than in the original SFT.

If A is a deterministic SFT then $\mathrm{qd}(A)/\equiv_{\mathrm{sfa}(\mathrm{qd}(A))}$ is minimal (Publication II, Theorem 4). This result mirrors the minimization theorem for classical transducers in [69, Theorem 2].

The generalization of quasi-determinization to STs has some additional considerations that are discussed in Publication II. Additionally, Publication II, Section 6 presents a register invariant based strengthening approach that can enable more control state reduction for STs.

The following example illustrates how CSR provides reduction for a fusion of SFTs.

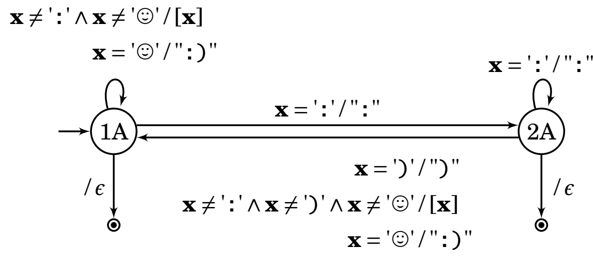


Figure 2.20. qd(*SmileysFused*)

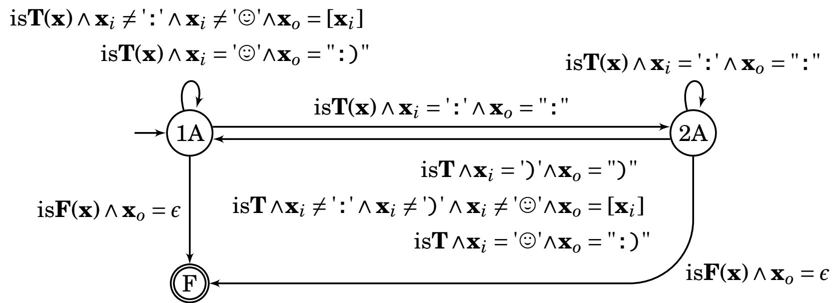


Figure 2.21. sfa(qd(*SmileysFused*))

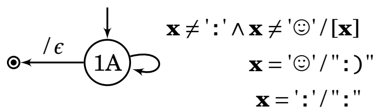


Figure 2.22. qd(*SmileysFused*)$/\equiv_{\text{sfa(qd(}\textit{SmileysFused}\text{))}}$

Example 8. Consider the pipeline of string transformations where: (1) *Smileyfy* first replaces all instances of ":)" with "☺", and (2) *Unsmileyfy* then replaces all instances of "☺" with ":)". Note that this pipeline is equivalent to just *Unsmileyfy*, as every "☺" produced by *Smileyfy* is expanded back to ":)" by *Unsmileyfy*. Figure 2.17 gives the SFT for *Smileyfy*, and the one for *Unsmileyfy* is found in Figure 2.18.

Now consider the fusion *SmileysFused* = *Smileyfy* ⊗ *Unsmileyfy*, given in Figure 2.19. Due to being equivalent to *Unsmileyfy*, CSR will reduce it to one control state. The rest of this example will work through the steps of applying CSR to *SmileysFused*.

Constant value analysis on the output in (2A) $\xrightarrow{\mathbf{x}=\texttt{':'}/[\mathbf{x}]}$ (2A) will find that $\mathbf{x} = \texttt{':'} \wedge \mathbf{x}' = \texttt{':'} \wedge \mathbf{x} \neq \mathbf{x}'$ is unsatisfiable. Evaluating the output formula $\mathbf{x}$ in the model of the guard $\mathbf{x} = \texttt{':'}$ gives ':' and thus the transition will be rewritten to (2A) $\xrightarrow{\mathbf{x}=\texttt{':'}/\texttt{":"}}$ (2A).

Now during quasi-determinization (2A) has the following outgoing transitions:

(2A) $\xrightarrow{\mathbf{x}=\texttt{':'}/\texttt{":"}}$ (2A) (2A) $\xrightarrow{\mathbf{x}=\texttt{'☺'}/\texttt{"::)"}}$ (1A)
(2A) $\xrightarrow{\mathbf{x}=\texttt{')'}/\texttt{":)"}}$ (1A) (2A) $\xrightarrow{/\texttt{":"}}$ ⊙
(2A) $\xrightarrow{\mathbf{x}\neq\texttt{':'}\wedge\mathbf{x}\neq\texttt{')'}\wedge\mathbf{x}\neq\texttt{'☺'}/[\texttt{':'},\mathbf{x}]}$ (1A)

These share a prefix of ":", which is removed and appended to the incoming transitions. Figure 2.20 shows the resulting qd(*SmileysFused*).

The SFT qd(*SmileysFused*) is encoded into sfa(qd(*SmileysFused*)) as shown in Figure 2.21. Minimizing this SFA using an existing algorithm [31] produces an equivalence relation $\equiv_{\mathrm{sfa}(\mathrm{qd}(\mathit{SmileysFused}))}$, which has the equivalence class {(1A),(2A)}. Notice that $\equiv_{\mathrm{sfa}(\mathit{SmileysFused})}$ would not have this equivalence class, as the transitions (1A) $\xrightarrow{\mathbf{x}=\texttt{'☺'}/\texttt{":)"}}$ (1A) and (2A) $\xrightarrow{\mathbf{x}=\texttt{'☺'}/\texttt{"::)"}}$ (1A) in Figure 2.19 have different outputs for the same input.

Taking the $\equiv_{\mathrm{sfa}(\mathrm{qd}(\mathit{SmileysFused}))}$-quotient of qd(*SmileysFused*) gives the SFT shown in Figure 2.22. Here (1A) has been chosen as the representative of its equivalence class, and thus the transition (1A) $\xrightarrow{\mathbf{x}=\texttt{':'}/\texttt{":"}}$ (2A) has been redirected to (1A) instead. ⊠

Note that even though the SFT in Figure 2.22 is minimal in the number of control states, it has one extra transition compared to *Unsmileyfy*. Furthermore, even if a minimal number of transitions is achieved, the representation of guards and outputs may not be minimal. The complexity of minimizing these will depend on the background theory in question. For example, with binary decision diagrams (BDDs) minimization is NP-complete [18], while Boolean formula minimization is NP^{P}-complete [20].

This section has presented CSR for the flat models of symbolic transducers. To apply CSR to BSTs the flattening introduced in Section 2.3.1 can be used. An equivalence relation $\equiv_{\mathrm{sfa}(\mathrm{st}(A))}$ for a flattening $\mathrm{st}(A)$ is also valid for A.

2.4 Frontends

Good frontend languages for specifying BSTs are required for the fusion techniques described in this dissertation to be useful. This section introduces a general-purpose frontend that translates a subset of imperative C# code into BSTs, followed by several domain specific frontends. In this introduction, the frontends are explained by way of example.

The following example illustrates the general-purpose C# frontend.

Example 9. The following C# class specifies the BST presented in Figure 2.17:

```
partial class Smileyfy : Transducer<char, char> {
    char Smiley() { return '\x263A'; }

    bool wasColon = false;

    public override IEnumerable<char> Update(char c) {
        if (wasColon) {
            if (c == ')')
                yield return Smiley();
            else {
                yield return ':';
                if (c != ':') yield return c;
            }
        } else {
            if (c != ':') yield return c;
        }
        wasColon = (c == ':');
    }

    public override IEnumerable<char> Finish() {
        if (wasColon) yield return ':';
    }
}
```

The code uses an instance variable `wasColon` to track whether ‘:’ has been matched, and a helper function `Smiley` to produce the character ‘☺’. ⊠

The C# frontend supports complex control flow via a symbolic execution based translation into BSTs. Any instance variables of type `bool` are turned into control state using a partial register exploration, which expands parts of the register into control states. Please see Publication III, Sections 3 and 4 for details.

Writing string pattern matchers as imperative code is cumbersome. The following regular expression based frontend is more appropriate for light-weight text parsing scenarios.

Example 10. The following code specifies a BST that parses the fourth column on each row in a comma-separated values file into a 32-bit integer.

```
[ParsingMatcher(@"(([^,]*,){3}(?<value>\d+),[^\n]*\n)*",
                 "int")]
partial class Col4Int : SpecialTransducer { }
```

Here `([^,]*,){3}` skips to the fourth column, `(?<value>\d+)` specifies a *named capture group* for the value to parse, and `,[^\n]*\n` skips the remaining columns on the row. The second parameter, `"int"`, specifies that each `value` captured should be parsed using a predefined BST that parses integers (shown in Publication I, Figure 4b). ⊠

The translation from the regular expression based frontend into BSTs employs a standard translation into (symbolic) automata. The frontend fuses the specified BST for parsing the individual captures onto the transitions that correspond to matching inside the named capture group. This can be seen as a *hierarchical* form of fusion. Please see Publication I, Section 5.2 for details.

Regular expressions are not sufficient for parsing the nested structure of XML, and instead a frontend based on XPath, a query language for XML, can be used.

Example 11. The following code specifies a BST that parses the contents of each `<year>` under a path of `<dblp><article>` into a 32-bit integer.

```
[XPathMatcher("/dblp/article/year", "int")]
partial class DBLPYear : SpecialTransducer { }
```

The same predefined transducer as in Example 10 is used to parse the contents of each matched tag. ⊠

The BSTs generated from the XPath frontend use an integer register to keep track of the nesting level of XML tags.

Data processing frameworks may also include various compression schemes. Huffman coding [50] is one that can be naturally implemented as an SFT (and thus a BST).

Example 12. The following code specifies BSTs for a Huffman decoder and encoder:

```
[HuffmanDecoder(@"example.txt")]
partial class ExampleDecoder : SpecialTransducer { }
[HuffmanEncoder(@"example.txt")]
partial class ExampleEncoder : SpecialTransducer { }
```

The distribution of bytes in the file "example.txt" is used to construct the Huffman coding. ExampleDecoder decodes variable length bit patterns into bytes, while ExampleEncoder outputs the appropriate bit patterns for the bytes it reads. ⊠

Please see Publication II, Section 7.1 for the constructions of encoding and decoding SFTs for Huffman coding (and other prefix codes).

Finally, there must be a way to specify pipelines of BSTs to fuse.

Example 13. If First and Second are BSTs specified elsewhere, then the following constructs their fusion First⊗Second:

```
partial class Fused : Composition<First, Second> { }
```

Fused can be further fused with other BSTs. ⊠

This interface for specifying fusions gives the user control over the order that fusions are performed in. This is important, as although fusion is semantically associative, the sizes of intermediate BSTs may depend on the order in which the fusions are performed.

2.5 Implementation

The tool for fusing pipelines of symbolic transducers described in this chapter has been implemented as a part of the Automata library[1], which provides C# implementations of algorithms for symbolic automata and transducers. The tool parses the users C# project to find definitions of BSTs and pipelines thereof using the Roslyn compiler framework[2].

All the frontends presented in Section 2.4 require that the transducer class is partial, which allows the code generator to easily inject a new function into each transducer class that implements the transduction. The following example gives a general idea of the kind of code that is generated.

Example 14. Consider the BST *IncOrOutputBranching* in Figure 2.5, for which the code generator would create code similar to the following:

```
partial class IncOrOutputBranching {
    IEnumerable<int> Transduce(IEnumerable<int> i) {
        int r = 1;
        IEnumerator<int> input = i.GetEnumerator();
    StateA:
        if (!input.MoveNext())
            goto StateAFinalizer;
```

[1] https://github.com/AutomataDotNet/Automata
[2] https://github.com/dotnet/roslyn

```
        if (input.Current > 0) {
            r = r + 1;
            goto StateA;
        } else {
            yield return input.Current;
            goto StateA;
        }
    StateAFinalizer:
        if (r > 1)
            yield return r;
        else
            throw new Exception();
        yield break;
    }
}
```

The function `Transduce` has for each control state a labeled block containing code generated from the control state's transition rule. The code generator translates if-then-else rules into `if` statements, undefined rules into throwing an exception, and base rules into assignments to the register, `yield return` statements for the outputs and a final `goto` to the label of the target control state's block. Each block starts with code for reading a new input, and if no more inputs are available then the code jumps to a block that implements the control state's finalizer. ☒

The design of the code generator takes into account various considerations, including:

- Common subexpressions inside the tree structure of transition rules are evaluated only once into a temporary variable.
- The code blocks for control states are generated in a greedily sorted order, that tries to place code for successor states close to predecessor states.
- When the input and output types are `byte` the `System.IO.Stream` class can be used for more efficient input and output.

The generated code has high throughput, as the evaluations in the next section show.

2.6 Evaluations

This section will introduce the evaluations that have been performed for the tool for fusing pipelines of symbolic transducers and highlight the key results. Details of the experimental setup and full descriptions of the benchmarked pipelines can be found in Publication I, Section 6 and Publication II, Section 8.

Comparisons with Modular Pipelines in Publication I, Section 6

This evaluation compares the throughputs of various modularly implemented pipelines. It finds that the fused C# code is on average 3.4× faster than reasonable hand-written implementations that use the standard libraries where possible.

Comparison with Hand-Fused Code in Publication I, Section 6.1

This evaluation compares the throughput of a monolithic implementation of HTML encoding (please see Example 1) from the .NET 4.5 standard libraries with fused code generated by the tool from a modular implementation. The throughput for the code generated from the monolithic implementation is comparable to that of the monolithic implementation.

Efficacy of RBBE in Publication I, Section 6

The number of if-then-else rules removed by RBBE is evaluated on fusions of the modular pipelines presented in the same section. In many cases RBBE can remove a large fraction of the if-then-else rules in these pipelines.

Efficacy of CSR in Publication II, Section 8

CSR is also evaluated on the modular pipelines used in Publication I, Section 6. On average 25% of the control states are removed. For a Huffman decoder for English text fused with line counting, which represents the original inspiration of CSR, 72% of the control states are removed.

2.7 Related Work

There is a rich body of work on stream computations with internal state [35, 61, 68, 70, 80]. There also exists a number of stream libraries that provide APIs for expressing stateful operations. Apache Flink [6] and Spark Streaming[3] both allow stateful stream operations and provide fault tolerance for them in a distributed setting. Conduit[4] and Highland.js[5] are examples of traditional stream libraries with explicit support for stateful operations.

The term *fusion* is used in literature for two related operations:

[3] `http://spark.apache.org/streaming/`
[4] `https://github.com/snoyberg/conduit`
[5] `http://highlandjs.org/`

(1) Removing intermediate streams between operations [29].

(2) Merging adjacent stream operations into a single one.

The first meaning is connected to the more general notion of *deforestation* [87]. The term deforestation will be used for the rest of this discussion, while the term fusion is reserved for the latter meaning. Fusion directly implies that deforestation is also performed. In the other direction, deforestation must replace the streams with some other communication mechanism, such as function calls, which can also be seen as a lightweight form of fusion. The evaluation in Publication I, Section 6 finds that the SMT based fusion approach presented in this dissertation provides on average a 2.6× speedup over deforestation using function calls.

Fusion for stateful operations has been studied in the StreamIt [83] language for operations with a linear state space representation [5], i.e., one where the outputs and new state are linear combinations of the inputs and previous state. An advantage of such a representation is that the fusion has a worst-case linear blow-up instead of the quadratic one for the fusion studied in this dissertation. On the other hand, the fusion of BSTs handles any stateful operation for which the state update is over a decidable theory.

In addition to fusion, StreamIt also performs *fission*, which is the opposite transformation where a single transformation is broken into a pipeline of transformations [83]. The motivation for this is that while fusion reduces communication cost, it also reduces the available pipeline parallelism [47]. Thus, in a parallel setting applying fusion might not always be appropriate and it might even be beneficial to apply fission [76]. Please refer to Publication I, Section 7 for a more extensive review of related work in the area of stream processing.

Previous work on symbolic transducers has focused on the flat representation introduced in [85], which also presented a composition algorithm (called fusion in this dissertation) for the flat, finite case of SFTs. Additionally [85] stated that STs are closed under composition, but did not provide an algorithm.

CSR builds on previous work on the minimization of finite state transducers [24, 69]. Transducer minimization has also been studied for natural language processing [36, 67]. As a related problem, minimization of control flow graphs has been studied in [28] by reduction to a variant of classical automata minimization. There exists a huge body of work on minimization for various automata models [3, 1, 63, 31, 17]. For more work related to minimization of transducers, please refer to Publication II, Section 9.

Previous work on register elimination in STs has focused on fully eliminating

the register to arrive at a representation of the same transducer in a finite state model [86, 32], which is not always possible. In contrast, the Boolean register elimination used in Section 2.4 always succeeds but does not remove non-Boolean parts of the register.

3. Automated Testing and Verification

This chapter introduces work on automated test generation and software verification, for which the common denominator here is *concolic testing* [41, 78, 77], also known as *dynamic symbolic execution*. Concolic testing is a test generation method that combines *conc*rete and symb*olic* execution. This combination provides a systematic approach to exploring program behavior, while avoiding false positives arising from imprecise symbolic execution. The work on concolic testing in this dissertation builds on top of the concolic testing tool LCT [56]. Two separate lines of work are discussed:

Verifying sequential programs Section 3.2 presents LCTD, a tool implementing the software verification algorithm Dash [9], which combines concolic testing with predicate abstraction. The tests executed explore an underapproximation of the program, which allows large amounts of easily reachable code to be cheaply covered. The predicate abstraction on the other hand maintains an overapproximation that is refined when test execution fails. In this way, the under- and overapproximation are refined together until either an error is reached or is proven unreachable. Dash only supports sequential programs.

Testing concurrent programs Concolic testing is combined with dynamic partial order reduction (DPOR) [38] for efficient testing of multi-threaded programs. DPOR is an algorithm for exploring reduced sets of interleavings of operations that that guarantee that no errors are hidden. DPOR works *dynamically*, i.e., it inspects program traces to find concurrent operations that are dependent on each other and introduces new schedules to be explored to ensure that all behaviors are covered. The combination of concolic testing and DPOR is natural: a combined execution tree is explored, where concolic testing introduces data based branching decisions, while DPOR introduces concurrency based scheduling decisions.

The result of two threads reading a shared variable does not depend on the order the reads are executed in, i.e., read operations commute. Section 3.3 includes

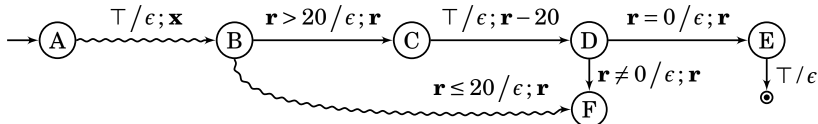


Figure 3.1. The program-under-test and the first test

an extension to DPOR that exploits this commutativity of read operations.

As DPOR already tracks which operations are *concurrent* it should intuitively be easy to extend it to also detect data races, in addition to the detection of deadlocks and assertion errors considered in previous work [38]. Section 3.3 discusses this observation and how DPOR can be used for race detection.

This chapter introduces the algorithms involved and the improvements made to them using high-level descriptions and examples. Please see Publication IV for an extensive description of Dash including how to implement it for the LLVM intermediate representation [57]. For a description of DPOR and the improvement presented in this dissertation, please see Publication V. Finally, Publication VI presents an approach for detecting data races during DPOR along with a proof of correctness.

3.1 Concolic Testing

Concolic testing is a test generation method based on symbolic execution that can systematically explore all paths through a program-under-test. In concolic testing concrete and symbolic execution happen simultaneously, such that for each concrete operation executed the symbolic executor performs a corresponding operation on the symbolic side. In the symbolic execution, the inputs to the program are represented as variables in the formulas that the symbolic operations produce. At the end of the test execution the symbolic execution will have gathered a *path constraint* as a conjunction of formulas describing each operation executed. The values for the input variables that satisfy a path constraint are exactly the set of inputs that cause the program to take the path in questions.

To systematically test a program, these path constraints are used to solve inputs that drive the program to previously unexplored paths. Repeating this process will explore new paths in the program until there are no more paths to explore.

Example 15. Figure 3.1 presents a target program for concolic testing. The

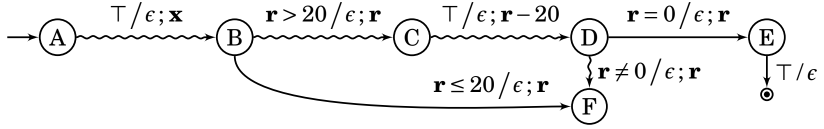


Figure 3.2. The second test

notation for STs from Chapter 2 is reused here as the computational model. The program is equivalent to the one in Publication IV, Figure 2.6. The property that will be tested is whether the program accepts any input string, which here means reporting whether (E) $\xrightarrow{\top/\epsilon}$ ⊙ is covered. Notice that only the first transition (A) $\xrightarrow{\top/\epsilon;\mathbf{x}}$ (B) depends on the input, and thus only the first input affects the execution. Alternatively, one could say that the program only reads one input value. The variable $\mathbf{x}_1$ will be used to refer to the first input.

Assume the first test execution selects a random value of 15 for its first input, which will cause the execution to follow the path marked in Figure 3.1 with a wavy line. The path constraint gathered is $\mathbf{x}_1 \leq 20$. To explore the other branch at (B) the part of this path constraint that corresponds to the branch (which happens to be the whole path constraint) is negated into $\mathbf{x}_1 > 20$, for which an SMT solver will return a satisfying assignment, e.g., $\{\mathbf{x}_1 \mapsto 38\}$.

The new input will drive the next test execution along the path marked in Figure 3.2, which produces a path constraint of $\mathbf{x}_1 > 20 \wedge \mathbf{x}_1 - 20 \neq 0$. Now for the other branch at (D) the last conjunct is negated, producing $\mathbf{x}_1 > 20 \wedge \mathbf{x}_1 - 20 = 0$. This path constraint is, however, unsatisfiable and the concolic testing process has finished as there are no other unexplored branches along either test execution. ⊠

The previous example illustrates the main idea behind concolic testing, but it does not show the advantages concolic testing over other forms of symbolic execution. Specifically, programs typically involve computation with values that do not directly depend on input and in concolic testing this computation is executed "concretely" by the program-under-test instead of being interpreted by a symbolic execution engine. This allows the implementer of the concolic tester to benefit from the performance offered by compilers instead of having to duplicate that work. Another advantage is that if there is any discrepancy between the semantics of the program-under-test and those implemented by the symbolic execution, then in concolic testing the tests will diverge from the expected path in a detectable way instead of such inaccuracy going unnoticed.

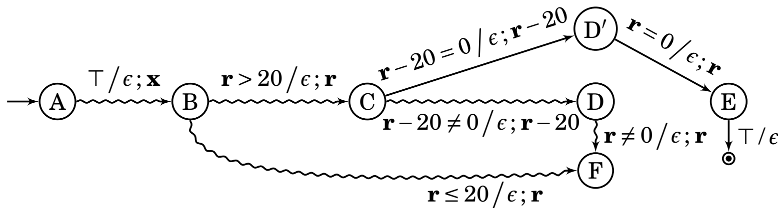


Figure 3.3. The first refinement

3.2 Abstraction Refinement with Concolic Testing

While systematic test generation is effective for exploring program behaviors to find bugs, it cannot in general prove safety properties, as exploring all possible paths is infeasible even for medium sized programs. Instead, software verification algorithms use *abstractions* which make checking for errors cheap. An abstraction that does not contain any errors is a proof of safety as long as it overapproximates the behaviors of the program. However, finding the right abstraction is challenging as a very precise abstraction may be too large to check for errors efficiently, while a very imprecise abstraction may contain erroneous behavior that the target program does not have.

A trace that leads to an error in an abstraction is called a *counterexample*, while a counterexample that does not exist in the target program is a *spurious counterexample*. To arrive at a sufficiently precise abstraction many software verification algorithms use *counterexample-guided abstraction refinement* [26], where an initial imprecise abstraction is iteratively refined using information from spurious counterexamples. A refinement step must remove the spurious counterexample while maintaining all true program traces. A good refinement step will generalize from a single spurious counterexample to remove a whole class of them.

The Dash algorithm from Beckman et al. [9] combines concolic testing with *predicate abstraction* [43], which abstracts the state space of a program to the different valuations of a set of Boolean predicates over concrete states. Dash operates similarly to a concolic tester up to the point that test generation fails, at which point it refines the abstraction in a way that eliminates the path for which the test generation failed. This refinement process will terminate if Dash is able to split the abstraction into a part covered with concolic testing and a part from which the error is reachable. The following example illustrates the high-level approach employed by Dash.

Example 16. Dash picks up from Figure 3.2, where the concolic testing finished

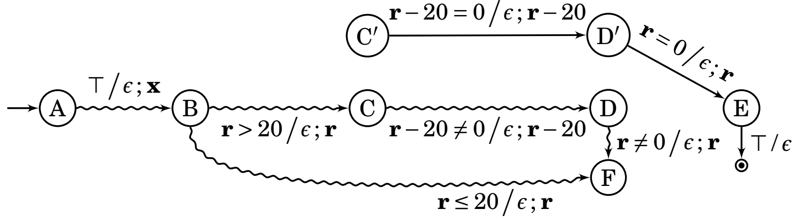


Figure 3.4. The second refinement

after failing to generate a test to cover (D) $\xrightarrow{\mathbf{r}=0/\epsilon;\mathbf{r}}$ (E). The abstraction is refined by splitting (D) with a predicate $\mathbf{r} = 0$, which places the existing test execution on the false side as shown in Figure 3.3.

Note that in the notation used for this example the predicates for (D) and (D′) are expressed on the incoming transitions, while in Publication IV the predicates are directly tied to the control states.

Dash now returns to concolic execution and attempts to generate a test to cross (C) → (D′). The path constraint $\mathbf{x}_1 > 20 \wedge \mathbf{x}_1 - 20 = 0$ is unsatisfiable, which justifies splitting (C) with the predicate $\mathbf{r} - 20 = 0$ as shown in Figure 3.4. This time, however, the transition from (B) to (C′) would have the guard $\mathbf{r} > 20 \wedge \mathbf{r} - 20 = 0$, which, being unsatisfiable, can be immediately omitted. This is similar to the pruning performed by fusion in Section 2.2.

At this point there is no longer any path to the finalizer (i.e., the error) in Figure 3.4, which proves its unreachability. ⊠

In this dissertation Dash has been implemented in the open source software verification tool LCTD, which builds upon the concolic testing tool LCT. The implementation targets sequential C programs compiled to the LLVM intermediate representation [57], and Publication IV includes a full description of how the predicates for refining the abstraction are constructed for LLVM basic blocks.

A main contribution made by Beckman et al. with Dash [9] is how the predicates for refining the abstraction are constructed. The way to combine concolic testing with abstraction refinement used in Dash was first introduced in the Synergy algorithm [44], and Dash introduces improved support for programs that use pointers. Specifically, the predicates used in Dash are specific to the pointer aliasing of the spurious counterexample that is being refined away, i.e., they only generalize for other spurious counterexamples that satisfy the same pointer aliasing equivalences. Although this results in fewer traces being removed from the abstraction, it allows the predicates used for refinement to be much more compact.

A side effect caused by the concrete execution driven refinement being specific to pointer aliasing is that the refinement predicate has to be evaluated for existing tests other than the one from which the test generation attempt was made, as they may fall onto either side of the split. In the YOGI tool's Dash implementation [72] this is handled by storing a tree of program states explored by the tests. To reduce the memory consumption from this strategy YOGI stores program states using a delta encoding, where only updates to the state are stored in the tree, together with a caching scheme to make state lookups efficient.

LCTD takes a different approach to the problem, by making the observation that for a large class of programs pointer aliasing only depends on the program path, which allows evaluating the predicates used for refinement using only concrete pointer values. This can be done despite the predicates containing references to other than pointer variables due to them acting as weakest preconditions for specific pointer aliasings. Thus, LCTD is able avoid storing anything other than pointers in the tree of states explored by concolic testing.

Array indexing in C is a common case of pointer usage where aliasing *is not* purely path dependent, which could be a problem for the state storage strategy used in LCTD. This problem is, however, sidestepped by lifting the handling of array indexing into the SMT solver using an encoding into the *theory of arrays* [81].

The use of underapproximation in reachability based branch elimination (RBBE) (Section 2.3.1) was directly inspired by the work on LCTD. However, RBBE only uses underapproximation once, while in Dash the under- and over-approximation are refined together.

3.2.1 Evaluations

In Publication IV, Section 5 LCTD is evaluated on a set of benchmark programs drawn from the 2015 international competition on software verification (SV-COMP) [10]. A comparison with results achieved by major tools in the 2015 SV-COMP shows that LCTD can achieve competitive results.

After the work in Publication IV, LCTD participated in the 2016 SV-COMP [11], where it did not replicate the results published in Publication IV. A major difference is that for the evaluation in Publication IV the input programs were hand-prepared for easy ingestion (which did not affect the difficulty of the core verification task), while for SV-COMP the tool had to support a wider variety of conventions in input programs. For future participation in SV-COMP, LCTD should be made more robust to various input programs.

Publication IV, Section 5 also includes a synthetic benchmark to show the

importance of checking the satisfiability of predicates created during refinement, and another benchmark to evaluate the theory of arrays based array indexing support.

3.3 Dynamic Partial Order Reduction with Concolic Testing

Concolic testing addresses testing of single-threaded programs by only running tests with input values that result in previously unexplored paths being covered. The execution of a multi-threaded program may also be affected by how operations from different threads are scheduled. However, some of these operations may be *independent* meaning that their executions do not affect each other. This results in many schedules being effectively equivalent due to them differing only on the ordering of independent operations. Such an equivalence class of schedules is called a Mazurkiewicz trace [64]. This is the analogue of the set of input values to a single-threaded program that cause tests to take the same path.

Intuitively, effective testing of multi-threaded programs requires an analogous approach to that of concolic testing, such that schedules belonging to different Mazurkiewicz traces can be systematically explored without enumerating all possible schedules. The work in this dissertation focuses on the dynamic partial order reduction (DPOR) algorithm [38], which implements such an approach. During a test execution DPOR identifies alternate scheduling decisions, called *backtracking points*, that will be explored in subsequent test executions. This way DPOR will explore at least one schedule from each Mazurkiewicz trace. The following example gives a high-level overview of DPOR and motivates the improvement to DPOR presented in this dissertation.

Example 17. Consider the following multi-threaded Java program:

```java
class RWExample {
    volatile int r = 0;
    public static void main(String[] args) {
        final RWExample shared = new RWExample();
        new Thread(() -> { // Thread A
            int local = shared.r;
            assert(local == 0);
        }).start();
        new Thread(() -> { // Thread B
            r = 1;
            int local = shared.r;
        }).start();
    }
}
```

A volatile integer `r` inside an instance of `RWExample` is used for shared state. The `main` function starts two threads: thread A, which first reads the shared variable `r` and then asserts that it is zero, and thread B, which first writes `1` into `r` and then also reads `r`. Intuitively, the assertion will hold or be violated depending on the scheduling of these operations:

- the assertion holds when the read in thread A is executed before the write in thread B, and
- the assertion is violated when the write in thread B is executed before the read in Thread A.

The rest of this example will walk through applying the original DPOR algorithm to this program.

The read in thread A will be referred to as *Ar*, and the write and read in thread B will be referred to as *Bw* and *Br*, respectively. DPOR starts with an initial arbitrary execution. Let this execution be [*Ar*,*Bw*,*Br*], in which the assertion holds due to *Ar* reading a value 0. During this execution DPOR will detect that *Ar* and *Bw* are dependent and concurrent. This results in a backtracking point being added to explore the execution where *Bw* is executed first.

Let the second execution be [*Bw*,*Ar*,*Br*], in which *Bw* has been forced as the first operation and the rest of the execution is arbitrary. In this execution the assertion is violated. During this execution DPOR will detect that *Ar* and *Br* are dependent (as they access the same shared variable) and concurrent, which results in a backtracking point for the execution starting with [*Bw*,*Br*].

The final execution is [*Bw*,*Br*,*Ar*], in which the assertion is again violated. This concludes the algorithm, with a total of three executions having been explored. ⊠

In the previous example the last backtracking point was added to explore another interleaving of *Ar* and *Br*, because the original DPOR algorithm considers them dependent due to them being operations on the same shared variable `r`. However, both operations only read the shared variable and they could be safely considered independent.

Flanagan and Godefroid [38] state that DPOR can be modified to treat reads on the same shared variable as independent. DPOR implements the tracking of which operations are concurrent using *vector clocks* [62] and how these are maintained during test executions is specific to the notion of dependency in use. Publication V presents an updated strategy for maintaining these vector clocks. The resulting algorithm, DPOR-CR, exploits the commutativity of read operations, i.e., the fact that reads on the same shared variable from different

threads are independent. This allows DPOR-CR to explore fewer executions, while still being guaranteed to find all assertion errors and deadlocks. For example, in Example 17 the last execution would not be explored, since it only reorders read operations. The assertion violation in thread A would still be found in the second execution.

Compared to the unmodified algorithm, DPOR-CR raises the worst-case time complexity for finding a backtracking point for a single thread from constant time to time linear in the length of the test execution. The best case remains the same and the evaluations did not encounter difficulties arising from the potential worst-case behavior. Please see Publication V, Section II.D for further details, including intuition for why the worst-case behavior is unlikely to be encountered.

The original work on DPOR by Flanagan and Godefroid [38] gives a guarantee that DPOR will reach all deadlocks and assertion errors. Publication VI expands this guarantee to data races, i.e., a situation where accesses to a shared variable are not properly protected by synchronization constructs. Specifically, Publication VI, Section 3 gives a proof that if there is a reachable data race then DPOR will explore a state where the racing operations are co-enabled. This property allows data races to be found during DPOR with negligible overhead.

For the work on DPOR in this dissertation, the concolic testing tool LCT [56] was extended with support for testing multi-threaded programs. This included adding support for threading and synchronization operations to the tool's Java instrumenter, as well as implementing runtime support for forcing schedules during test executions. The resulting version of LCT supports systematic testing of multi-threaded software that also uses inputs. The combination of concolic testing and DPOR explores a combined execution tree, which interleaves nodes produced by the respective algorithms. Specifically, concolic testing introduces data based branching decisions, while DPOR introduces concurrency based scheduling decisions. Please see Publication V, Section III for how DPOR and concolic testing are combined, and Section IV for a full discussion concerning the implementation of DPOR in LCT.

One notable feature of LCT that influences algorithm design, is that LCT is designed to be *distributed* in the sense that it can use multiple machines to run test executions in parallel. In addition to the improvements to DPOR, Publication V shows how to integrate a complimentary technique called *sleep sets* [40] into DPOR in the distributed context of LCT. A subtlety not mentioned in Publication V is how backtracking points added by DPOR must interact with sleep sets. Namely, an improper implementation of sleep sets might prevent

backtracking points from being explored if they happen to be in the sleep set, resulting in incompleteness. For a full discussion please see Kari Kähkönen's dissertation [55], end of Section 2.3.

3.3.1 Evaluation

In Publication V, Section V DPOR-CR is evaluated against the unmodified DPOR algorithm and another partial order reduction algorithm called "race detection and flipping", which is implemented by the jCUTE [78] tool. However, Kähkönen [55] notes that jCUTE fails to cover all Mazurkiewicz traces for certain types of programs. The evaluation shows that DPOR-CR shows significant reductions in the number of test executions explored over the original DPOR algorithm. Please see Publication V, Section V for descriptions of the benchmark programs, the evaluation data and further discussion.

3.4 Related Work

The approach taken in this dissertation to improving DPOR is that of tracking a more accurate dependency relation. For actor programs, TransDPOR [82] similarly extends DPOR to exploit the feature that actors do not share state, which ensures that the dependency relation for any set of co-enabled transitions is transitive. This allows TransDPOR to be lazier in adding transitions to backtracking points. For an even more fine-grained notion of dependency, *conditional dependency* is defined with respect to reachability, i.e., dependency takes data values into account [39].

The original DPOR algorithm provides a way to construct persistent sets dynamically, which allows the construction of smaller sets than techniques relying on static analysis. The reduction achieved by this approach is dependent on the order in which backtracking points are explored, and heuristics for good orders have been investigated [58]. For further reduction, DPOR is often combined with sleep sets, which block transitions to ensure that only one *complete* test is explored for each Mazurkiewicz trace [40]. However, even with minimal persistent sets combined with sleep sets the execution tree will contain partial executions that are later blocked by sleep sets [2]. *Source sets* are an alternative to persistent sets that solve this issue. By using them in conjunction with *wakeup trees* the Optimal DPOR algorithm [2, 4] is guaranteed to explore a minimal number of executions, while still covering all deadlocks and assertion errors. Another approach that offers exploring exactly one test for each Mazurkiewicz trace is

using *unfoldings*, which can accurately track the whole happens-before relation of a program [52]. This approach even allows exploring fewer tests than there are Mazurkiewicz traces if only local state reachability (e.g., assertion errors) is of interest. However, with unfoldings checking for global state reachability (e.g., deadlocks) requires additional work [75]. Kähkönen [55] also combines the unfolding based testing approach with concolic testing for efficient testing of multi-threaded programs with inputs. Concolic testing has also been combined with partial order reduction in the race detection and flipping algorithm [79], which is implemented in the CUTE and jCUTE testing tools [78]. A separate line of research on partial order reduction rests on the notion of *confluence* [16], which is a similar notion to that of independence [45].

Since 2012, research on practical software verification tools for C programs has seen significant focus at the yearly international competition on software verification (SV-COMP) [12]. The tool described in this dissertation, LCTD, participated in 2016 without reaching the results reported in Publication IV due to tooling issues [11]. For bug finding, i.e., finding an error in the subset of benchmarks that do contain an error, bounded model checking is a very effective approach. Prominent tools include the SAT based CBMC and the SMT based ESBMC. Ultimate Automizer [46] represents a recent approach to software verification, that has seen success overall and especially in termination checking. The automata based approach in Ultimate Automizer allows abstractions that are not tightly tied to the structure of the control flow graph. Other tools that provide good overall verification performance include CPAChecker [13], which provides a framework for easy integration of verification approaches and uses a combination of abstract domains for verification, and SMACK [22], which provides a framework for modular verification by translating LLVM intermediate representation into the Boogie verification language [8]. Please refer to Publication IV, Section 6 for a broader review of software verification techniques.

4. Conclusions

Symbolic methods use solvers to efficiently reason about groups of values. This dissertation has presented applications of symbolic methods in two areas: optimizations for stream processing as an application of symbolic transducers; and software verification and test generation, where concolic testing is the unifying theme. The symbolic representation in symbolic transducers allows compact representations even for transducers with large alphabets, while preserving important properties of classical transducers. Concolic testing provides a way to systematically explore program paths, which can be readily combined with other testing and verification methods. The design of these applications was supported by SMT solvers, which provide high-level interfaces to powerful symbolic reasoning facilities.

In the area of stream processing, symbolic methods support development of systems that let programmers use high-level abstractions to express their intent, while still allowing compilers to generate efficient code. The tool for compiling pipelines of symbolic transducers presented in Chapter 2 employs fusion as its main optimization, which allows pipelines composed of modular stages to match the performance of real-world hand-fused code. In addition to the pruning of unsatisfiable branches integrated into fusion, reachability based branch elimination (RBBE) and control state reduction (CSR) exploit contextual information exposed by fusion to provide additional reduction.

The backend of the tool produces efficient fused code, which is on average 3.4× faster than a suite of hand-written baselines. Both RBBE and CSR provide significant reduction, with the average number of branches removed by RBBE being 11% of the branches in the fused transducer and the corresponding average for control states removed by CSR being 25%[1].

For flexible specification of symbolic transducers, several frontends have been

[1] As both RBBE and CSR are applied after every step of fusion, an unreduced fused transducer could be larger than these numbers would suggest.

developed. XPath and Regex based frontends target pattern matching in XML and lightweight text formats, respectively. Symbolic transducers may also be automatically generated from input data, as is done in the frontends for Huffman encoding and decoding. Finally, the general-purpose frontend translates imperative C# code into symbolic transducers.

The translation from C# to transducers uses symbolic execution, which in this dissertation is also used in its more familiar field of automated test generation. The LCTD tool presented in this dissertation implements the Dash [9] algorithm, which combines *concolic testing*, an approach to implementing efficient symbolic execution, with a predicate abstraction. The combination of over- and underapproximation served as direct inspiration to the usage of underapproximation in RBBE. With LCTD the Dash algorithm is instantiated for programs in the LLVM internal representation and a technique to avoid storing complete program states is presented. With an improvement to how the abstraction is refined LCTD is able to verify complex programs from the SV-COMP benchmark suite [10].

This dissertation also combines concolic testing with dynamic partial order reduction (DPOR) [38], a reduction method for exploring reduced sets of interleavings for multi-threaded programs. DPOR is guaranteed to cover all reachable deadlocks and assertion errors, and in this dissertation has also been retrofitted for race detection. A new version of the algorithm called DPOR-CR that exploits the commutativity of read operations is also presented and is shown to achieve significant reductions in the number of interleavings explored over the original DPOR algorithm. The implementations are extensions to LCT, an open source concolic testing tool for sequential Java programs.

Future Work

Branching symbolic transducers (BSTs) extend symbolic transducer (ST) by introducing *branches* as a control flow construct. This raises the question of what other control flow constructs would be useful. For example, allowing outputs and register updates also inside the tree structure of rules, instead of just on the leaves, would enable more sharing with no immediately apparent downsides. On the other hand, whether supporting loops in rules is a good idea is less clear, as this would at least change the pruning inside fusion from a satisfiability query to a reachability one. Allowing *epsilon transitions*, i.e., transitions that do not consume input, would have similar considerations.

Currently fusion is applied indiscriminately until a single transducer remains.

However, this might not always be optimal, for example if the size of the resulting transducer blows up. This can cause inconveniently long time being spent in fusion and may not even be ideal for performance when the amount of code being generated causes increased CPU instruction cache misses. In a parallel setting fusion can also reduce *pipeline parallelism*, while stages in an unfused pipeline can be run in parallel. It would therefore be valuable to explore *when does fusion actually pay off*. A simple strategy is to block fusion using a hard limit for the size of the resulting transducer. This is similar to a common strategy used in traditional compilers, such as LLVM, to decide when to apply the similar optimization of function inlining.

If fusion were not always applied, it would be useful for RBBE to exploit context from the surrounding pipeline. This would still allow eliminating branches that are not reachable in any context. This work is likely to benefit from techniques similar to interprocedural analysis in software verification.

The current reachability analysis employed by RBBE is not structured as an abstraction refinement loop. A more efficient analysis that would allow faster reduction for larger transducers could likely be achieved by adapting a modern reachability analysis algorithm, such as Dash or IC3/PDR [19, 25], for branch elimination. A slight difference between RBBE and software verification is that in RBBE it may be more appropriate to trade completeness for improved runtime: in software verification failing to prove the unreachability of a bug precludes a proof of safety, while in RBBE not being able to eliminate a branch might not have any significant effects.

The frontends for specifying BSTs are in this dissertation exposed as something users can *explicitly* adopt to benefit from fusion. However, automatically detecting pieces of code that correspond to transductions and serial compositions thereof would allow the techniques presented in this dissertation to be applied *implicitly*, thus also allowing legacy codebases to benefit from fusion.

Since the work in this dissertation, an improved DPOR algorithm called Optimal DPOR [4] was developed, which provides a more current starting point for future research. Optimal DPOR guarantees that only one test is executed per each Mazurkiewicz trace. However, the proof in Publication VI that data races can be found simply by checking for racing co-enabled operations uses the property that DPOR explores a *persistent set* of operations from each state, while Optimal DPOR uses slightly different notion of partial order reduction called *source sets* and does not always explore a persistent set from each state. To use the same method for checking for data races with Optimal DPOR, the proof of Theorem 1 in Publication VI should be restated using source sets if possible.

References

[1] P. A. Abdulla, L. Kaati, and J. Högberg. Bisimulation minimization of tree automata. In *Proceedings of the 11th International Conference on Implementation and Application of Automata (CIAA 2006)*, pages 173–185. Springer, 2006. https://doi.org/10.1007/11812128_17.

[2] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st Symposium on Principles of Programming Languages (POPL 2014)*, pages 373–384. ACM, 2014. https://doi.org/10.1145/2535838.2535845.

[3] P. A. Abdulla, Y.-F. Chen, L. Holík, and T. Vojnar. Mediating for reduction (on minimizing alternating Büchi automata). *Theoretical Computer Science*, 552(2): 26–43, Oct. 2014. https://doi.org/10.1016/j.tcs.2014.08.003.

[4] P. A. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Source sets: A foundation for optimal dynamic partial order reduction. *Journal of the ACM*, 64(4):25:1–25:49, Aug. 2017. https://doi.org/10.1145/3073408.

[5] S. Agrawal, W. Thies, and S. Amarasinghe. Optimizing stream programs using linear state space analysis. In *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES 2005)*, pages 126–136. ACM, 2005. https://doi.org/10.1145/1086297.1086315.

[6] A. Alexandrov, R. Bergmann, S. Ewen, J.-C. Freytag, F. Hueske, A. Heise, O. Kao, M. Leich, U. Leser, V. Markl, F. Naumann, M. Peters, A. Rheinländer, M. J. Sax, S. Schelter, M. Höger, K. Tzoumas, and D. Warneke. The Stratosphere platform for big data analytics. *The VLDB Journal*, 23(6):939–964, Dec. 2014. https://doi.org/10.1007/s00778-014-0357-y.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in Spark. In *Proceedings of the 2015 International Conference on Management of Data (SIGMOD 2015)*, pages 1383–1394. ACM, 2015. https://doi.org/10.1145/2723372.2742797.

[8] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Proceedings of the 4th International Conference on Formal Methods for Components and Objects (FMCO 2005)*, pages 364–387. Springer, 2006. https://doi.org/10.1007/11804192_17.

[9] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *Proceedings of the 2008 International Symposium on Software Testing and*

Analysis (ISSTA 2008), pages 3–14. ACM, 2008. https://doi.org/10.1145/1390630.1390634.

[10] D. Beyer. Software verification and verifiable witnesses (report on SV-COMP 2015). In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2015)*, pages 401–416. Springer, 2015. https://doi.org/10.1007/978-3-662-46681-0_31.

[11] D. Beyer. Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016). In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2016)*, pages 887–904. Springer, 2016. https://doi.org/10.1007/978-3-662-49674-9_55.

[12] D. Beyer. Software verification with validation of results (report on SV-COMP 2017). In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2017)*, pages 331–349. Springer, 2017. https://doi.org/10.1007/978-3-662-54580-5_20.

[13] D. Beyer and M. E. Keremoglu. CPACHECKER: A tool for configurable software verification. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV 2011)*, pages 184–190. Springer, 2011. https://doi.org/10.1007/978-3-642-22110-1_16.

[14] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of the 36th Annual Design Automation Conference (DAC 1999)*, pages 317–320. ACM, 1999. https://doi.org/10.1145/309847.309942.

[15] D. L. Bird and C. U. Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, Sept. 1983. https://doi.org/10.1147/sj.223.0229.

[16] S. Blom and J. van de Pol. State space reduction by proving confluence. In *Proceedings of the 14th International Conference on Computer Aided Verification (CAV 2002)*, pages 596–609. Springer, 2002. https://doi.org/10.1007/3-540-45657-0_50.

[17] N. Blum. An o(n log n) implementation of the standard method for minimizing n-state finite automata. *Information Processing Letters*, 57(2):65–69, Jan. 1996. https://doi.org/10.1016/0020-0190(95)00199-9.

[18] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996. https://doi.org/10.1109/12.537122.

[19] A. R. Bradley. SAT-based model checking without unrolling. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, pages 70–87. Springer, 2011. https://doi.org/10.1007/978-3-642-18275-4_7.

[20] D. Buchfuhrer and C. Umans. The complexity of Boolean formula minimization. *Journal of Computer and System Sciences*, 77(1):142–153, 2011. https://doi.org/10.1016/j.jcss.2010.06.011.

[21] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2): 142–170, June 1992. https://doi.org/10.1016/0890-5401(92)90017-A.

[22] M. Carter, S. He, J. Whitaker, Z. Rakamarić, and M. Emmi. SMACK software verification toolchain. In *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*, pages 589–592. ACM, 2016. https://doi.org/10.1145/2889160.2889163.

[23] R. Chaiken, B. Jenkins, P.-A. Larson, B. Ramsey, D. Shakib, S. Weaver, and J. Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment*, 1(2):1265–1276, Aug. 2008. https://doi.org/10.14778/1454159.1454166.

[24] C. Choffrut. *Contributions à l'étude de quelques familles remarquables de fonctions rationnelles*. PhD thesis, Université Paris 7, LITP, Paris, France, 1978.

[25] A. Cimatti and A. Griggio. Software model checking via IC3. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV 2012)*, pages 277–293. Springer, 2012. https://doi.org/10.1007/978-3-642-31424-7_23.

[26] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5): 752–794, Sept. 2003. https://doi.org/10.1145/876638.876643.

[27] E. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of the 40th Annual Design Automation Conference (DAC 2003)*, pages 368–371. ACM, 2003. https://doi.org/10.1145/775832.775928.

[28] T. Colcombet and P. Fradet. Enforcing trace properties by program transformation. In *Proceedings of the 27th Symposium on Principles of Programming Languages (POPL 2000)*, pages 54–66. ACM, 2000. https://doi.org/10.1145/325694.325703.

[29] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: From lists to streams to nothing at all. In *Proceedings of the 12th International Conference on Functional Programming (ICFP 2007)*, pages 315–326. ACM, 2007. https://doi.org/10.1145/1291151.1291199.

[30] J. Damji. A tale of three Apache Spark APIs: RDDs, DataFrames, and Datasets. https://databricks.com/blog/2016/07/14/, July 2016.

[31] L. D'Antoni and M. Veanes. Minimization of symbolic automata. In *Proceedings of the 41st Symposium on Principles of Programming Languages (POPL 2014)*, pages 541–553. ACM, 2014. https://doi.org/10.1145/2535838.2535849.

[32] L. D'Antoni and M. Veanes. Extended symbolic finite automata and transducers. *Formal Methods in System Design*, 47(1):93–119, Aug. 2015. https://doi.org/10.1007/s10703-015-0233-4.

[33] L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proceedings of 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, pages 337–340. Springer, 2008. https://doi.org/10.1007/978-3-540-78800-3_24.

[34] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, Jan. 2008. https://doi.org/10.1145/1327452.1327492.

[35] D. Debarbieux, O. Gauwin, J. Niehren, T. Sebastian, and M. Zergaoui. Early nested word automata for XPath query answering on XML streams. *Theoretical Computer Science*, 578(C):100–125, May 2015. https://doi.org/10.1016/j.tcs.2015.01.017.

[36] S. Drobac, K. Lindén, T. A. Pirinen, and M. Silfverberg. Heuristic hyperminimization of finite state lexicons. In *Proceedings of the Ninth International Conference on Language Resources and Evaluation (LREC 2014)*, Reykjavik, Iceland, May 2014.

[37] J. Edvardsson. A survey on automatic test data generation. In *Proceedings of the 2nd Conference on Computer Science and Engineering*, pages 21–28, 1999.

[38] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL 2005)*, pages 110–121. ACM, 2005. https://doi.org/10.1145/1040305.1040315.

[39] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*, volume 1032 of *LNCS*. Springer, 1996. https://doi.org/10.1007/3-540-60761-7.

[40] P. Godefroid, G. J. Holzmann, and D. Pirottin. State-space caching revisited. *Formal Methods in System Design*, 7(3):227–241, Nov. 1995. https://doi.org/10.1007/BF01384077.

[41] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, pages 213–223. ACM, 2005. https://doi.org/10.1145/1065010.1065036.

[42] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox fuzzing for security testing. *Queue*, 10(1):20–27, Jan. 2012. https://doi.org/10.1145/2090147.2094081.

[43] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV 1997)*, pages 72–83. Springer, 1997. https://doi.org/10.1007/3-540-63166-6_10.

[44] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A new algorithm for property checking. In *Proceedings of the 14th International Symposium on Foundations of Software Engineering (FSE 2014)*, pages 117–127. ACM, 2006. https://doi.org/10.1145/1181775.1181790.

[45] H. Hansen and M. Timmer. A comparison of confluence and ample sets in probabilistic and non-probabilistic branching time. *Theoretical Computer Science*, 538 (12):103–123, June 2014. https://doi.org/10.1016/j.tcs.2013.07.014.

[46] M. Heizmann, J. Hoenicke, and A. Podelski. Software model checking for people who love automata. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV 2013)*, pages 36–52. Springer, 2013. https://doi.org/10.1007/978-3-642-39799-8_2.

[47] M. Hirzel, R. Soulé, S. Schneider, B. Gedik, and R. Grimm. A catalog of stream processing optimizations. *ACM Computing Surveys*, 46(4):46:1–46:34, Mar. 2014. https://doi.org/10.1145/2528412.

[48] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *Proceedings of the 12th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2011)*, pages 248–262. Springer, 2011. https://doi.org/10.1007/978-3-642-18275-4_18.

[49] P. Hooimeijer, B. Livshits, D. Molnar, P. Saxena, and M. Veanes. Fast and precise sanitizer analysis with BEK. In *Proceedings of the 20th Conference on Security (SEC 2011)*. USENIX Association, 2011.

[50] D. A. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952. https://doi.org/10.1109/JRPROC.1952.273898.

[51] F. Ivančić, Z. Yang, M. K. Ganai, A. Gupta, I. Shlyakhter, and P. Ashar. F-SOFT: Software verification platform. In *Proceedings of the 17th International Conference on Computer Aided Verification (CAV 2005)*, pages 301–306. Springer, 2005. https://doi.org/10.1007/11513988_31.

[52] K. Kähkönen, O. Saarikivi, and K. Heljanko. Using unfoldings in automated testing of multithreaded programs. In *Proceedings of the 27th International Conference on Automated Software Engineering (ASE 2012)*, pages 150–159. ACM, 2012. https://doi.org/10.1145/2351676.2351698.

[53] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. https://doi.org/10.1145/360248.360252.

[54] G. Kovásznai, A. Fröhlich, and A. Biere. Complexity of fixed-size bit-vector logics. *Theory of Computing Systems*, 59(2):323–376, Aug. 2016. https://doi.org/10.1007/s00224-015-9653-1.

[55] K. Kähkönen. *Automated Systematic Testing Methods for Multithreaded Programs*. PhD thesis, Aalto University, Helsinki, Finland, 2015. URL http://urn.fi/URN:ISBN:978-952-60-6039-2.

[56] K. Kähkönen, T. Launiainen, O. Saarikivi, J. Kauttio, K. Heljanko, and I. Niemelä. LCT: An open source concolic testing tool for Java programs. In *Proceedings of the 6th Workshop on Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE 2011)*, pages 75–80. Elsevier, 2011.

[57] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO 2004)*, pages 75–86. IEEE, Mar. 2004. https://doi.org/10.1109/CGO.2004.1281665.

[58] S. Lauterburg, R. K. Karmani, D. Marinov, and G. Agha. Evaluating ordering heuristics for dynamic partial-order reduction techniques. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE 2010)*, pages 308–322. Springer, 2010. https://doi.org/10.1007/978-3-642-12029-9_22.

[59] J. Le Dem and N. Li. Efficient data storage for analytics with Apache Parquet 2.0. https://www.slideshare.net/cloudera/hadoop-summit-36479635, June 2014.

[60] T. Lipcon, D. Alves, D. Burkert, J.-D. Cryans, A. Dembo, M. Percy, S. Rus, D. Wang, M. Bertozzi, C. P. McCabe, and A. Wang. Kudu: Storage for fast analytics on fast data. `http://kudu.apache.org/kudu.pdf`, Sept. 2015.

[61] A. Maletti, J. Graehl, M. Hopkins, and K. Knight. The power of extended top-down tree transducers. *SIAM Journal on Computing*, 39(2):410–430, June 2009. `https://doi.org/10.1137/070699160`.

[62] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226. North-Holland, 1989.

[63] R. Mayr and L. Clemente. Advanced automata minimization. In *Proceedings of the 40th Annual Symposium on Principles of Programming Languages (POPL 2013)*, pages 63–74. ACM, 2013. `https://doi.org/10.1145/2429069.2429079`.

[64] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324. Springer, 1987. `https://doi.org/10.1007/3-540-17906-2_30`.

[65] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1992. UMI Order No. GAX92-24209.

[66] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. Dremel: Interactive analysis of web-scale datasets. *Proceedings of the VLDB Endowment*, 3(1-2):330–339, Sept. 2010. `https://doi.org/10.14778/1920841.1920886`.

[67] S. Mesfar and M. Silberztein. Transducer minimization and information compression for NooJ dictionaries. In *Proceedings of the 2009 Conference on Finite-State Methods and Natural Language Processing: Post-proceedings of the 7th International Workshop (FSMNLP 2008)*, pages 110–121. IOS Press, 2009. `https://doi.org/10.3233/978-1-58603-975-2-110`.

[68] T. Milo, D. Suciu, and V. Vianu. Typechecking for XML transformers. *Journal of Computer and System Sciences*, 66(1):66–97, Feb. 2003. `https://doi.org/10.1016/S0022-0000(02)00030-2`.

[69] M. Mohri. Minimization of sequential transducers. In *Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching (CPM 1994)*, pages 151–163. Springer, 1994. `https://doi.org/10.1007/3-540-58094-8_14`.

[70] B. Mozafari, K. Zeng, L. D'Antoni, and C. Zaniolo. High-performance complex event processing over hierarchical data. *ACM Transactions on Database Systems*, 38(4):21:1–21:39, Dec. 2013. `https://doi.org/10.1145/2536779`.

[71] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proceedings of the VLDB Endowment*, 4(9):539–550, June 2011. `https://doi.org/10.14778/2002938.2002940`.

[72] A. V. Nori and S. K. Rajamani. An empirical study of optimizations in YOGI. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010) - Volume 1*, pages 355–364. ACM, 2010. `https://doi.org/10.1145/1806799.1806852`.

[73] K. Ousterhout, R. Rasti, S. Ratnasamy, S. Shenker, and B.-G. Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th Conference on Networked Systems Design and Implementation (NSDI 2015)*, pages 293–307. USENIX Association, 2015.

[74] M. Rys. U-SQL query execution and performance tuning. `https://www.slideshare.net/MichaelRys/usql-query-execution-and-performance-tuning`, Feb. 2016.

[75] O. Saarikivi, H. Ponce De León, K. Kähkönen, K. Heljanko, and J. Esparza. Minimizing test suites with unfoldings of multithreaded programs. *ACM Transactions on Embedded Computing Systems*, 16(2):45:1–45:24, Feb. 2017. `https://doi.org/10.1145/3012281`.

[76] S. Schneider, M. Hirzel, B. Gedik, and K.-L. Wu. Auto-parallelizing stateful distributed streaming applications. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT 2012)*, pages 53–64, New York, NY, USA, 2012. ACM. `https://doi.org/10.1145/2370816.2370826`.

[77] K. Sen. *Scalable Automated Methods for Dynamic Program Analysis*. PhD thesis, University of Illinois at Urbana-Champaign, Champaign, IL, USA, 2006. AAI3242987.

[78] K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV 2006)*, pages 419–423. Springer, 2006. `https://doi.org/10.1007/11817963_38`.

[79] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Proceedings of the 2nd International Haifa Verification Conference on Hardware and Software, Verification and Testing (HVC 2006)*, pages 166–182. Springer, 2007. `https://doi.org/10.1007/978-3-540-70889-6_13`.

[80] J. H. Spring, J. Privat, R. Guerraoui, and J. Vitek. Streamflex: High-throughput stream programming in Java. In *Proceedings of the 22nd Annual Conference on Object-oriented Programming Systems and Applications (OOPSLA 2007)*, pages 211–228. ACM, 2007. `https://doi.org/10.1145/1297027.1297043`.

[81] A. Stump, C. W. Barrett, D. L. Dill, and J. Levitt. A decision procedure for an extensional theory of arrays. In *Proceedings of the 16th Annual Symposium on Logic in Computer Science (LICS 2001)*, pages 29–37. IEEE, 2001. `https://doi.org/10.1109/LICS.2001.932480`.

[82] S. Tasharofi, R. K. Karmani, S. Lauterburg, A. Legay, D. Marinov, and G. Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the 32nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems (FORTE 2012)*, pages 219–234. Springer, 2012. `https://doi.org/10.1007/978-3-642-30793-5_14`.

[83] W. Thies, M. Karczmarek, and S. P. Amarasinghe. StreamIt: A language for streaming applications. In *Proceedings of the 11th International Conference on Compiler Construction (CC 2002)*, pages 179–196. Springer, 2002. `https://doi.org/10.1007/3-540-45937-5_14`.

[84] T. van Dijk, A. Laarman, and J. van de Pol. Multi-core BDD operations for symbolic reachability. In *Proceedings of the 11th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2012)*, pages 127–143. Elsevier, 2012. https://doi.org/10.1016/j.entcs.2013.07.009.

[85] M. Veanes, P. Hooimeijer, B. Livshits, D. Molnar, and N. Bjorner. Symbolic finite state transducers: Algorithms and applications. In *Proceedings of the 39th Annual Symposium on Principles of Programming Languages (POPL 2012)*, pages 137–150. ACM, 2012. https://doi.org/10.1145/2103656.2103674.

[86] M. Veanes, T. Mytkowicz, D. Molnar, and B. Livshits. Data-parallel string-manipulating programs. In *Proceedings of the 42nd Annual Symposium on Principles of Programming Languages (POPL 2015)*, pages 139–152. ACM, 2015. https://doi.org/10.1145/2676726.2677014.

[87] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, Jan. 1988. https://doi.org/10.1016/0304-3975(90)90147-A.

[88] T. White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1st edition, 2009. ISBN 978-1449311520.

[89] Y. Xie and A. Aiken. Scalable error detection using Boolean satisfiability. In *Proceedings of the 32nd Symposium on Principles of Programming Languages (POPL 2005)*, pages 351–363. ACM, 2005. https://doi.org/10.1145/1040305.1040334.

[90] R. Xin and J. Rosen. Project Tungsten: Bringing Apache Spark closer to bare metal. https://databricks.com/blog/2015/04/28, Apr. 2015.

[91] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2nd Conference on Hot Topics in Cloud Computing (HotCloud 2010)*, pages 10–10. USENIX Association, 2010.

Errata

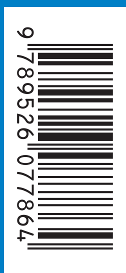
Publication III

Example 8 in Section 6 fails to demonstrate a case where register invariant based strengthening adds reduction. The strengthening in the example does not achieve indistinguishability for states q_1 and q_2, because after the transformation the strengthened outgoing transitions $q_1 \xrightarrow{y \geq 3\,/\,[y \geq 3];0} q_3$ and $q_2 \xrightarrow{y<3\,/\,[y<3];0} q_3$ will no longer accept the same register values.

The strengthening approach as it is presented in Section 6 does still provide reduction for sets of control states which are only distinguishable due to behavior on unreachable register values.

Symbolic methods use solvers to efficiently reason about groups of values. In stream processing, techniques based on symbolic transducers unlock advanced optimizations than can greatly increase throughput. Dynamic symbolic execution, a.k.a. concolic testing, is a way of systematically exploring program behavior, which can be combined with predicate abstraction to automatically prove program correctness or with partial-order reduction for efficient testing of concurrent programs. The symbolic methods presented in this dissertation make extensive use of modern solvers, which provide high-level interfaces to powerful symbolic reasoning facilities.

Aalto-DD 255/2017



ISBN 978-952-60-7786-4 (printed)
ISBN 978-952-60-7787-1 (pdf)
ISSN-L 1799-4934
ISSN 1799-4934 (printed)
ISSN 1799-4942 (pdf)

Aalto University
School of Science
Department of Computer Science
www.aalto.fi

BUSINESS +
ECONOMY

ART +
DESIGN +
ARCHITECTURE

SCIENCE +
TECHNOLOGY

CROSSOVER

DOCTORAL
DISSERTATIONS