Checking Format Compatibility of Programs Using Automata

By

Evan E. Driscoll

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2013

Date of final oral examination: 8/21/2013

The dissertation is approved by the following members of the Final Oral Committee:
Thomas W. Reps, Professor, Computer Sciences
Susan B. Horwitz, Professor, Computer Sciences
Somesh Jha, Professor, Computer Sciences
Benjamin R. Liblit, Associate Professor, Computer Sciences
Steffen Lempp, Professor, Mathematics

© Copyright by Evan E. Driscoll 2013
All Rights Reserved

Dedicated to my family

Acknowledgments

I owe many people for their help on the way to creating this dissertation. The following acknowledges many, but not all, of the people who have helped me along the line.

It has been a pleasure to work alongside Tom Reps during my time in Madison. I have learned much from him and being a part of his group. His encouragement as well is appreciated, especially during the point at which I was seriously considering leaving graduate school to work for industry. Tom: it has been a pleasure, and I'm sure the next few years will be equally illuminating and interesting.[1]

I would also like to thank the other members of my committee. My work on the topics in the thesis began with Susan Horwitz; at a time when she and Tom were on sabbatical and I remained in Madison, Susan and I had regular phone calls talking about some of the early incarnations of this work as I (and Susan and Tom) were still figuring out what we wanted to do. Ben Liblit, in addition to kindly participating on my committee, has helped me with many questions through my years; and Ben's class on software artifacts was one of the top couple of CS classes I have had the pleasure of taking. (In fact, I first learned about Daikon, which I use in this thesis, from Ben's class.) Somesh Jha and Steffen Lempp also have taught very interesting classes, and Somesh in particular has provided valuable feedback when my work was in earlier stages.

The other students in Tom's research group have been equally important to me during my time in Madison. Denis Gopan, Nick Kidd, Akash Lal, Junghee Lim, and Gogul Balakrishnan were around for at least my first year, and I fondly remember the occasional ice cream trips to Union South and the "PL group" chess night that we had a few times. Nick Kidd in particular deserves a special callout for creating and being the principal author of the WALi library, which I have made extensive use of and modifications to. Amanda Burton, one of Tom's later students, also deserves a callout for a similar reason: Amanda

[1]Nevertheless, I think I am obligated to throw that contraction in here, for old times' sake. ☺

was the original author of the NWA code that I eventually turned into the OpenNWA library, and the fact that this dissertation has a section talking about OpenNWA should not be taken as me wanting to detract from her contribution. Prathmesh Prabhu, Emma Turetsky, and Aditya Thakur have also made substantial contributions to the library, and I have worked closely with them on it. For the last couple of years, Adytia also has been my go-to person basically any time when I want to ask about different options for coding something, how to solve a certain problem, or just about anything. Rich Joiner was the first user of OpenNWA who was not actively involved in its development (though he made some helpful additions after being a user), and having an independent eye on the code was very helpful. Venkatesh Srinivasan, Tushar Sharma, Divy Vasal, Matt Elder, and Bill Harris have all given valuable feedback about one thing or another that I have been working on while in Madison, and most have taught me a great deal. Matt and Bill in particular are good friends.

This work was supported by NSF under grants CCF-0540955, 0810053, 0904371; by ONR under grants N00014-07-M-0407, 09-1-0510, 10-M-0251; by ARL under grant W911NF-09-1-0413; by AFRL under grants FA9550-09-1-0279 and FA8650-10-C-7088; and by DARPA under cooperative agreement HR0011-12-2-0012. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsoring agencies.

Lastly I would like to thank my family for their support. Both my immediate family, Sally and Giles Driscoll and my brother Eric, as well as my extended family have been very supportive of my time at UW, and I almost never heard a “when are you graduating?!” out of them. Anyone would be fortunate to have parents like mine.

Contents

Contents **v**

List of Figures, Tables, and Listings **ix**

Abstract **xi**

1 Introduction **1**

1.1 File Formats *2*

1.1.1 Surface syntax of the ICO format *3*

1.1.2 Semantic constraints on the input *4*

1.1.3 Input semantics *6*

1.2 Format Inference and Compatibility Checking *6*

1.2.1 Format inference *6*

1.2.2 Modeling programs vs modeling formats *8*

1.2.3 Format compatibility *8*

1.3 Dissertation Overview *10*

Part I Tools

2 Nested-Word Automata (NWAs) **15**

2.1 Definitions and Theory *15*

2.1.1 Informal Description of NWAs [background] *17*

2.1.2 Formal Definition of NWAs [background] *20*

2.1.3 NWAs and ε transitions [new] *22*

2.1.4 NWA determinization [adaption/background] *28*

2.1.5 Kleene star [adaption] *32*

2.1.6 Weakly-hierarchical-preserving NWA Reversal [new] *34*

2.2 The OpenNWA Library [new] *39*

2.2.1 Supported Operations *39*

2.2.2 Client Information *40*

2.2.3 Inter-operability with WPDSs *40*

2.2.4 OpenNWA uses *41*

2.3 Related Work *43*

3 Extended Finite Automata (XFAs) and Weighted Finite Automata (WFAs) 45

3.1 Formal definition of extended finite automata (XFA) [background] *46*

3.2 Weighted finite automata [background] *48*

3.3 Interpreting an XFA as a WFA [adaption/new] *50*

3.4 Symbolic ε closure [new/adaption] *53*

3.4.1 Performance comparison of ε closure methods *58*

3.5 State-Determinization [adaption/background] *59*

3.5.1 Lifting the data set from D to Q × D and adjusting transformers *64*

3.5.2 Determinize the state portion of the WFA *67*

3.6 Language containment *69*

3.6.1 The powerset semiring [background and new] *70*

3.6.2 WFA universality [new] *71*

3.6.3 WFA cross product [adaption/new] *74*

3.6.4 Basic inclusion test [new] *75*

3.6.5 Speeding up operations with antichains [as indicated] *75*

3.6.6 Complexity of XFA universality and inclusion testing [new] *81*

3.7 Representing relations as BDDs [background] *82*

3.7.1 A brief introduction to BDDs *82*

3.7.2 Representing non-Boolean functions and relations with a BDD *86*

3.8 XFA implementation in WALi [new] *87*

4 A Binary Front End for Daikon 89

4.1 Daikon background *89*

4.2 Snotra: a new Daikon front end *90*

4.2.1 Motivation *93*

4.2.2 How Snotra works *96*
4.2.3 Instrumentation for field values *97*
4.2.4 Instrumentation for loop trip counts *100*
4.2.5 Example instrumentation *106*

Part II Application Compatibilty Checking

5 Control-Flow Format Compatibility **111**
5.1 Overview *111*
5.2 Building FA and NWA Models of a Program *114*
5.2.1 Knowledge about I/O procedures *117*
5.2.2 Benefits of Using NWAs *118*
5.3 Enriching NWAs for Compatibility *121*
5.4 Using PCCA for more than types *124*
5.5 PCCA Implementation *125*
5.5.1 Seeding the System with I/O Procedures *127*
5.5.2 Removing Irrelevant Procedures *128*
5.6 Experiments *128*

6 Adding Loop Counters With XFAs **135**
6.1 Overview *136*
6.2 Inferring Format Models *138*
6.2.1 Modeling control flow *138*
6.2.2 Finding I/O relations *140*
6.2.3 Modeling I/O relations *140*
6.3 Optimizations *143*
6.3.1 Setting killed variables to a single value *143*
6.3.2 Collapsing ε sequences *145*
6.4 Tradeoffs with our instrumentation strategy *146*
6.5 Experiments *147*
6.5.1 XFA benefits evaluation *148*
6.5.2 Synthetic performance-scaling evaluation *154*
6.5.3 ICO specification performance scaling evaluation *156*

7 Related Work **161**

8 Conclusions **169**

Bibliography **175**

List of Figures, Tables, and Listings

Figure 1.1 The ICO file format 2
Table 1.2 Sizes and equivalences of “C” types and Windows types 4
Figure 1.3 True and false positives and negatives 9

Figure 2.1 An example program, NWA, and NWA run 19
Figure 2.2 Rules defining transitions *added* to an NWA during ε closure 23
Figure 2.3 Diagram of ε removal 23
Figure 2.4 Example invalid $^?$NWA constructed from a grammar 25
Figure 2.5 Invalid context-free grammar to $^?$NWA construction rules 26
Figure 2.6 Diagram of translating between ε semantics 28
Figure 2.7 Diagram of NWA Kleene star construction. 34
Figure 2.8 Rules defining transitions created by Kleene star 35
Figure 2.9 Rules defining transitions created during NWA reversal 36

Figure 3.1 Factoring a relation 54
Listing 3.2 Smith et al.'s ε-closure procedure 54
Figure 3.3 Solving WFA ε closure via translation to an (F)WPDS 56
Table 3.4 ε-closure algorithm performance 59
Figure 3.5 Example XFA with nondeterminism in the data value 61
Figure 3.6 Example XFA and determinization 62
Figure 3.7 Illustration of why the third step of XFA determinization, lifting transformers to the power set, is necessary 71
Listing 3.8 The non-antichains, early-cutoff universality algorithm for FAs. 76
Figure 3.9 Illustration of the benefit of antichains in universality checking as well as cutting off the search early 78
Figure 3.10 Transforming an FA into an XFA. 82

Figure 4.1 Diagram of Daikon's operation, including example program and invariants 91
Figure 4.2 Example declarations file and trace file from Daikon 92
Figure 4.3 A block diagram of Snotra's operation. 96
Figure 4.4 Splitting a single source-level `while` loop into two natural loops. 103
Figure 4.5 Irreducible control flow 105
Figure 4.6 Original program 107
Figure 4.7 Example program instrumented by Snotra 108

Listing 5.1 Example producer 112
Listing 5.2 Example consumer 112
Listing 5.3 Example consumer 112
Figure 5.4 Inferred automata for Listings 5.1 and 5.2 116
Listing 5.5 Components that illustrate the benefits of NWAs 121
Figure 5.6 Illustration of a missing security check (`securityManager.checkAccept`) in the Apache Harmony library 126
Table 5.7 PCCA experiments 129
Figure 5.8 The specification of *gzip*'s header format 131

Figure 6.1 Example producer and consumer 136
Figure 6.2 Example program to read a simple image format 136
Figure 6.3 The XFA read gadget 139
Figure 6.4 XFA state transition procedure δ and data relations U 140
Table 6.5 XFA ICO specification vs. specification experiments. 151
Table 6.6 *png2ico* vs. *png2ico* experiments. 154
Figure 6.7 Fixing the size of each logical variable to 3 bits, a chart of the effect of the number of concatenated loops. 157
Figure 6.8 Fixing the size of each logical variable to just 1 bit, a chart of the effect of the number of concatenated loops 157
Figure 6.9 Fixing the number of concatenated cycles to 2, a chart of the effect of the number of bits 158
Figure 6.10 Fixing the number of concatenated cycles to 3, a chart of the effect of the number of bits 158
Table 6.11 ICO specification performance scaling tests 159

Abstract

This dissertation describes methods for automatically analyzing programs to determine compatibility of software components. Complex systems today are made up of many communicating programs or program components. It is vitally important to ensure that the messages that one component sends to another are understood by the receiving component, otherwise runtime errors will occur.

The techniques described model two software components that are designed to work together, one as a producer of messages and one as a consumer of them, using three forms of automata from formal language theory. Each model's language is an approximation of the messages that the underlying component can either write (for the producer) or read (for the consumer). Once the models are created, they can be checked for compatibility by testing whether the language of the producer's model is a subset of the language of the consumer's. A counterexample to language inclusion represents a message that the producer is able to emit that the consumer is not prepared to accept (or perhaps a spurious counterexample due to the approximation).

We looked at three forms of automata to play the role of the program models: standard finite automata, nested-word automata (NWAs, originally defined by Alur and Madhusudan), and extended finite automata (XFAs, originally defined by Smith, Estan, and Jha). NWAs and XFAs both bring separate precision benefits to the table, letting us model the programs and their formats more precisely.

As part of the dissertation, we also make several new theoretical contributions to both NWAs and XFAs. For example, for NWAs we found an easy-to-correct but significant error in Alur and Madhusudan's description of the the Kleene star construction, and we describe new issues related to introducing epsilon transitions to NWAs. We view XFAs as weighted finite automata (WFAs), and describe a new algorithm for WFAs language inclusion that we use for XFA language inclusion, and describe how to use other ideas from the literature to improve the performance of WFA/XFA language-inclusion testing.

1 Introduction

Complex systems today are made up of many communicating programs or program components. In such systems, it is vitally important to ensure that the messages that one component sends to another are understood by the receiving component, otherwise runtime errors will occur. Incompatibilities in the message format can drive up the cost of developing a system because different components of a system are often developed by different development teams or different subcontractors, thus compatibility problems may not be detected until integration time. The cost of fixing errors found late in the development process is usually much higher than that of errors found earlier. For instance, Garlan, Allen, and Ockerbloom found that integrating four commercial, off-the-shelf (COTS) components into a larger system took several times longer than anticipated (5 person-years instead of 1 person-year) [37].

In addition to large systems built out of components, programs that are developed completely independently also often need to cooperate using a common file format. For instance, photo editors, viewer applications, web browsers, etc. all need to be able to understand the image formats that they share.

In this dissertation, we explore language-theoretic techniques for detecting incompatibilities between two such programs or components. The goal is twofold: (1) to infer models of the message format that each component operates on, and (2) to check compatibility of those formats. If the format models were completely accurate, then compatibility of the models would correspond exactly to compatibility of the programs in question. (However, because the models will be approximations to the actual formats, the final compatibility check does not produce a guaranteed result.)

We investigate three different types of automata, which serve as the models of the communication format: standard finite automata, nested-word automata, and extended finite automata. Along the way, we also developed many new theoretical results for nested-word automata and extended finite automata, which are presented in Chapters 2 and 3, along with a discussion of our implementations.

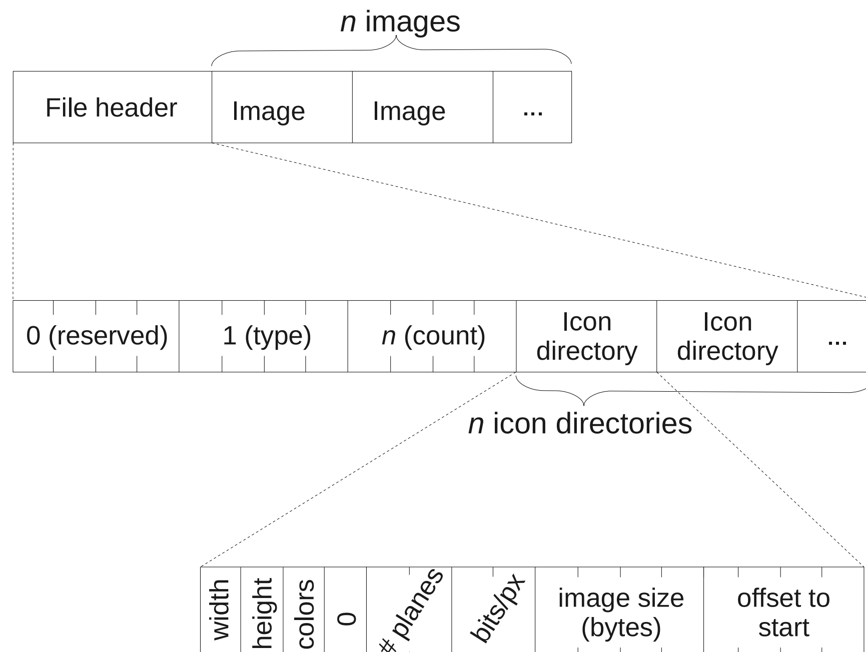


Figure 1.1: ICO file format [19, 41]. The 0 and 1 in the file header and the 0 in the icon directory are constants ("type" is 2 for cursors, which share almost the same format); "bits/px" in the directory is the number of bits per pixel, and each of the narrow boxes marks one byte. The format of the actual images is omitted for clarity.

In the remainder of the introduction, we discuss the structure of file formats (Section 1.1), briefly describe the inference and compatibility techniques used in the dissertation (Section 1.2), and provide a chapter-by-chapter overview (Section 1.3).

1.1 File Formats

As an illustration of the sort of information that we wish to infer, consider the format of the ICO (icon) file format [19, 41]. This format is pretty rich syntactically: it is capable of holding several different actual images (e.g., multiple icons at different resolutions or color depths so that software displaying the image can choose the most appropriate); and the format of each image can differ.

Figure 1.1 illustrates the ICO format. The format begins with a header containing the number of images and information about them, which is followed by the actual image data.

At a high level, there are three levels of information that one needs to be aware of to implement a program that handles a particular format:

1. Surface syntax. In the context of file formats, the surface syntax consists of the type and size of each field in the format. For instance, in the ICO format the height and width of each image are stored in the corresponding icon directory as single unsigned bytes.[1] Certain file formats may have some or all of the data in just a big "blob" — for instance, most of the data in a compressed format is just raw bytes that do not have any apparent syntactic structure.

2. Semantic constraints on the syntax. These are analogous to static-semantic restrictions on program source, which are checked by the compiler during name analysis and type checking. We give example constraints in the ICO format in Section 1.1.2.

3. Semantics. In the ICO file format, this would be something like "a word `0x00000000` in the image data itself will result in a black pixel."

The distinction between these three levels need not be hard and fast. For instance, consider a format that stores data in a compressed format and is decompressed before use. As mentioned above, the compressed data will likely have little to no apparent syntactic structure, but the uncompressed data may have another layer of syntax, semantic constraints, and semantics.

In this dissertation, we propose techniques to infer the raw syntax as well as certain semantic constraints. Semantics are not addressed.

1.1.1 Surface syntax of the ICO format

The surface syntax of a file format is a description of what types can appear where in a file. We will look at this in the context of the ICO format. The ICO file format was designed by Microsoft, and the specification is usually expressed using typedefs defined in the Windows header files. However, we will use standard C types instead, except that `byte` will be used in place of `unsigned char` for brevity. Equivalences between the standard types and Windows types are shown in Table 1.2, along with the size of each type that will be assumed throughout the dissertation.

[1] A height or width of 0 means 256.

# bytes	"C" type	Windows type
1	`byte` (for `unsigned char`)	`BYTE`
2	`short`	`WORD`
4	`int`	`DWORD`

Table 1.2: The sizes in bytes of C types that will be assumed throughout this dissertation, along with the names used by the Windows headers and documentation.

We can now describe the surface syntax of the ICO format. We will start with just the icon directory, which consists of four `byte`s, two `short`s, and two `int`s. We can describe that with the following simple regular expression:

```
byte byte byte byte short short int int
```

Moving up to the file header, we see that it consists of three `int`s then some number of icon directories. At this point we cannot really put a number on them, as we are only describing the surface-level syntax; restricting the number of icon directories to be equal to the count field will be imposed later. We can thus describe the surface syntax of the ICO format as a context-free grammar:

$\langle$*ico-file*$\rangle$ $\rightarrow$ $\langle$*ico-header*$\rangle$ $\langle$*ico-image*$\rangle^*$
$\langle$*ico-header*$\rangle$ $\rightarrow$ `int int int` $\langle$*ico-directory*$\rangle^*$
$\langle$*ico-directory*$\rangle$ $\rightarrow$ `byte byte byte byte short short int int`
$\langle$*ico-image*$\rangle$ $\rightarrow \cdots$

Because there is no recursion, this grammar actually defines a regular language, and a regular expression for it can be created by treating each of the nonterminals as a macro and simply collapsing all of the definitions.

1.1.2 Semantic constraints on the input

Of course, in some sense the reason that we can say that this is a regular language is because of the division, somewhat artificial, that I am choosing between the surface syntax and additional constraints. Not every string that matches the language of $\langle$*ico-file*$\rangle$ is a valid ICO file; there are additional requirements in place.

For instance, for the ICO format, here are some of the additional restrictions:

1. The fourth `byte` of each ⟨*ico-directory*⟩ must equal the constant 0.
2. The number n stored in the third `int` of the ⟨*ico-header*⟩ (in little-endian format) must match the number of icon directories present. (More precisely: in a parse tree using the above grammar, the ⟨*ico-header*⟩ node must have exactly n ⟨*ico-directory*⟩ children.)
3. The same number n must also match the number of ⟨*image*⟩ occurrences.
4. The "offset to start" field in each ⟨*ico-directory*⟩ must match the byte offset in the file of the corresponding ⟨*image*⟩.
5. The width, height, colors, # planes, and bits/pixel in each ⟨*ico-directory*⟩ must match duplicated information inside each ⟨*image*⟩ — not shown in Fig. 1.1. (Actually, for technical reasons, the height inside the ⟨*image*⟩ must be double that of the height in the corresponding ⟨*ico-directory*⟩.)

This list is not exhaustive, but it gives a flavor of the sort of constraints that fall into this category. Our goal will be to infer and check some constraints that look like these. We will not necessarily be able to infer all of them, but we do describe how to deal with some.

I separate semantic constraints from the surface syntax for the same reason that, for instance, parsing standard programming languages is typically described separately (and often carried out separately) from name analysis and type checking. Language syntax is almost always specified by a CFG, but non-context-free constraints are imposed, such as "if the code defines a procedure that expects k arguments, then calls to it must pass k arguments"; such constraints are checked by the compiler separately.

One difference between these two scenarios is as follows. For many languages it is possible to produce a reasonable parse or abstract-syntax tree without understanding any of the constraints, then simply reject programs that do not meet the list of constraints. In our context, the surface syntax may be ambiguous without understanding the constraints; for instance, if the format of ⟨*image*⟩ matches that of ⟨*ico-directory*⟩, then a parser won't know when to stop reading ⟨*ico-directory*⟩s and start reading ⟨*image*⟩s. However, because the automata that we work with can be nondeterministic, such ambiguities do not pose a problem for our method of compatibility checking.

1.1.3 Input semantics

The previous two sections describe what determines whether a file is valid, but of course have nothing to say about what a valid file *means*. For instance, consider what a program that displays an image should do to calculate what to show to the user. For historical reasons, the actual images inside of an image file consist of two pixels arrays called the "and mask" and the "xor mask." The color that should be displayed on screen at a pixel is a function of the color "underneath" the icon, the *and mask* at that pixel, and the *xor mask* at that pixel.[2]

The actual semantics of a file format outside the scope of this dissertation.

1.2 Format Inference and Compatibility Checking

As mentioned before, checking compatibility between programs consists of two goals, which we address in two steps. First, we infer the formats each program operates upon. Second, we check compatibility.

1.2.1 Format inference

The previous section described *what* we would like to infer about programs; now we give a brief overview of *how*. Our techniques split the inference step into two parts: inference of the surface-level syntax is separate from the semantic constraints. Inference is performed separately on the producer program (inferring the output format) and the consumer program (inferring the input format), but the methods of doing inference are largely the same.

To infer the surface-level syntax, we compute the interprocedural control-flow graph (CFG) of the program in question,[3] then transliterate it to an automaton to compute the

[2]The actual formula is $\mathit{out} = (\mathit{in}\ \mathrm{AND}\ \mathit{andmask})\ \mathrm{XOR}\ \mathit{xormask}$. For monochrome icons on a monochrome display, this scheme allows the icon to set a pixel white or black, but also to be transparent or to invert the color under the icon. (The last is more useful for mouse cursors, which largely share the same format.) For color displays, "inverting" does not make much sense, but the format still provides a means of achieving transparency.

[3]The control-flow graph of a program is a directed graph where program statements are vertices and there is an edge from vertex p to q if it is possible that execution can proceed directly from p to q. In an interprocedural CFG, edges carry an additional marker that specifies whether they are traversed on a procedure call, on a procedure return, or in normal intraprocedural flow.

control-flow abstraction of the program. In the resulting automata, procedure calls that perform either output operations (for the producer) or input operations (for the consumer) become transitions that read an input symbol.

We use three different kinds of automata in this dissertation: standard finite automata (FAs), nested-word automata (NWAs), and extended finite automata (XFAs). The latter two are extensions of standard FAs and are described in detail later. NWAs fall at an intermediate point between standard FAs and pushdown automata: they keep decidability and closure under language operations, but also retain the ability to express some traditionally context-free properties. XFAs combine an FA with an additional data state; in our context, the data state will be used to check the semantic constraints.

Chapter 5 explores what can be done using just the control-flow abstraction without any semantic constraints. The FA and NWA models are used in that chapter. Chapter 6 adds semantic constraints, using XFAs as models.

To infer the semantic constraints, it is possible to use any analysis technique that produces appropriate invariants over program values; we use a dynamic analysis based on Daikon [32, 67]. The program in question is first instrumented to contain extra code that tracks certain values that may be of interest, for example loop trip counts. (A loop trip counter is a variable that tracks the number of iterations of a loop. It is separate from the variables used by the actual programmer to track the loops, because such program variables may count down instead of up, may be increased by two each time, or may have any number of variations on the usual pattern, and or in fact not exist at all.) The power of the invariant-finding technique together with the instrumentation that was added determines what semantic constraints can be inferred.

My goal is to infer semantic constraints like Item 2 in the list given in Section 1.1.2: e.g., the “count” field in the ICO header must match the number of icon directories. This constraint is found when the invariant detector determines that the value read at the “count” field (an instrumentation variable) matches the trip count on the loop that reads the directories. (Unbounded repetition has to correspond to either some form of looping construct or recursion, which are both detectable for purposes of adding a trip count. Section 6.4 discusses specifics and limitations of trip counts.)

1.2.2 Modeling programs vs modeling formats

The reader may have noticed that there is a bit of a shift in what I am saying we will be modeling: I begin by stating that my techniques model file *formats*, but in the overview of the technique I describe things in terms of how to model *programs*.

In reality, these two notions are not as different as it may at first appear. Consider the case of a standard FA. An automaton defines a language; because the language is uniquely determined, the automaton itself can be used as an exact model for its language. (Typically automata are described as "reading" strings and thus correspond to the consumer in our setting, but there's no reason why it cannot be viewed as *generating* strings as well, which would put it into the producer role. Thus automata can serve both sides of the producer-consumer relationship.)

In fact, by the same argument the original program can serve as an exact model of its format. However, we need to be able to test language containment. Because programs are large and Turing complete, it is not possible to check containment if we are using the program itself as a model. (Language containment is undecidable even for context-free languages.) We therefore need to *abstract* the program to produce a *model* of the program, and in our context it is here that the FAs/NWAs/XFAs arise. Because the inferred automata model the programs, which in turn model the format, we can say that the inferred automata model the input.

For this reason, this dissertation will treat "a model of the format" and "a model of the program" as being interchangeable.

1.2.3 Format compatibility

Once we have inferred the formats of each component, we much check them for compatibility. Our models define languages of messages — the model of the producer defines its output language and the model of the consumer defines its input language. Checking whether the programs are compatible just means that we want to determine the truth of "every message in the (output) language of the producer is in the (input) language of the consumer."

Phrasing compatibility as language containment may be either *exactly* what we do, or just a conceptual description. For standard FAs, language containment is what we want

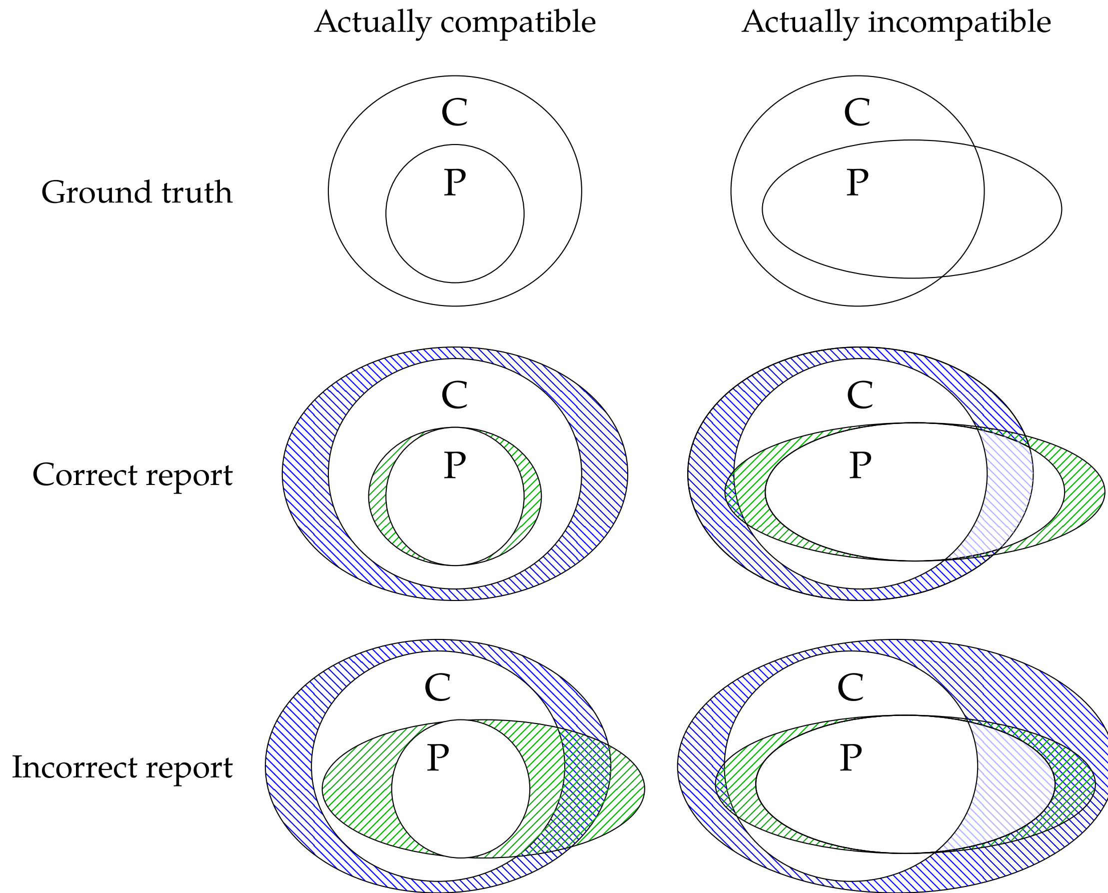


Figure 1.3: True and false positives and negatives. In each of the pictures, the unshaded circles represent the true language of the components and the hashed regions represent additional messages accepted by the models due to the overapproximation.

and can be answered using standard techniques. For NWAs, because of the format of the inferred automata, we must modify the consumer's automaton before checking language containment (Section 5.3). For XFAs, language containment is what we want, and can be answered using the techniques discussed in Section 3.6.

Unfortunately, because the models of both the producer and consumer are overapproximations of the actual behavior (i.e., the models accept more messages than the programs are capable of outputting or reading), there are four possibilities for the final result, including both false positives and false negatives, as illustrated in Fig. 1.3.

1.3 Dissertation Overview

The dissertation is divided into two parts. The first part discusses tools, both in terms of software as well as the theoretical background on the various less-common automata types that we use. The second part covers applications.

In Part 1:

Chapter 2 discusses nested-word automata (NWAs). NWAs are an extension of finite automata that retain the decidability and closure characteristics of FAs but can express richer languages. (It is also possible to view NWAs as a restriction of standard pushdown automata.) We provide the definition that we use, based on the work of Alur and Madhusudan (primarily [7]), discuss several theoretical results that I have obtained, and then present OpenNWA, which is a library that implements NWAs.
Contributions: Several theoretical results about NWAs and the OpenNWA library.

Chapter 3 discusses extended finite automata (XFAs). XFAs are an extension of finite automata that carry some state information around in scratch memory, instead of storing everything in the control portion of an automaton. Assuming the scratch memory is bounded in size, XFAs are of equivalent power to FAs but can be smaller. We provide the definition that we use, based on the work of Smith et al. ([76, 77]), discuss their relations to weighted finite automata (WFAs), and discuss several new algorithms for XFAs and/or WFAs.
Contributions: Relating XFAs to WFAs, a fast method for performing ε closure, a method for determining language containment of WFAs, and information about the implementation of XFAs within WALi, the *w*eighted *a*utomaton *li*brary.

Chapter 4 discusses a front end for the Daikon dynamic invariant detector that operates on x86 binary code with debugging information.
Contributions: A discussion of how we apply the ideas of program instrumentation and dynamic invariant detection to obtain the invariants we need for compatibility checking.

In Part 2:

Chapter 5 discusses the producer/consumer conformance analyzer (PCCA), which uses finite automata and nested-word automata to use the control abstraction to check compatibility.
Contributions: we discuss how to use automata to model the format of programs, how to use such models for compatibility checking, how to use NWAs to gain context sensitivity.

Chapter 6 discusses incorporating semantic constraints into program models.
Contributions: we discuss how to use XFAs to incorporate more information about data formats, how to infer such models, and how to perform compatibility with XFA models.

Chapter 7 presents work related to format inference and compatibility checking.

Chapter 8 concludes and presents some ideas for future work.

Part I

Tools

2 Nested-Word Automata (NWAs)

This chapter will explore nested-word automata (NWAs). NWAs form a middle ground between standard FAs and pushdown automata (PDAs): NWAs retain the ability to recognize some properties that are traditionally considered context free, for example matching parentheses; at the same time, nested-word languages are closed under operations such as language inclusion and complementation, which is not true for context-free languages, and thus more operations are decidable.

This chapter starts with theoretical concerns: definitions and some algorithms. (NWAs themselves are not my invention, but I did produce a couple of novel algorithms and observations.) We then discuss a library for NWAs called OpenNWA, discussing the operations and features it supports and uses.

Because this chapter alternates between discussing my work and background material, sections are marked with either "background", which indicates that it only presents background material, "adaption", which indicates that the section presents an adaption of an existing technique, or "new", which indicates that the section is mostly or entirely novel.

2.1 Definitions and Theory

This section will discuss the theoretical underpinnings of nested-word automata. We provide informal and formal definitions (Sections 2.1.1 and 2.1.2), then describe three algorithms that operate on NWAs: determinization (Section 2.1.4), Kleene star (Section 2.1.5), and reversal (Section 2.1.6). The first expresses the same algorithm as existing work in a different way, the second corrects an existing algorithm, and the third is a new algorithm which better preserves NWA properties of interest.

Confusingly, there are three kinds of automata directly related to NWAs,[1] two of which have actually been called "nested-word automata." In the remainder of this introduction,

[1] Actually there are even more once you start looking at modular VPAs, multi-entry modular VPAs, etc., but we do not consider these variations, nor other related constructs, such as tree automata.

we briefly relate the three definitions in order to set the stage for those who have prior knowledge. However, we do *not* discuss specifics of how the definitions differ. Readers who are familiar with the area and wish for a refresher of the literature and how our definition relates may continue reading, while others may wish to skip ahead to Section 2.1.1.

Alur and Madhusudan introduced all three automata types. The first paper to describe a form of automaton similar to NWAs did so using the term *visibly pushdown automata* (VPAs) [6]. Followup work in DLT [7] provided the first definition of NWAs using that name. While at this point nested-word automata were closely related to visibly pushdown automata, the correspondence was not yet immediate. Finally, Alur and Madhusudan produced a comprehensive treatment of the subject for the JACM [8] that expanded the definition of NWAs and introduced two named variants: *linearly-accepting* and *weakly-hierarchical* NWAs. Applied appropriately, these two restrictions result in the older definitions.

To reiterate, there are three kinds of NWAs which are of interest at this point:

- Fully general NWAs, called "JACM NWAs" in this dissertation
- Linearly-accepting NWAs, which can be seen as an alternative encoding of a VPA
- Weakly-hierarchical, linearly-accepting NWAs, which match the DLT definition

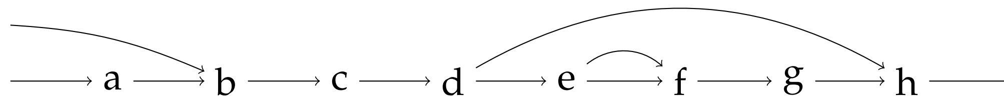
All three types are capable of accepting the same set of languages; the differences are only at the level of the automaton. However, the minimum automaton size that recognizes a given language can vary by definition. It is possible to convert an arbitrary JACM NWA into a linearly-accepting NWA that accepts the same language by doubling the number of states. It is possible to convert an arbitrary JACM NWA into a weakly-hierarchical one by increasing the number of states by a factor of $2|\Sigma|$. These constructions are described in Alur and Madhusudan [8, theorems 3.1 and 3.2]

This dissertation defines "NWA" to match the DLT definition, augmented with ε transitions (Section 2.1.3). This also is the version of NWAs implemented in OpenNWA. While this may seem like a sub-optimal choice because the automata may need to be larger, most of the construction algorithms outlined by Alur and Madhusudan [8] impose one or both restrictions. As a result, supporting JACM NWAs would require conversions during these operations. (For instance, determinization requires a linearly-accepting NWA [8, theorem 3.3], while concatenation requires a weakly-hierarchical NWA [8, theorem 2.6].)

2.1.1 Informal Description of Nested-Word Automata (NWAs) [background]

This section describes nested-word automata [7] and related terms at an intuitive level, and gives an example of how they are used in program analysis. For the formal definitions, see Section 2.1.2.

A nested word is an ordinary (linear) string of symbols over some alphabet Σ paired with a *nesting relation*. The nesting relation describes a hierarchical relation between input positions, for instance between matched parentheses. Graphically, a nested word can be depicted in a manner such as the following:

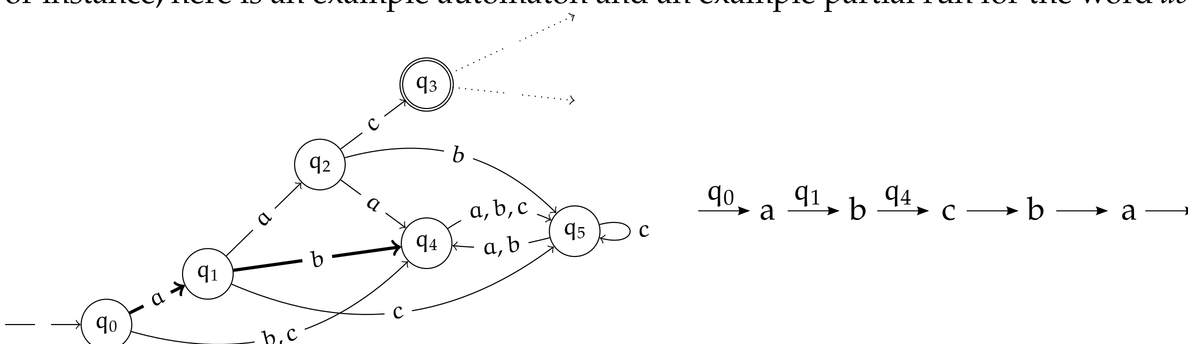


In this image, following just the horizontal arrows illustrates the linear word *abcdefgh*, while the curved edges ("*nesting edges*") indicate the nesting relation $\{(-\infty, 2), (4, 8), (5, 6)\}$. (See the following section for a formal definition of the nesting relation.) For a nesting relation to be valid, nesting edges must only point forward in the word and may not share a position or cross.

Positions in the word that appear at the left end of a nesting edge are called *call positions*, those that appear at the right end are called *return positions*, and the remaining are *internal positions*. It is possible to have *pending* calls and returns, which are not matched within the word itself. For a given return, the source of the incoming nesting edge is called the *call predecessor*. In the previous example, *d* and *e* are in call positions; *b*, *f*, and *h* are in return positions; and *a*, *c*, and *g* are in internal positions. All calls are matched, but *b* is the return position of a pending return.

A *nested-word automaton* (NWA) is a generalization of ordinary finite automata that recognizes languages of nested words. An NWA's transitions are split into three sets—internal transitions, call transitions, and return transitions. Internal transitions work the same as transitions in a standard finite automaton. Call transitions work similarly, with a difference that is explained below. However, return transitions also look at and must match the state the automaton was in before reading the call predecessor.

To understand how an NWA works, consider first the case of an ordinary finite automaton reading a linear word. Picture the word in somewhat of an unusual way: as a graph with one node per symbol in the word and one edge pointing from each symbol to the next. Also include one edge with no source pointing at the first symbol and one edge with no target pointing from the last symbol. We can think of the machine's operation as labeling each edge with the state the machine is in after reading the symbol at that edge's source. For instance, here is an example automaton and an example partial run for the word *abcba*:



To find the next state, the automaton looks for a transition out of q_4 with the symbol c — in this case, with a target of q_5 — and labels the next edge in the word with q_5.

The operation of an NWA proceeds in a fashion similar to a standard finite automaton, except that the machine also labels the nesting edges. When the machine reads an internal position, it chooses a transition and labels the next linear edge the same way a finite automaton would. When the machine reads a call position, it picks a matching call transition and labels the next linear edge in the same way, but also labels the outgoing *nesting* edge with the state that the machine is leaving. When the machine reads a return position, it looks not only at the preceding linear state but also at the state on the incoming nesting edge. It then chooses a return transition that matches both, and labels the next linear edge with the target state. An example NWA and a run accepted by it are shown in Fig. 2.1.

Our NWAs allow ε internal transitions, which operate in an analogous way to ε rules in ordinary finite automata. Section 2.1.3 discusses ε transitions further.

Operationally it is sometimes beneficial to think of an NWA by analogy to a restricted pushdown automaton, which behaves as follows:

- When reading a call, the PDA pushes (just) the current state onto the stack (which corresponds to labeling a nesting edge)

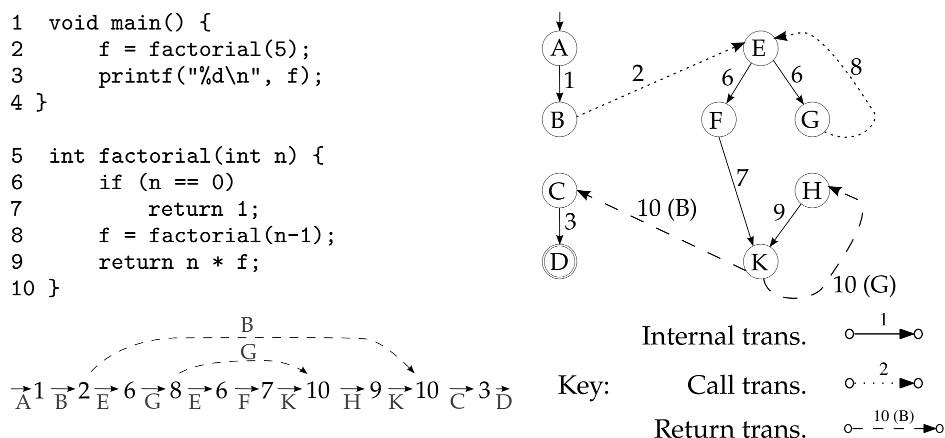


Figure 2.1: An example program, corresponding NWA, and accepted word. (Note that the linear portion of the word is $12686\cdots$, not $ABEGE\cdots$.) State labels are arbitrary; transition symbols give the line number of the corresponding statement. Some nodes are elided.

- When reading an internal position, the PDA may not modify the stack
- When reading a return, the PDA must pop exactly one item from the stack, the value of which can help determine the successor state. (This step corresponds to reading the value on the innermost nesting edge.)

This definition is roughly that of a related model called a visibly pushdown automaton (VPA) [6, 8], and VPAs and NWAs and their languages can be seen as alternative encodings of each other.

Modeling a Program as an NWA

NWAs can be used to encode the interprocedural control-flow graph (ICFG) of a program. Intraprocedural ICFG edges become internal transitions, interprocedural call edges become call transitions, and interprocedural return edges become return transitions. For an ICFG return edge (*exit-site*, *return-site*), we use the call site that corresponds to *return-site* in the call-predecessor position of the NWA's transition. The symbols on a transition depend on the application, but frequently are the corresponding statement.

The restriction imposed by matching the call predecessor's state at a return gives us the desirable property that nested words accepted by the NWA correspond exactly to the valid paths through the program (data values ignored); it is not possible for the NWA to call a

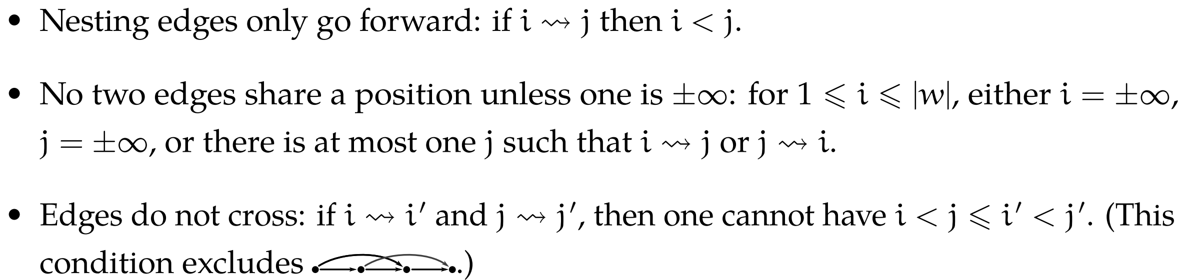
procedure from one call site and return to a different point. (Said more formally, the NWA will reject words that correspond only to such a path.)

An example program, the corresponding NWA, and an example word accepted by that NWA are shown in Fig. 2.1. The fact that we are using an NWA allows us to exclude paths such as *1-2-6-8-6-7-10-3*, which is invalid.

2.1.2 Formal Definition of Nested-Word Automata (NWAs) [background]

Now we can define the terms required to formalize NWAs. For now, we disallow ε transitions.

Definition 2.1. A *nested word* $(w, \rightsquigarrow)$ over alphabet Σ is an ordinary (linear) word $w \in \Sigma^*$ together with a *nesting relation* $\rightsquigarrow$. The relation $\rightsquigarrow$ is a collection of edges (over the positions in w) that do not cross. Formally, $\rightsquigarrow \subseteq \{-\infty, 1, 2, \ldots, |w|\} \times \{1, 2, \ldots, |w|, +\infty\}$ such that:

- Nesting edges only go forward: if $i \rightsquigarrow j$ then $i < j$.
- No two edges share a position unless one is $\pm\infty$: for $1 \leqslant i \leqslant |w|$, either $i = \pm\infty$, $j = \pm\infty$, or there is at most one j such that $i \rightsquigarrow j$ or $j \rightsquigarrow i$.
- Edges do not cross: if $i \rightsquigarrow i'$ and $j \rightsquigarrow j'$, then one cannot have $i < j \leqslant i' < j'$. (This condition excludes .)

A *nested-word language* is any set of nested words; such a language is a *regular nested-word language* if it is accepted by an NWA as defined below.

When $i \rightsquigarrow j$ holds, for $1 \leqslant i \leqslant |w|$, i is called a *call* position. If $i \rightsquigarrow +\infty$, then i is a *pending call*; otherwise i is a *matched call*, and the (unique) position j such that $i \rightsquigarrow j$ is called its *return successor*.

Similarly, when $i \rightsquigarrow j$ holds, for $1 \leqslant j \leqslant |w|$, j is a *return* position. If $-\infty \rightsquigarrow j$, then j is a *pending return*, otherwise j is a *matched return*, and the (unique) position i such that $i \rightsquigarrow j$

is called its *call predecessor*. We will often abuse the term "call predecessor" to refer to the *state* that the automaton was in before it read the call predecessor.

A position $1 \leqslant i \leqslant |w|$ that is neither a call nor a return is an *internal* position.

Note that these terms refer to positions within w and not to the symbol itself, which is what you may expect if you are familiar with visibly pushdown languages [6]. In other words, in an NWA it is possible for a symbol σ to be used in all three roles (internal, call, and return) in different positions.

A nested word is *balanced* if it has no pending calls or returns. A nested word is *unbalanced-left* (or a *nested-word prefix*) if it has only pending calls, and it is *unbalanced-right* (or a *nested-word suffix*) if it has only pending returns.

Definition 2.2. A *nested-word automaton* (NWA) A is a 5-tuple $(Q, \Sigma, Q_0, \delta, F)$, where Q is a finite set of states, Σ is a finite alphabet, $Q_0 \subseteq Q$ is a set of initial states, and $F \subseteq Q$ is a set of final states. δ holds the transition functions, and comprises three components, $(\delta_c, \delta_i, \delta_r)$, where:

- $\delta_i : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function for internal positions of the input word.
- $\delta_c : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function for call positions.
- $\delta_r : Q \times Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function for return positions.

Starting from a state in Q_0, an NWA A reads a nested word $(w, \rightsquigarrow)$ from left to right, and performs transitions according to the current input symbol and $\rightsquigarrow$. Suppose A is in state q when reading input symbol σ at position i. If i is an internal (resp, call) position in $\rightsquigarrow$, then A makes a transition to a state q' (if one is available) that is an element of $\delta_i(q, \sigma)$ (resp, $\delta_c(q, \sigma)$). If i is a return position, let k be the call predecessor of i (so $k \rightsquigarrow i$) and q_c be the state A was in just before the transition it made on the kth symbol; A changes to a state q' in $\delta_r(q, q_c, \sigma)$. If there is a computation of A on input $(w, \rightsquigarrow)$ that terminates in a state $q \in F$, then A accepts $(w, \rightsquigarrow)$.

The above definition does not describe how q_c, the state A was in just before reading the call predecessor of a return, is determined. The informal definition described it in terms of labeling the nesting edge, but the natural interpretation is closer to that of a

visibly-pushdown automaton, whereby the NWA would keep a stack of the states on call edges that have yet to be matched.

We will sometimes treat the transition functions as ternary or quaternary relations instead of functions; in other words, we will treat $(q, \sigma, q') \in \delta_i$ and $q' \in \delta_i(q, \sigma)$ as equivalent.

2.1.3 NWAs and ε transitions [new]

None of the formulations of NWAs or VPAs discussed previously (including JACM NWAs) allow ε transitions. This section discusses our extension to the NWA definition which *does* allow them, discusses why epsilon transitions must be limited to internal transitions, and describes an alternative semantics for how ε transitions interact with return transitions.

Extending NWAs to support ε transitions

We allow ε transitions on internal edges. The definition of an NWA is extended as would be expected; δ_i is redefined as follows:

- $\delta_i : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$

and the semantics of the NWA are extended to allow it to change its current state from q to any state in $\delta_i(q, \varepsilon)$ at any point.

Existing algorithms need only minor modifications to support these new semantics, and in fact supporting ε transitions makes some constructions such as Kleene-star simpler to express.

We can also define an ε-removal process. Given an NWA A, this process will produce a new NWA A' that does not have ε transitions and accepts the same language. A' will have the same number of states as A, but perhaps more transitions. (In particular, this process does *not* do any determinization.) Let $\textsc{eclose}(q)$ be the ε closure of state q defined in the usual way.

A' will start with exactly the same states and *non-*ε transitions as A. We then add additional transitions as defined in the rules in Fig. 2.2. The rules are depicted diagrammatically in Fig. 2.3.

$$\textsc{Internals}\quad \frac{q \in Q \qquad q_\varepsilon \in \varepsilon\textsc{close}(q) \qquad \sigma \in \Sigma \qquad q' \in \delta_i(q_\varepsilon, \sigma)}{(q, \sigma, q') \in \delta_i'}$$

$$\textsc{Calls}\quad \frac{q \in Q \qquad q_\varepsilon \in \varepsilon\textsc{close}(q) \qquad \sigma \in \Sigma \qquad q' \in \delta_c(q_\varepsilon, \sigma)}{(q, \sigma, q') \in \delta_c'}$$

$$\textsc{Returns}\quad \frac{q, q_c \in Q \qquad q_\varepsilon \in \varepsilon\textsc{close}(q) \qquad \sigma \in \Sigma \qquad q' \in \delta_r(q_\varepsilon, q_c, \sigma)}{(q, q_c, \sigma, q') \in \delta_r'}$$

Figure 2.2: Rules defining transitions *added* to an NWA during ε removal. (Non-ε transitions that exist in the original automaton are copied over unchanged, and are not depicted in this figure.) These rules are diagrammed in Fig. 2.3.

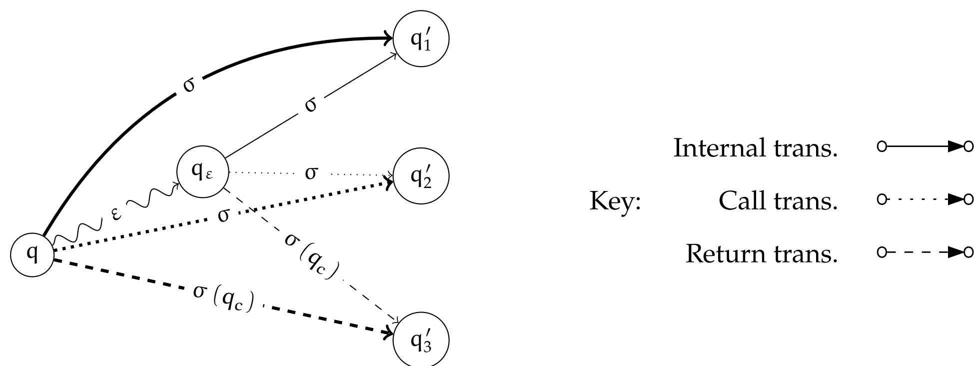


Figure 2.3: Illustrating NWA ε removal. The thick lines show new transitions that are added, and the wavy line shows a sequence of ε transitions. The new internal transition is present because of the old internal transition, and similarly for the call and return transitions. For the return transitions, q_c can be any state.

Why must we only allow ε transitions on internal edges?

Unfortunately, it is impossible to be as general as one might wish to be. In particular, it is not possible to allow arbitrary call and return ε transitions.

A critical feature of NWAs, and the source of the name *visibly pushdown* automata, is that the call and return positions in each input word are "visible" — where the NWA will push and pop is apparent just by looking at the input. Allowing the automaton to spontaneously take an ε transition on a call means that it is no longer possible to tell just by looking at the input word where the calls occur.

What follows is a proof sketch that it is in general impossible to include ε transitions on calls and returns. We will define a translation from an arbitrary context free grammar

G to an ${}^{?}$NWA A_G, such that if the nesting relation is dropped from each word in $L(A_G)$, the result is $L(G)$. That is, $L(G) = \{w \mid (w, \rightsquigarrow) \in L(A_G)\}$. (We borrow some linguistics notation and use "${}^{?}$NWA" to indicate that the result is not actually a legal NWA.) This translation allows us to establish the following contradiction: to compute the intersection of two context-free languages expressed as G_1 and G_2, translate their grammars to a pair of ${}^{?}$NWAs A_{G1} and A_{G2} and compute their intersection [8, theorem 3.5]. Because no symbols in Σ appear on call or return transitions in L_{G1} or L_{G2} and they will not be introduced during intersection, it is possible to interpret $A_{G1} \cap A_{G2}$ as a VPA whose language is $L(G_1) \cap L(G_2)$.

The translation creates an ${}^{?}$NWA that mimics the way a nondeterministic recursive descent parser would recognize an input.[2] A_G tracks a "current" production and location within that production, as well as a stack of where it was before. When then next symbol in the current production is a terminal, A_G reads the next input symbol and continues if they match. When the next symbol in the current production is a nonterminal, A_G stacks the current production and location, nondeterministically chooses the next production matching the nonterminal in question that it should next use, and starts at the beginning of that new production. When A_G reaches the end of the current production, it pops the current production off the stack and continues where the previous one left off. Figure 2.4 gives an example of this translation for an unambiguous grammar for the language $\{ww^{\mathcal{R}}\}$, which cannot be recognized by a deterministic PDA or by a VPA. We now define this translation formally.

Suppose that we are given a CFG $G = (N, \Sigma, P, S)$. (N is the set of nonterminals, Σ the terminals, P the productions, and S the start nonterminal.) For each production $p \in P$, let $|p|$ be the number of symbols that appear on the right-hand side of p. Each production p has $|p| + 1$ *positions* $\{0, 1, 2, \cdots, |p|\}$, corresponding to the point before and after each symbol. Let p_i be the ith symbol of p's right-hand side, one-indexed (i.e., p_1 comes after position 0 and before position 1).

The ${}^{?}$NWA A_G will have a state for each position of each production, as well as distinguished start and accepting states. Formally, $Q = \{q_{p,n} \mid p \in P \text{ and } 0 \leqslant n \leqslant |p|\} \cup \{q_0, q_f\}$. The transitions are defined by the rules shown in Fig. 2.5.

[2] A slightly-modified version of this construction could be used as a CFG-to-PDA transformation as part of a proof of their equivalence, but it differs somewhat from the transformations presented by, for example, Hopcroft et al. [40, §6.3.1] and Sipser [75, Lemma 2.21].

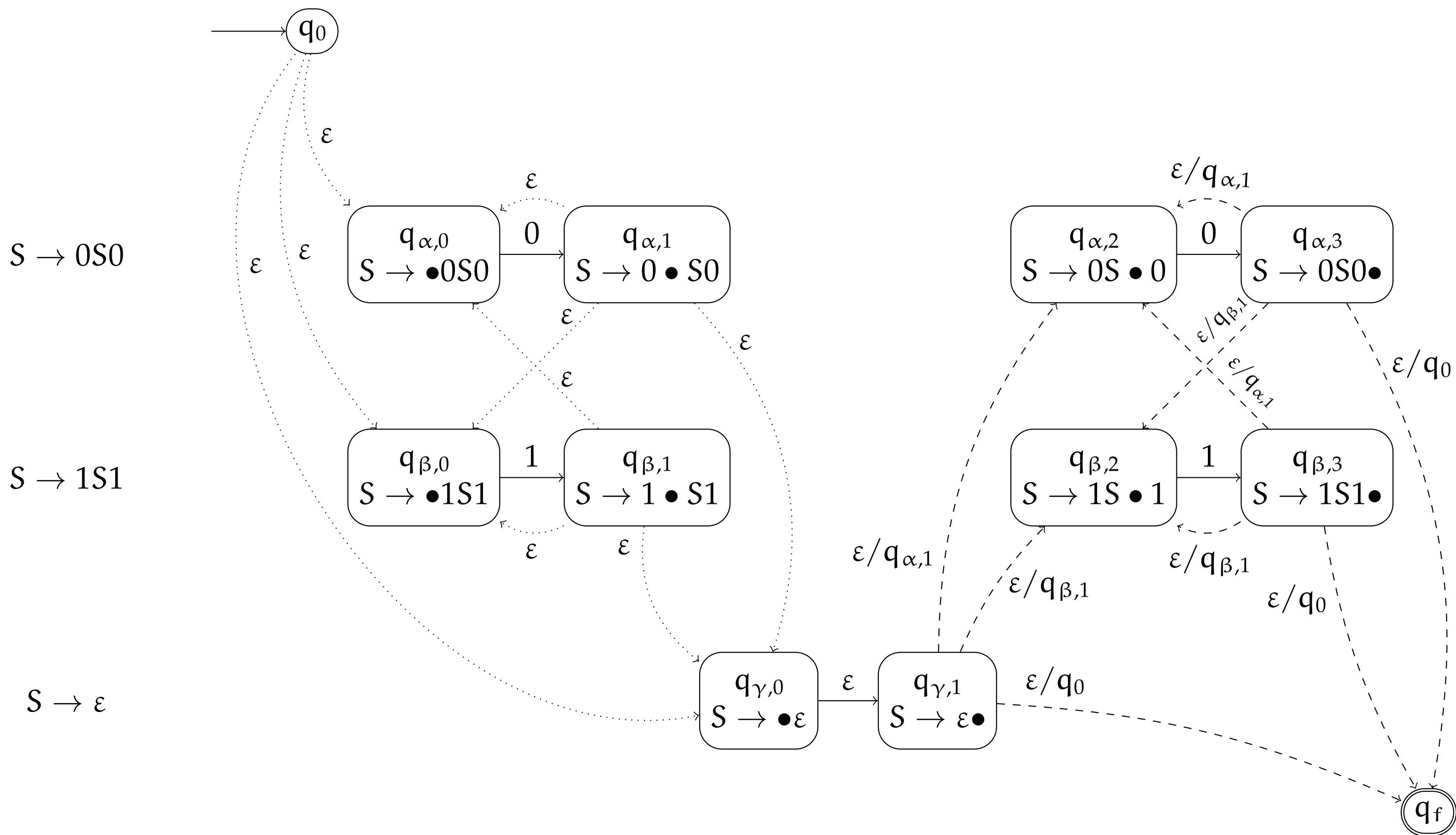


Figure 2.4: An invalid NWA (using ε call and return transitions) generated from the CFG $S \to 0S0 \mid 1S1 \mid \varepsilon$ using the translation discussed in the text and Fig. 2.5. Each state is labeled with both the $q_{p,i}$ form discussed in the text as well as the LR-item it represents.

$$\textsc{Terminals}\quad \frac{p \in P \qquad 1 \leqslant i \leqslant |p| \qquad p_i \in \Sigma}{(q_{p,i-1}, \sigma, q_{p,i}) \in \delta_i}$$

$$\textsc{Nonterminals}\quad \frac{p, p' \in P \qquad 1 \leqslant i \leqslant |p| \qquad p_i \in N \qquad p' \equiv (p_i \to \cdots)}{(q_{p,i-1}, \varepsilon, q_{p',0}) \in \delta_c \qquad (q_{p',|p'|}, q_{p,i-1}, \varepsilon, q_{p,i}) \in \delta_r}$$

$$\textsc{Begin}\quad \frac{p \in P \qquad p' \equiv (S \to \cdots)}{(q_0, \varepsilon, q_{p,0}) \in \delta_c}$$

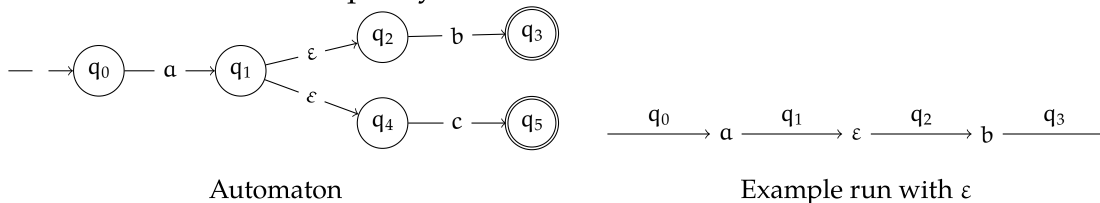
$$\textsc{End}\quad \frac{p \in P \qquad p' \equiv (S \to \cdots)}{(q_{p,|p|}, q_0, \varepsilon, q_f) \in \delta_c}$$

Figure 2.5: Rules defining the transitions created during translation from a CFG to an invalid NWA.

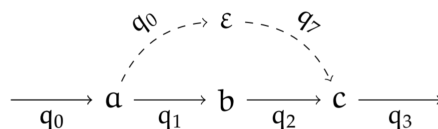
Allowing ε moves on nesting edges

Interestingly, even if we restrict ourselves to allowing ε transitions in δ_i only, there is still at least one choice to be made: do we allow ε moves on the call predecessor? The answer to this question does not affect the class of languages that can be recognized or even the number of states in an automaton, but it can affect the number of transitions.

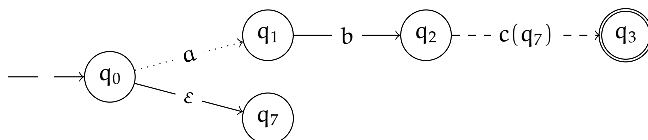
To explain this question, consider what happens with an internal ε transition (or in a standard FA). Using a diagram like that of Fig. 2.1, taking an ε transition (an "ε move") can be seen as splitting a linear edge in the input word and putting ε between the two halves, just as if it were another input symbol:



There are two kinds of edges in a nested word, but the extension allowing ε transitions earlier in this section only allows splitting the linear edges. Can we do the same thing for nesting edges? In other words, can we have something like the following?



It turns out we can. The above is a valid (and accepted) run of the following automaton under a different semantics for ε transitions than we have presented so far:



Note that even though this alternative semantics changes what return transitions can be taken (the (q_2, q_7, c, q_3) transition is not usable under the first semantics but will be under this alternative semantics), there are still only internal ε transitions.

We can define the alternative sematics as follows. The structure of the automaton does not change. However, consider the following statement from the definition of an NWA:

> If i is a return position, let k be the call predecessor of i (so $k \rightsquigarrow i$) and q_c be the state A was in just before the transition it made on the kth symbol; A changes to a state q' in $\delta_r(q, q_c, \sigma)$.

We now wish to allow q_c to be not just the state the automaton was in at the call predecessor, but *any* state reachable from there via ε transitions. Thus we can change the statement above to:

> If i is a return position, let k be the call predecessor of i (so $k \rightsquigarrow i$) and q_c be the state A was in just before the transition it made on the kth symbol. A chooses any state q_ε such that there is an ε path from q_c to q_ε, and then changes to a state q' in $\delta_r(q, q_\varepsilon, \sigma)$.

Once we have the alternative semantics, it seems quite likely that each of the different NWA constructions (or easy variants of them) will maintain their correctness under the alternative semantics. For instance, consider the determinize algorithm discussed in the next section and presented on Page 31. To make the determinize algorithm respect the alternative semantics, we simply have to add an additional, optional composition with *Close*, the relation representing the ε closure. This option is discussed in the following section.

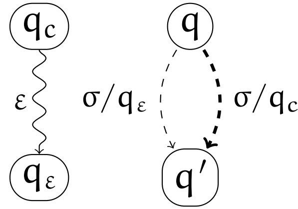


Figure 2.6: Illustrating translating between ε semantics. The thick line shows the new transition that is added as a result of the presence of the other two when translating an NWA that uses the modified ε semantics to the original semantics.

It is possible to translate NWAs between the two semantics. Given an NWA A, let $L(A)$ be the language under the original semantics, and $L_{alt}(A)$ be the language under the alternative semantics.[3]

We will start by constructing an A' such that $L(A) = L_{alt}(A')$. Unfortunately, there is no real way to ensure that A' does not take extra ε transitions on the call predecessor (these are what we must prevent). Thus what we will do is just perform ε removal on A to obtain A' as described in Figs. 2.2 and 2.3. Because $L(A') = L_{alt}(A')$ if A has no ε transitions and $L(A) = L(A')$ by ε removal, we get that $L(A) = L_{alt}(A')$.

In the other direction, we wish to construct an A' such that $L(A') = L_{alt}(A)$. In this case, A allows extra return transitions; we will add these return transitions explicitly. For each return transition (q, q_c, σ, q'), add a new return transition $(q, q_\varepsilon, \sigma, q')$ for each $q_\varepsilon \in \varepsilon\text{CLOSE}(q_c)$. This transformation is illustrated in Fig. 2.6.

2.1.4 NWA determinization [adaption/background]

We found the explanations of how to determinize NWAs that are given by Alur and Madhusudan [7, 8] to be confusing (and contradictory between the two accounts), and so we reformulated the algorithm using relational operations. In this dissertation, we will sometimes refer to a state in a determinized automaton as a "cell" and reserve the term "state" for the input automaton.[4]

[3]Interpreting a single automaton under two different semantics may already be familiar from interpreting a pushdown automaton as accepting via final states versus accepting via an empty stack [40, §6.2.2]. In that case as well, it is possible to translate PDAs between the two semantics.

[4]This terminology is from De Wulf's et al.'s paper on antichains [25].

Each cell R in the determinized automaton is a binary relation on states in the original. In a standard determinized FA, a cell $\{q_0, q_1, \cdots, q_n\}$ means the automaton can be in state q_0 of the input FA, or in state q_1, etc. For NWAs, a cell $\{(p_0, q_0), (p_1, q_1), \cdots, (p_n, q_n)\}$ means that the NWA is one of the states $\{q_0, q_1, \cdots, q_n\}$, but the relation carries around extra meaning.

If a cell in the determinized automaton contains a pair (p, q), then this means the input automaton can begin in the state p, immediately perform a call transition, follow a path with balanced calls and returns, and finally arrive in state q. In other words, it represents a transition sequence in the original NWA that results in exactly one new pending call. In such a configuration, if the input automaton then reads a return symbol, q is the source of the return transition and p is the call predecessor. These two pieces of information are exactly what the automaton needs to know to determine which transitions it can take. The call predecessor p needs to be stored explicitly because it is possible to arrive at the same state q with different call predecessors.

At the start of the run and any time the automaton has not read any unmatched calls, the first component of each pair in the current state will be some $q \in Q_0$; in this situation, the initial states act as call predecessors. This situation is an exception to the "immediately take a call transition" portion of the above description.

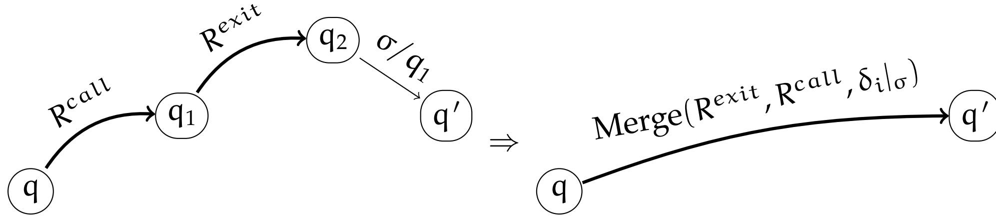
We use the following notation in the determinize algorithm, shown on Page 31:

$(Q, \Sigma, \delta, Q_0, Q_f)$	The components of the input automaton `nwa`
$\delta_i\|_\sigma$	The binary relation $\{(p, q) \mid \exists\sigma : (p, \sigma, q) \in \delta_i\}$
$\delta_c\|_\sigma$	The binary relation $\{(p, q) \mid \exists\sigma : (p, \sigma, q) \in \delta_c\}$
$\delta_r\|_\sigma$	The ternary relation $\{(p, c, q) \mid \exists\sigma : (p, c, \sigma, q) \in \delta_r\}$
$R\,;S$	Relational composition of the binary relations R and S
R^*	Reflexive-transitive closure of the binary relation R
R^T	Relational transpose; $R^T = \{(a, b) \mid (b, a) \in R\}$
Q^{new}, δ^{new}	Components of the determinized NWA

We use the following auxiliary function to compute the target of a return transition:

$$\begin{aligned} \text{Merge}(R^{exit}, R^{call}, \delta|_\sigma) = \{(q, q') \quad \mid \quad & \exists q_1, q_2.\, (q, q_1) \in R^{call} \\ & \text{and } (q_1, q_2) \in R^{exit} \\ & \text{and } (q_2, q_1, q') \in \delta|_\sigma\} \end{aligned}$$

The operation of $Merge$ can be diagrammed as follows:

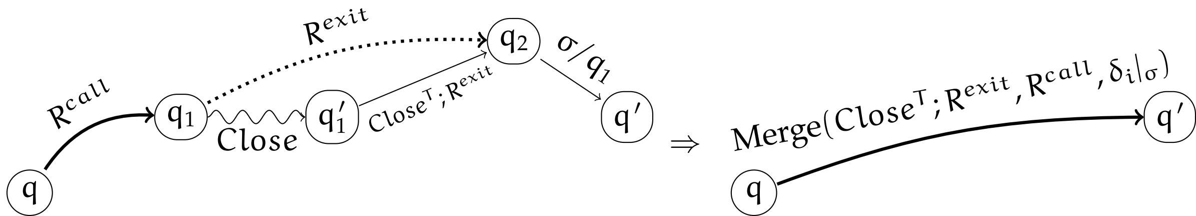


Here the bold lines that curve up represent a call followed by a matched path (or starting at a start state and taking a matched path), and hence correspond to a cell. The line that goes down from q_2 to q' represents a return transition. At a return, the call predecessor must match the state at the source of the upwards-curved edge, as that state corresponds to the call that is being matched.

If we would like the alternative ε semantics discussed in the last part of Section 2.1.3, we could modify $Merge$. Requiring that q_1 matches between R^{exit} and $\delta|_\sigma$ enforces the requirement that a return transition must match the actual call predecessor, so it is this requirement that we should relax. Instead, we could require that $(q_1, q_2) \in R^{exit}$ and that there is a q_1' such that $(q_2, q_1', q') \in \delta|_\sigma$ and $q_1' \in \varepsilon\textsc{close}(q_1)$:

$$\begin{aligned}\text{Merge}_{alt}(R^{exit}, R^{call}, \delta|_\sigma) = \{(q, q') \quad | \quad & \exists q_1, q_1', q_2.\, (q, q_1) \in R^{call} \\ & \text{and } (q_1, q_1') \in \mathit{Close} \\ & \text{and } (q_1, q_2) \in R^{exit} \\ & \text{and } (q_2, q_1', q') \in \delta|_\sigma\}\end{aligned}$$

The last requirement can be also expressed as $(q_1, q_1') \in \mathit{Close}$, where $\mathit{Close} = (\delta_i|_\varepsilon)^*$, and that leads to a better way of handling ε transitions in this context: leave $Merge$ unmodified, and simply compose R^{exit} with the transpose of Close before passing it in:



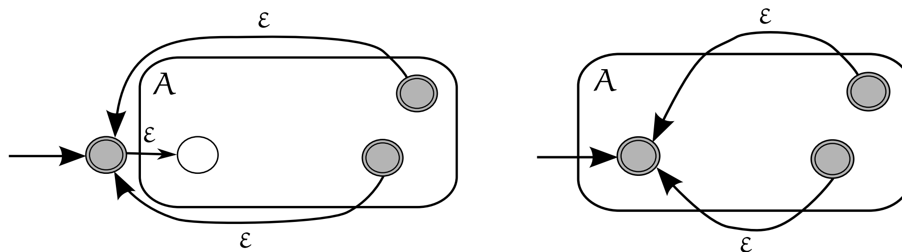
The determinize algorithm presented on Page 31 takes this approach.

1 **determinize(NWA nwa)**
2 $\text{Close} = (\delta_i|_\varepsilon)^*$
3 $R_0 = Q_0 \times Q_0 \,;\text{Close}$
4 $Q^{new} = \{R_0\}$
5 Insert R_0 in *WL*
6 **while** $WL \neq \varnothing$ **do**
7 select and remove a relation R from *WL*
`// Note that R is a state in` Q^{new}
8 mark R
9 **for** $\sigma \in \Sigma$ **do**
`// Compute internal transitions`
10 $R^i = R\,;\delta_i|_\sigma\,;\text{Close}$
11 $Q^{new} = Q^{new} \cup \{R^i\}$
12 Insert $R \xrightarrow{\sigma} R^i$ into δ_i^{new}
13 **if** R^i *unmarked* **then**
14 $WL = WL \cup \{R^i\}$
`// Compute call transitions`
15 $R^c = \text{Close}\,;\delta_c|_\sigma\,;\text{Close}$
16 $Q^{new} = Q^{new} \cup \{R^c\}$
17 Insert $R \xrightarrow{\sigma} R^c$ into δ_c^{new}
18 **if** R^c *unmarked* **then**
19 $WL = WL \cup \{R^c\}$
`// Compute return transitions where R appears as the exit node`
20 **for** $R^{call} \in Q^{new}$ **do**
21 $R^r = \texttt{Merge}(R, R^{call}\,[;\,\mathit{Close}^T]^\dagger, \delta_r|_\sigma)\,;\text{Close}$
22 $Q^{new} = Q^{new} \cup \{R^r\}$
23 Insert $(R, R^{call}, \sigma, R^r)$ into δ_r^{new}
24 **if** R^r *unmarked* **then**
25 $WL = WL \cup \{R^r\}$
`// Compute return transitions with R as the call predecessor`
26 **for** $R^{exit} \in Q^{new}$ **do**
27 $R^r = \texttt{Merge}(R^{exit}, R\,[;\,\mathit{Close}^T]^\dagger, \delta_r|_\sigma)\,;\text{Close}$
28 $Q^{new} = Q^{new} \cup \{R^r\}$
29 Insert $(R^{exit}, R, \sigma, R^r)$ into δ_r^{new}
30 **if** R^r *unmarked* **then**
31 $WL = WL \cup \{R^r\}$
`// end worklist while loop`
32 $Q_f^{new} = \{R \in Q^{new} |\ \text{there is } (p, q) \in R \text{ with } q \in Q_f\}$
33 **return** $(Q^{new}, \Sigma, \delta^{new}, \{R_0\}, Q_f^{new})$
34 † Include this extra ;*Close* to obtain the alternative ε-transition semantics (see the end of Section 2.1.3).

2.1.5 Kleene star [adaption]

The construction of Kleene star presented in [8, Theorem 3.6] has a minor error; we present a corrected version, which also takes advantage of our ability to have ε transitions.

The error is analogous to not adding a distinguished initial state in the traditional Thompson construction for FAs, but instead just making the initial state accept and connecting the accepting states of the automaton to the initial state with ε transitions.[5] The following diagrams illustrate the problem (note that in A, the initial state does not accept):



Correct FA Kleene star construction for A^* Incorrect FA Kleene star construction for A^*

The bug in Alur and Madhusudan's presentation can, in fact, be exhibited using the same example (it is not necessary to use NWA calls or returns). Alur confirmed our diagnosis [4]. Below, we present a version that uses ε-transitions, and thus it looks a bit different from Alur and Madhusudan's version. However, the high-level idea of the construction is the same as the original version.

Kleene star is complicated in the case of NWAs because of the ability to have unbalanced words in the automaton's language. When concatenating a nested word w_1 containing pending calls with a nested word w_2 containing pending returns, some or all of the pending calls and returns will "connect" in $w_1 w_2$ in the natural way. The construction for Kleene star (and also concatenation [8, theorem 3.6], which we did not modify and do not describe) needs to take special care to make sure that the correct return transitions can be taken when reading a newly-matched return. In particular, when an NWA A is reading just w_2 and sees a pending return, A takes a return transition where the call predecessor is an initial state of A. However, if we naïvely use the same construction as in ordinary FAs, then when reading the corresponding position in $w_1 w_2$, the automaton will instead match transitions based on the state it was in when reading the newly-matched call in w_1. Alur and Madhusudan's

[5]The initial state of A^* must accept because ε is in L^*; and because of this property it is incorrect to merely add epsilon transitions from the old final states to the old initial states. If there is a cycle from the initial state back to itself, the word corresponding to that path would be accepted even though it should not be.

solution to this problem is to have the machine "pretend" that it is reading a pending return instead of a matched return in the appropriate places when reading w_2.

When computing $A^* = A^*$ for some NWA A, the resulting NWA has two "copies" of A. These are denoted by primed and unprimed version of states from A in the definition below. (See Fig. 2.7.)

Suppose that A^* is reading a word $w = w_1 w_2 \cdots w_n$, where each $w_i \in L(A)$. A^* begins in a start state of A'. Henceforth it maintains the following invariant on the state that A^* is in with respect to the portion of w read so far: if the next symbol σ is in a return position, then that symbol is a *pending* return in the current w_i iff A^* is in the A' portion; i.e., if the current state is primed. (Note that this return only needs to be pending in the current w_i. In the full string w, σ may match a call in an earlier w_j or it may be pending in the whole string.)

Internal transitions thus keep A^* in the same copy of A, and call transitions always take A^* to the unprimed copy of A (because if it then reads a return, the return will match that call). Return transitions can target either copy of A: if the call predecessor is unprimed, then the target will be unprimed; if the call predecessor is primed, then the target will be primed. This behavior is enabled by the Internal, Call, and Locally-Matched-Return rules in Fig. 2.8.

The description above describes A^*'s operation under "normal" conditions. If A^* is in a final state (either primed or unprimed) of the automaton A, it is also allowed to guess that it should "restart" by taking an ε transition to a distinguished start and final state q_0. This guess is correct if it just read the last character in w_i (making the next character the first one in w_{i+1}). Note that q_0 only has transitions to the A' portion of A^*, maintaining the invariant. This behavior is enabled by the Start and Restart rules in Fig. 2.8.

The reason for the two copies of A comes into play when A^* reads a return σ while in the A' portion. By the invariant, σ is pending in the current w_i. In the original automaton A, the transitions that the machine can use are return transitions (q, r, σ, p) where the call predecessor r is in Q_0. We need to make sure that A^* can take those same transitions. There are two cases we need to consider:

1. For the cases where σ is pending in the whole string w, we need to have a version of the return transition with q_0 in the call-predecessor position, so we add (q', q_0, σ, p'). This behavior is enabled by the Locally-Pending-Return rule.

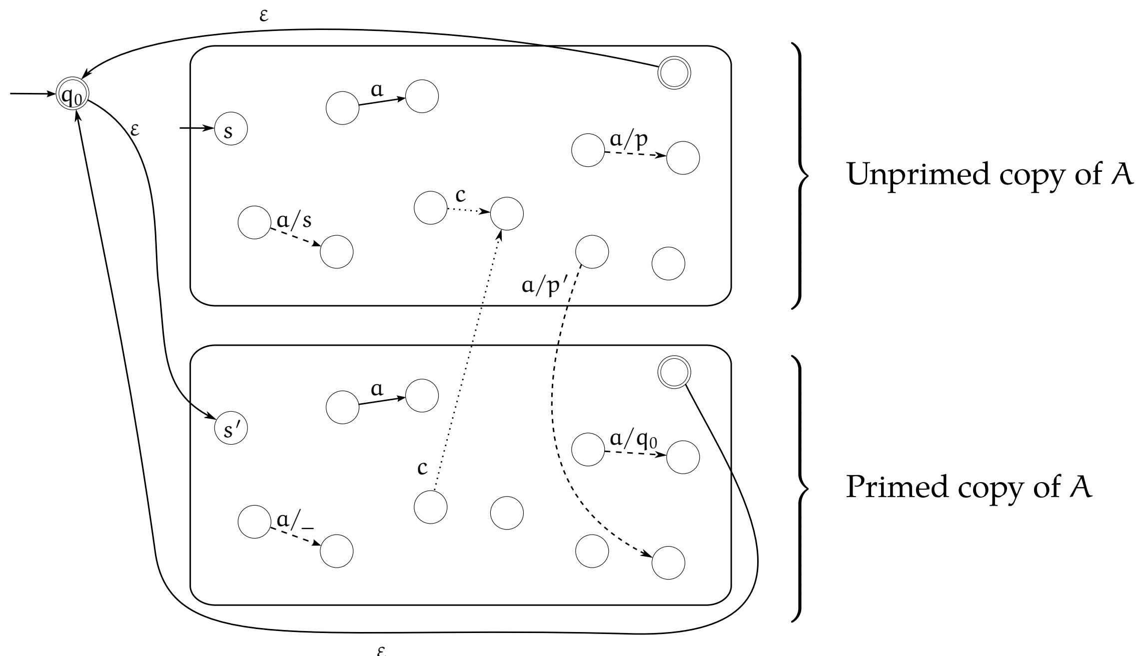


Figure 2.7: Diagram of NWA Kleene star construction.

2. For the cases where σ is matched with a call in some earlier w_j, it does not matter what state the machine was in before that call; thus we add (q', s, σ, p') for each state s in $Q \cup Q'$. This behavior is enabled by the GLOBALLY-PENDING-RETURN rule.

Formally, if the original NWA is $(Q, \Sigma, Q_0, \delta, Q_f)$, then the result of performing Kleene star on that NWA is $(Q^*, \Sigma, Q_0^*, \delta^*, Q_f^*)$. The sets of states are defined by $Q^* = Q \cup Q' \cup \{q_0\}$ (with $Q' = \{q' \mid q \in Q\}$ and $q_0 \notin Q$), and $Q_0^* = Q_f^* = \{q_0\}$. The transitions in δ^* are defined by the rules in Fig. 2.8.

2.1.6 Weakly-hierarchical-preserving NWA Reversal [new]

The NWA reversal algorithm presented by Alur and Madhusudan's JACM article [8, theorem 3.7] does not work with their earlier definition of NWAs, which matches our use. Their earlier definition and our definition is called a *weakly-hierarchical* NWA, and the reversal algorithm in JACM does not preserve the weakly-hierarchical property of NWAs. We thus

$$\textsc{Start}\quad \frac{q \in Q_0}{(q_0, \varepsilon, q') \in \delta_i^*}$$

$$\textsc{Restart}\quad \frac{q \in Q_f}{(q, \varepsilon, q_0) \in \delta_i^* \qquad (q', \varepsilon, q_0) \in \delta_i^*}$$

$$\textsc{Internal}\quad \frac{(q, \sigma, p) \in \delta_i}{(q, \sigma, p) \in \delta_i^* \qquad (q', \sigma, p') \in \delta_i^*}$$

$$\textsc{Call}\quad \frac{(q, \sigma, p) \in \delta_c}{(q, \sigma, p) \in \delta_c^* \qquad (q', \sigma, p) \in \delta_c^*}$$

$$\textsc{Locally-Matched-Return}\quad \frac{(q, r, \sigma, p) \in \delta_r}{(q, r, \sigma, p) \in \delta_r^* \qquad (q, r', \sigma, p') \in \delta_r^*}$$

$$\textsc{Locally-Pending-Return}\quad \frac{(q, r, \sigma, p) \in \delta_r \qquad r \in Q_0 \qquad s \in Q \cup Q'}{(q', s, \sigma, p') \in \delta_r^*}$$

$$\textsc{Globally-Pending-Return}\quad \frac{(q, r, \sigma, p) \in \delta_r \qquad r \in Q_0}{(q', q_0, \sigma, p') \in \delta_r^*}$$

Figure 2.8: Rules defining transitions created by Kleene star

present a new algorithm for NWA reversal that produces a weakly-hierarchical NWA.[6] To our knowledge, this algorithm has not been found or published by others, although the DLT paper asserted without proof that regular nested-word languages are closed under reversal [7, Theorem 2], even under its more restrictive NWA definition that we use.

A nested word $n = (w, \rightsquigarrow)$ is reversed by reversing the linear word w and exchanging calls and returns. Formally, $n^{\mathcal{R}} = (w^{\mathcal{R}}, \{(|w| + 1 - r, |w| + 1 - c) \mid (c, r) \in \rightsquigarrow\})$. (Pending calls and returns are handled by defining $|w| + 1 - (+\infty) = -\infty$ and $|w| + 1 - (-\infty) = +\infty$.) Roughly speaking, call transitions in A correspond to return transitions in $A^{\mathcal{R}}$ and vice versa, and we reverse the direction of all transitions as in the standard FA construction. We describe the construction from the perspective of $A^{\mathcal{R}}$ — that is, a "call transition" is a call transition in $A^{\mathcal{R}}$, and a "call" is a call in the reversed string.

Perhaps unsurprisingly, pending returns pose a problem because the role of initial and final states are exchanged. Because of this complication, the algorithm for reversing an

[6]The algorithm presented produces smaller automata than would be achieved by applying Alur and Madhusudan's reversal construction (which does not change automaton size) followed by a conversion to a weakly-hierarchical NWA (which increases the size by a factor of $2\,|\Sigma|$); the construction we present doubles the size, which is a savings of a factor of $|\Sigma|$ over the reverse-then-convert approach just outlined.

$$\textsc{Internal}\quad \frac{(p, \sigma, q) \in \delta_i}{(q, \sigma, p) \in \delta_i^{\mathcal{R}} \qquad (q', \sigma, p') \in \delta_i^{\mathcal{R}}}$$

$$\textsc{Reversed-Call}\quad \frac{(q_x, _, \sigma_r, q_r) \in \delta_r}{(q_r, \sigma_r, q_x), (q_r', \sigma_r, q_x) \in \delta_c^{\mathcal{R}}}$$

$$\textsc{Call-Return}\quad \frac{(q_c, \sigma_c, q_e) \in \delta_c \qquad (q_x, q_c, \sigma_r, q_r) \in \delta_r}{(q_e, q_r, \sigma_c, q_c), (q_e, q_r', \sigma_c, q_c') \in \delta_r^{\mathcal{R}}}$$

$$\textsc{Pending-Return}\quad \frac{(q_c, \sigma, q_e) \in \delta_c \qquad q_f \in Q_f}{(q_e', q_f, \sigma, q_c') \in \delta_r^{\mathcal{R}}}$$

Figure 2.9: Rules defining transitions created during NWA reversal

NWA has a similar flavor to that of the Kleene-star procedure. The automaton $A^{\mathcal{R}}$ has two "copies" of A (primed and unprimed), and maintains the same invariant as the Kleene-star construction: if the next symbol σ is in a return position, then that symbol is a *pending* return iff $A^{\mathcal{R}}$ is in the A' portion.

If the original NWA is $(Q, \Sigma, Q_0, \delta, Q_f)$, then the result of reversing that NWA is $(Q \cup Q', \Sigma, Q_f', \delta^{\mathcal{R}}, Q_0)$, where $\delta^{\mathcal{R}}$ is obtained using the rules in Fig. 2.9.

Proof. In this section, we will prove that $(L(A))^{\mathcal{R}} \subseteq L(A^{\mathcal{R}})$. The reverse direction (to get the equality) proceeds in a similar fashion.

Let A be an NWA. It may be nondeterministic, but for simplicity of presentation assume it does not have ε transitions. Let $A^{\mathcal{R}}$ be the machine constructed following the above procedure. For a word w, let w_i be the ith character of w, let $w[: i]$ be $w_1 w_2 \cdots w_{i-1}$, let $w[i :]$ be $w_i w_{i+1} \cdots w_{|w|}$, and let $w[i : j]$ be $w_i w_{i+1} \cdots w_{j-1}$.

$\mathbf{L}(\mathbf{A})^{\mathcal{R}} \subseteq \mathbf{L}(\mathbf{A}^{\mathcal{R}})$**.** We will prove the following statement:

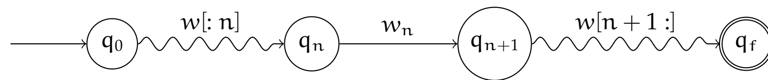
- If $w \in L(A)$, then for $n \in 0, 1, \ldots, |w|$, there are q_0, q_n, q_f such that:
  - $q_n \in \delta(q_0, w[: n])$, and
  - Both of the following hold:
    * $q_n \in \delta^{\mathcal{R}}(q_f, (w[n :])^{\mathcal{R}})$ if $(w[n :])^{\mathcal{R}}$ has no pending calls
    * $q_n' \in \delta^{\mathcal{R}}(q_f, (w[n :])^{\mathcal{R}})$ if $(w[n :])^{\mathcal{R}}$ has pending calls

We will prove this assertion by "finite backwards induction" (the base case will be $n = |w|$ and we will show that if that statement is true for $n = k$, then it will be true for $n = k - 1$).

The base case is trivial: let $q_n \in \delta(q_0, w) \cap Q_f$ and $q_f = q_n$. There has to be such a q_n if we assume that $w \in L(A)$.

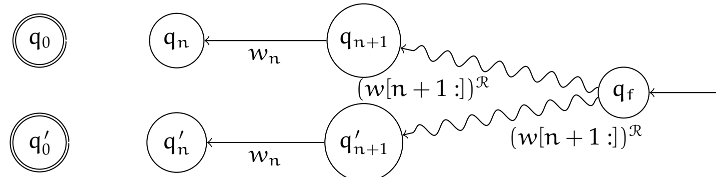
We split the inductive case into four parts depending on the role of the symbol w_n.

$\mathbf{w_n}$ is at an internal position. Because w is accepted by A, we know that A contains a path matching w that can be illustrated as follows:



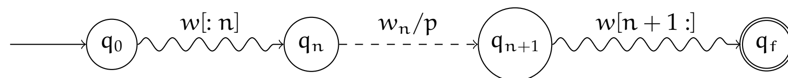
The meaning of this diagram is that A contains states q_0, q_n, q_{n+1}, and q_f (some of these states may be the same), that there is a path from q_0 from q_n on the substring $w[: n]$, there is an internal transition (q_n, w_n, q_{n+1}), and there is a path from q_{n+1} to q_f on $w[n + 1 :]$ that matches any calls read during the $w[: n]$ portion.

The inductive hypothesis tells us that, in $A^{\mathcal{R}}$, one of q_{n+1} or q'_{n+1} is reachable by reading $(w[n :])^{\mathcal{R}}$, depending on whether there are pending returns in that substring. The presence of (q_n, w_n, q_{n+1}) in A along with the INTERNAL construction rule guarantees that the two internal transitions (q_{n+1}, w_n, q_n) and (q'_{n+1}, w_n, q'_n) are present in $A^{\mathcal{R}}$. We can diagram this portion of $A^{\mathcal{R}}$ as follows:

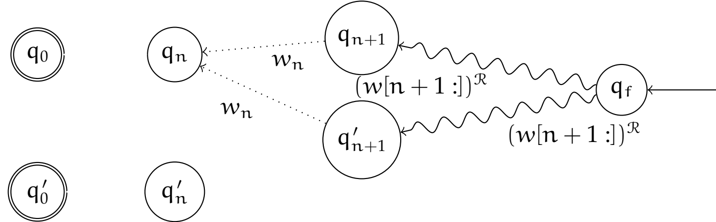


It is now clear that there is a path from q_f to either q_n or q'_n that can be traversed while reading $(w[: n + 1])^{\mathcal{R}}$. Because appending an internal position to $(w[: n + 1])^{\mathcal{R}}$ does not affect any pending calls, q'_n is reachable iff $(w[: n + 1])^{\mathcal{R}}$ has pending calls, otherwise q_n is reachable. This reestablishes our inductive hypothesis.

$\mathbf{w_n}$ is in a call position in $w^{\mathcal{R}}$. We use a similar argument to the previous. We can diagram a path through A as follows:

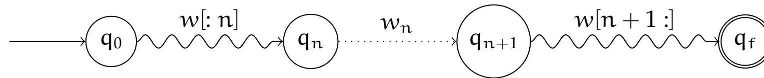


for some call predecessor p. By the presence of the transition (q_n, p, w_n, q_{n+1}) and the REVERSED-CALL rule, we know that $A^{\mathcal{R}}$ has the following transitions

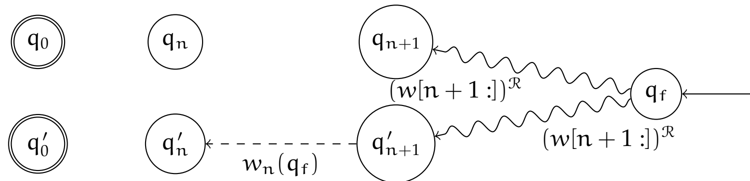


Again, the wavy lines indicate possible paths. By induction, either q_{n+1} or q'_{n+1} is reachable. It is always possible to take a call transition where the symbol matches, which means that q_n is guaranteed to be reachable. Finally, because w_n is in a call position, $(w[n:])^{\mathcal{R}}$ has pending calls (at least the one just read), which along with the fact that q_n is reachable reestablishes the inductive claim.

$\mathbf{w_n}$ is a pending return in $w^{\mathcal{R}}$**.** A contains the path



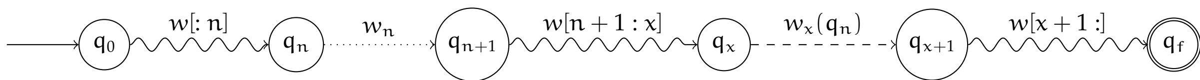
The relevant transition in $A^{\mathcal{R}}$ is



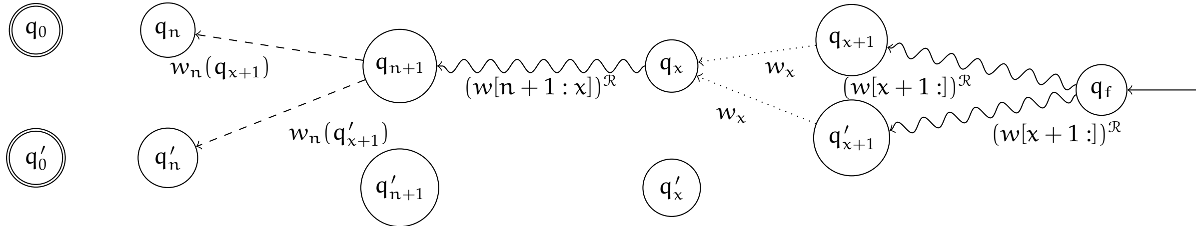
which we know is present by the PENDING-RETURN rule.

Because w_n is a pending return in $w^{\mathcal{R}}$, when $A^{\mathcal{R}}$ is reading $w^{\mathcal{R}}$ the automaton will use any initial state — in particular, it can use q_f — as a call predecessor. Thus $A^{\mathcal{R}}$ will be able to reach q'_n. Furthermore, because w_n was a pending return in the reversed string there cannot be any pending calls (or w_n would have matched one), which along with the fact that q'_n is reachable reestablishes the inductive claim.

$\mathbf{w_n}$ is a matched return in $w^{\mathcal{R}}$**.** A contains the path



for the x for which $n \rightsquigarrow x$, where $w[n+1 : x]$ (and hence the path between them) is matched. Then by the CALL-RETURN rule, we know that $A^{\mathcal{R}}$ contains the following states, transitions, and paths:



Thus as $A^{\mathcal{R}}$ reads $w^{\mathcal{R}}$, when it reads w_x it will place either q_{x+1} or q'_{x+1} on the nesting edge. At that point, $A^{\mathcal{R}}$ has a pending call, thus by induction it must be able to reach q_{n+1}. By the construction, it will then be able to reach either q_n or q'_n depending on which of q_{x+1} and q'_{x+1} it stacked earlier. Which of those happens depends on whether $q[x+1 :]$ has pending calls or not; $(w[n :])^{\mathcal{R}}$ has pending calls iff $(w[x+1 :])^{\mathcal{R}}$ has pending calls. Again, these situations correspond to the two options in the inductive hypothesis. □

2.2 The OpenNWA Library [new]

We have created an implementation of nested-word automata called OpenNWA. The library has an implementation of an NWA data structure and most of the automaton algorithms discussed in [8]. In addition, it is packaged with and interacts closely with the WALi library for weighted automata and weighted pushdown systems (WPDSs) [44], allowing for reachability queries on the configuration space of NWAs as well as for the computation of weights that summarize properties of sets of nested words (Section 2.2.3).

In this section, we list the operations that OpenNWA supports and then describe some of the highlights of the library features.

2.2.1 Supported Operations

As previously mentioned, OpenNWA supports most automata-theoretic operations:

- intersection
- union
- Kleene star
- reversal
- concatenation
- determination
- complement
- emptiness checking
- example word generation

For the most part, we use the algorithms developed by Alur and Madhusudan [7, 8]; however, the algorithms presented in Section 2.1 for Kleene star and reverse are substituted, and we perform emptiness checking and example word generation via conversion to a WPDS [30, 31].

In addition to these operations, OpenNWA also supports something we call *wild* transitions, which match any single symbol. Wilds can appear on any transition type. (They are particularly useful if the entire NWA's alphabet is not available up-front, e.g., if the list of all statements in a program is unknown [82, §3.1].) Wild transitions are not used in the format-compatibility work described in this dissertation, however, and will not be discussed further.

2.2.2 Client Information

OpenNWA provides a facility that we call *client information*. This feature allows the user of the library to attach arbitrary information to each node in the NWA. For instance, as discussed in Section 2.2.4, McVeto uses NWAs internally and uses client information to attach a formula to each node in the NWA.

The library tracks this information as best as it can through each of the operations it supports, and supports callback functions to compute new client information when it does not have the information it needs. For instance, during concatenation, the nodes in the resulting NWA have the same client information as the corresponding nodes in the input automata. However, for intersection, the nodes in the resulting NWA represent pairs of states, and the NWA class provides callback functions for computing the client information for each output state from the client information of states being paired.

2.2.3 Inter-operability with WPDSs

Weighted pushdown systems (WPDSs) can be used to perform interprocedural static analysis of programs [72]. The PDS proper provides a model of the program's control flow, while the weights on PDS rules express the dataflow transformers. Algorithms exist to query the configuration space of WPDSs, which corresponds to asking a question about the data values that can arise at a set of configurations in the program's state space. A configuration consists of a control location and a list of items on the stack.

The NWA library supports converting an NWA into a WPDS implemented by WALi. This feature allows a user of the NWA library to issue queries about the configuration space of an NWA. For instance, our `isLanguageEmpty()` function uses WPDS algorithms to effectively ask a query of the form "Is it possible to start in an initial configuration and consume a nested word to reach a configuration where the automaton is in an accepting state?"

NWAs themselves are not weighted, but the library provides a facility for determining the weights of the WPDS rules during the conversion. The user provides an instance of a subclass of `WeightGen`, which acts as a a factory function. The function is called with the states in question and returns the weight of the resulting WPDS rule. It is, of course, possible to use the client information of the states in question to determine the weight.

The library also allows conversions from a WPDS into an NWA [31, §7.1].

2.2.4 OpenNWA uses

OpenNWA is used by the Producer-Consumer Conformance Analyzer (PCCA) as discussed in this dissertation (Chapter 5); here we briefly describe two other uses of OpenNWA.

Machine-code model checking

McVeto is a machine-code verification engine that, given a binary and a description of a bad target state, tries to find either (i) an input that forces the bad state to be reached or (ii) a proof that the bad state cannot be reached [82].

McVeto uses a model of the program called a *proof graph*, which is an NWA that overapproximates the program's behavior. States in a proof graph are labeled with formulas; transitions are labeled with program statements or conditions. Each formula is associated with its state using OpenNWA's client information.

The initial proof graph is a very coarse overapproximation of the program, which McVeto progressively refines. One principle technique for refinement uses directed test generation to produce a concrete trace of the program's behavior, performs *trace generalization* [82, §3.1] to convert the trace into an overapproximating NWA (the "folded trace"), and intersects the current proof graph with the folded trace to obtain a refined proof graph. The formula on a state in the refined proof graph is computed by conjoining the formulas on the

states that are being paired from the current proof graph and the folded trace; OpenNWA callback functions compute the new formulas.

McVeto makes use of `prestar()` (Section 2.2.3) for two purposes: first, to determine whether the target state is reachable in the proof graph, and second, to determine which "frontier" to extend next during directed test generation [82, §3.3]. The `prestar()` query uses shortest-distance weights in order to find the shortest path to the target state. If the target state is reachable in the proof graph, then McVeto needs to continue exploration, in which case the frontier closest to the target state is heuristically a good candidate to explore next; otherwise, the target is guaranteed to be unreachable in the program.

JavaScript security-policy checking and weaving

The JAM tool checks a JavaScript program against a security policy, using counterexample-guided abstraction refinement (CEGAR, [20]) either to verify that the program is correct with respect to that policy already or to insert dynamic checks into the program to ensure that it will behave correctly.

JAM builds two models of the input program, one that overapproximates the control flow of the program and one that overapproximates the data flow. The policy is expressed as an NWA of forbidden traces. By intersecting the policy automaton with both program models, JAM obtains an automaton that expresses traces that possibly violate the policy. (For technical reasons, the intersection of the policy automaton and the data-flow automaton are not done using OpenNWA, although the final intersection is.)

Once JAM has the combined NWA, which represents possible policy violations, it asks OpenNWA for a shortest word in the language of the NWA. If the language is empty (i.e., there is no shortest word), the program always respects the policy. If OpenNWA returns an example word w, JAM checks whether w corresponds to a valid trace through the program. If w is valid, then JAM inserts a dynamic check to halt concrete executions corresponding to w that would violate the policy. If w is not valid, than JAM can either refine the abstraction and repeat, or insert a dynamic check as if w were actually feasible. (The dynamic check forces the abstraction to be refined sufficiently so that the same counterexample w will never be returned on a subsequent analysis round.)

JAM's refinement process benefits greatly from the ability to request a *shortest* accepted word, instead of any word, because it speeds up validity checking and refinement.

2.3 Related Work

Our focus has been on using NWAs for modeling programs, but as suggested by the introduction, there are a number of other application areas as well. Alur and Madhusudan each maintain a page giving a significant bibliography of papers that present theoretical and practical results related to NWAs and VPAs [5, 52].

The NWA language containment check that OpenNWA supports and that is used by PCCA performs determinization of one of the argument automata; the same is true of checking the universality of an NWA.[7] Friedmann, Klaedtke, and Lange describe how to adapt to NWAs existing "Ramsey-based" techniques for testing the universality and inclusion of standard FAs [36]. Ramsey-based techniques avoid the need to explicitly determinize the automaton, and result in a significant speedup. Friedmann et al.'s work postdated the bulk of our work on OpenNWA and the non-XFA version of PCCA, and so at the moment we do without this benefit.

There are a number of libraries that support standard finite automata, such as OpenFST [3], AT&T's FSM library [59], `dk.brics.automaton` [60], libalf's components [13], and many others. We took inspiration for portions of our APIs from these projects, particularly OpenFST. There is also experimental code included in WALi for converting between OpenFST acceptors and NWAs with no call or return transitions.

VPAlib [62] is a Java library that implements VPAs. In addition to being unsuited for our purposes because of the choice of language, WALi's implementation is also far more complete. For instance, VPAlib does not support concatenation, complementation (although it does support determinization), checking emptiness, or getting an example word.

Madhusudan's VPA page [52] lists several other VPA tools, but all appear to be targeted at a particular problem and none besides VPAlib seem to be publicly available.

[7] An automaton A is universal iff $L(A) = \Sigma^*$.

3 Extended Finite Automata (XFAs) & Weighted Finite Automata (WFAs)

In this chapter, we describe extended finite automata (XFAs) and weighted finite automata (WFAs). XFAs are an extension to standard FAs where the automata contain some extra "scratch memory" in addition to the state. (This is a bit like the RAM model of computation, except that the scratch memory is bounded.) When the XFA takes a transition, the transition it chooses also describes how it modifies its scratch memory.

Unlike NWAs, XFAs do not add any additional power over FAs. However, the scratch memory (and perhaps even more to the point, the fact that XFAs leave open the possibility for compactly representing the memory operations) can allow them to be more compact than an equivalent FA.

XFAs were invented by Smith, Estan, and Jha to more compactly represent acceptors being used in intrusion detection systems [76]. Smith et al. assume they are provided with many regular expressions that specify malware signatures, and want to scan incoming packets to determine if any signature matches. Smith et al. claim "DFAs are time-efficient but space-inefficient, NFAs are space-efficient but time-inefficient, ... [and] for a large class of signatures XFAs have time complexity similar to DFAs and space complexity similar to or better than NFAs."

XFAs caught our eye because we are interested in checking properties about the number of times loops iterate and other properties that the scratch memory of XFAs can store. However, Smith et al. were only interested in matching concrete strings, and not interested in other properties of XFAs, such as language containment or emptiness checking. That left us to find methods to perform these operations without materializing the entire state space explicitly.

WFAs are finite automata augmented with values called *weights*, which can track additional information.[1] A WFA reads a string and outputs a weight instead of a strict yes/no

[1]Weights in this sense are more general than just a numeric value as can be found in typical graph problems, for instance; see Section 3.2.

answer. (Mohri has called WFAs "string-to-weight transducers" [57].) WFAs have a much longer history than XFAs, and it is possible to view XFAs as a special case of WFAs. We treat them as such, and present algorithms that we developed for XFAs in a somewhat more general setting of WFAs.

Section 3.1 defines XFAs, Section 3.2 defines WFAs, and Section 3.3 describes the connection between the two. We then cover algorithms for ε closure (Section 3.4), determinization (Section 3.5), and language containment (Section 3.6). Finally, we describe how we represent the scratch memory transformers using binary decision diagrams (BDDs) (Section 3.7).

As with the NWA chapter, this chapter discusses both my work and background material, and sections are marked "background", "adaption", and "new" as appropriate.

3.1 Formal definition of extended finite automata (XFA) [background]

In the chapter introduction, we briefly described what an XFA is; here we present Smith et al.'s formal definition [76, Definition 2]:

Definition 3.1. A (nondeterministic) XFA is a tuple $(Q, D, \Sigma, \delta, U, QD_0, F)$ where:

- Q is the set of states, Σ is the alphabet, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ is the transition function. (These elements are standard.) As with NWAs, we will abuse notation and also treat δ as a relation, i.e., a subset of $Q \times (\Sigma \cup \{\varepsilon\}) \times Q$.
- D is a finite set of *data values*, which together form the XFA's *data set*. (Smith et al. call this the "data domain".)
- $U : \delta \to \mathcal{P}(D \times D)$, where $U(t)$ gives the data-set transformer ("update function" in Smith's terminology) associated with a transition $t \in \delta$. (Note that here we are taking a relational view of δ.) Each transformer is a binary relation on D, i.e. a subset of $D \times D$. (We will also sometimes abuse notation and treat each transformer as a function $D \to \mathcal{P}(D)$, which is another way of representing a relation.)
- $QD_0 \subseteq Q \times D$ and $F \subseteq Q \times D$ give the sets of initial and accepting configurations, respectively. A *configuration* is simply a pair from $Q \times D$.

As an XFA A consumes input, it tracks both a current state $q \in Q$ as well as a data value $d \in D$, which together form a configuration (q, d). A will start in a configuration in QD_0. If A's current configuration is (q, d) and it reads a symbol σ, it takes two steps to determine the next configuration (q', d'):

First, A nondeterministically chooses the successor state q' from $\delta(q, \sigma)$.

Second, A nondeterministically chooses the successor data value d' from $U(q, \sigma, q')(d)$. That is, it looks up the transition (q, σ, q') in U to obtain a transformer $\tau : D \to \mathcal{P}(D)$, and then chooses a successor from $\tau(d)$. (If $\tau(d) = \varnothing$, the XFA "blocks," just like if $\delta(q, \sigma)$ was empty.)

If it is possible for A to read an input word and end in a configuration in F, then the word is accepted. The language of an XFA is the set of words it accepts.

To avoid ambiguity, we always use an unadorned term *state* to refer to just the Q portion of the XFA, use *data value* for the D portion, and *configuration* for both together.

There are two things to note about the formulation above. First, the process for finding the possible successor configurations is described as a two-step process of finding the next state and then the next data value. An alternative formulation would be to define δ_3 as a function from $Q \times D \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q \times D)$, where the automaton in configuration (q, d) reading σ would choose a successor configuration from $\delta_3(q, d, \sigma)$. These two formulations are actually equivalent; it is merely a question of how δ and U are represented, and no meaningful structural changes need to occur to change between them. The two-stage description was chosen for three reasons: (1) it matches Smith's description, (2) it makes it easier to describe automaton operations, and (3) it aligns the description of XFAs with those of weighted finite automata (WFAs), described in the following section.

Second, XFAs are not more expressive than standard finite automata, but they can be more compact. There is a trivial transformation of an XFA to an FA. The FA's set of states is just $Q \times D$; the FA's initial and final set of states are just QD_0 and F, respectively; and the FA's transition function is δ_3 introduced in the previous paragraph (interpreting $\delta_3(q, d, \sigma)$ as $\delta_3((q, d), \sigma)$). The benefit of XFAs over FAs is that the entire $Q \times D$ state space need not be explicitly materialized, and instead much of the information can be tracked in the data value over an automaton with a small set of states.

3.2 Weighted finite automata [background]

The final form of automata we will define are weighted finite automata (WFAs). WFAs are well-studied in the literature, with applications in areas such as speech recognition [57, 64], machine translation [45], image compression [2], model checking probabilistic systems [9], and program analysis [72, §3.1.3,§3.2]. (Speech recognition and machine translation also use weighted string transducers, which are not covered here.) Definitions of WFAs vary by source (e.g., Mohri [57], Pereira et al. [64], and Ésik and Kuich [33] all provide slightly different formalisms for the same thing); we will give a definition similar to Mohri's that is chosen to mirror that of XFAs.

WFAs associate a weight with each transition, where weights are elements of a semiring:

Definition 3.2. A *semiring* is a tuple $(S, \oplus, \otimes, \overline{0}, \overline{1})$ that meets the following requirements:

- S is a set containing $\overline{0}$ and $\overline{1}$ as distinct elements
- $\oplus$ ("combine") and $\otimes$ ("extend") are binary operators on S
- $(S, \oplus, \overline{0})$ and $(S, \otimes, \overline{1})$ both form monoids. That is: $\oplus$ and $\otimes$ are both associative, $a \oplus \overline{0} = \overline{0} \oplus a = a$ for each $a \in S$, and $a \otimes \overline{1} = \overline{1} \otimes a = a$ for each $a \in S$.
- $\oplus$ is commutative
- $\otimes$ distributes over $\oplus$, i.e. $a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$ and $(a \oplus b) \otimes c = (a \otimes c) \oplus (b \otimes c)$ for each $a, b, c \in S$
- $\overline{0} \otimes a = a \otimes \overline{0} = \overline{0}$ for each $a \in S$

A semiring is *idempotent* if $a \oplus a = a$ for all $a \in S$; we will assume that semirings are idempotent, usually without comment. A *weight* is an element from a semiring.

There can be extend and combine operators indexed over a set, written as $\bigotimes_{i \in I} w_i$ and $\bigoplus_{i \in I} w_i$. By definition, $\bigotimes_{i \in \varnothing} w_i = \overline{1}$ and $\bigoplus_{i \in \varnothing} w_i = \overline{0}$.

For example, the natural numbers and the usual arithmetic operations forms a non-idempotent semiring $(\mathbb{N}, 0, 1, +, \cdot)$. An important and common (idempotent) semiring is the *tropical semiring* (also known as the *min-plus* semiring) $(\mathbb{R} \cup \{\infty\}, \infty, 0, \min, +)$.

We will often treat the operators and $\overline{0}/\overline{1}$ elements of a semiring as implicit, identifying the whole semiring with the set S. We define an ordering on a semiring by defining $a \sqsubseteq b$ iff $a \oplus b = b$. (For example, in the tropical semiring, $\sqsubseteq$ coincides with $\geqslant$: $a \geqslant b$ iff $a \min b = b$.) A semiring's *height* is the maximum number of distinct elements $s_1, s_2, \cdots, s_n$ such that $s_1 \sqsubseteq s_2 \sqsubseteq \cdots \sqsubseteq s_n$, or the height is infinite if no such maximum exists. A semiring is *bounded* if there are no infinite ascending chains in $\sqsubseteq$.

Note that $\sqsubseteq$ is a partial order, and the semiring forms a join semilattice where $\oplus$ is join. Many algorithms "move up" in the lattice defined by $\sqsubseteq$, and the boundedness property is hence important for arguing termination. The semiring that we use in our applications (defined in Section 3.3) has finite height and is thus bounded.

Definition 3.3. Given a semiring S, a (nondeterministic) *weighted finite automaton* on S is a tuple $(Q, \Sigma, \delta, W, W_0, W_f)$ where:

- Q is a set of states, Σ an alphabet, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$ the transition function (which we will sometimes treat as a relation)
- $W : \delta \to S$ associates each transition with a weight
- $W_0 : Q \to S$ associates each state with an *initial weight*
- $W_f : Q \to S$ associates each state with a *final weight*

Suppose we are given a string s and WFA A. A *path through* A *matching* s is a sequence of transitions $\pi = t_1, t_2, \cdots t_{|s|}$ where the target state of each t_i matches the source state of t_{i+1} and the symbol labeling t_i matches the ith symbol of s. We extend the definition of W to paths by taking the extend of the weights along the path:

$$W(\pi) \triangleq \bigotimes_{i=1}^{|s|} W(t_i).$$

Finally, we will extend W again to accept *strings* as input: if we let $\mathrm{paths}(s)$ be the set of paths in A matching s, then

$$W(s) \triangleq \bigoplus_{\pi \in \mathrm{paths}(s)} \left[W_0(\mathrm{source}(t_1)) \otimes W(\pi) \otimes W_f(\mathrm{target}(t_{|s|})) \right].$$

A WFA can be viewed as defining two constructs, depending on the application. The first is a mapping from strings to weights, defined by the final extension of W above. (In the literature, such a mapping is often called a *formal power series*.) The second is a language, defined as the set of strings with non-zero weight, $L(A) = \{s \in \Sigma^* \mid W(s) \neq \bar{0}\}$. We will be interested in the language interpretation.

3.3 Interpreting an XFA as a WFA [adaption/new]

It is possible to interpret an XFA as a WFA, and in so doing we can apply WFA operations to an XFA.

First, given an XFA's data set D, binary relations on D form a semiring:

- The set S is $\mathcal{P}(D \times D)$
- $\bar{0}$ is the empty relation $\varnothing$
- $\bar{1}$ is the identity relation $\{(d, d) \mid d \in D\}$
- $\oplus$ is the union of the tuples in the two relations
- $\otimes$ is relational composition: $E \otimes F = E; F = \{(\alpha, \gamma) \mid \exists \beta : (\alpha, \beta) \in E \text{ and } (\beta, \gamma) \in F\}$

This trivially satisfies the requirements to be a bounded, idempotent semiring.

Given an XFA $A = (Q, D, \Sigma, \delta, U, QD_0, F)$, we can build a WFA $B = (Q, \Sigma, \delta, W, W_0, W_f)$ where:

- $W = U$
- $W_0(q) \triangleq \{(d, d) \mid (q, d) \in QD_0\}$
- $W_f(q) \triangleq \{(d, d) \mid (q, d) \in F\}$

Essentially all we are doing is interpreting each transformer in U as a weight, and then making the initial and final weight of each state match the original XFA.

The only trick is how to handle initial and final configurations. In B, the initial and final weights have to be *weights*, which correspond to transformers, while in A they are only values. One way to look at this disparity is to view each transformer as being a $|D| \times |D|$ Boolean matrix (where position i, j is 1 iff (d_i, d_j) is in the transformer), and then note that the XFA uses $1 \times |D|$ vectors (resp., $|D| \times 1$ vectors) as initial (resp., final) "weights", where position $1, i$ (resp., $i, 1$) is 1 iff d_i is an initial (resp., final) data value for the given state). WFAs do not allow us to do that, as we only have $|D| \times |D|$ transformers. However, this is OK: because we are only interested in whether the overall weight of a string is $\overline{0}$ or not, all we need to do is create a transformer that preserves the "zero-ness," and those above perform that task.

The following example illustrates the equivalence for when $D = \{d_1, d_2\}$. Suppose we are given a path π that starts at state q_0 and ends at state q_n; to determine whether that path is actually feasible, we compute the effect on the data state over that path, including initial and final values. If the path is feasible (i.e., it is possible to always choose a next data successor that respects U), we will say that π is a *witness* that the corresponding string accepts, or, for brevity, that the path accepts.

Using the XFA interpretation, we would see something like the following:

$$\begin{array}{cc} d_0 & d_1 \end{array} \qquad \begin{array}{cc} d_0 & d_1 \end{array}$$
$$\begin{bmatrix} a & b \end{bmatrix} \times \begin{matrix} d_0 \\ d_1 \end{matrix} \begin{bmatrix} c & d \\ e & f \end{bmatrix} \times \begin{matrix} d_0 \\ d_1 \end{matrix} \begin{bmatrix} g \\ h \end{bmatrix} = \begin{bmatrix} acg + beg + adh + bfh \end{bmatrix}$$

The first matrix (on the left-hand side of the equality) holds information about what the initial data values are for the state at the start of the path in question (e.g., a is 1 iff $(q_0, d_1) \in QD_0$). The second matrix is the composition of all of the transformers along the path (e.g., c is 1 iff $(d_1, d_1) \in W(\pi)$). The third matrix holds information about what final data values are for the state at the end of the path (e.g., h is 1 iff $(q_n, d_2) \in F$). On the right-hand side, the multiplications should be interpreted as *logical and* and the additions as *logical or*. Thus this path is a witness to the corresponding string being accepted when at least one of of acq, beg, adh, and bfh are 1.

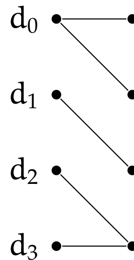
Using the WFA interpretation, the operations would be expressed as:

$$\begin{array}{c} \\ d_0 \\ d_1 \end{array}\begin{array}{c} \begin{array}{cc} d_0 & d_1 \end{array} \\ \begin{bmatrix} a & 0 \\ 0 & b \end{bmatrix} \end{array} \times \begin{array}{c} \begin{array}{cc} d_0 & d_1 \end{array} \\ \begin{bmatrix} c & d \\ e & f \end{bmatrix} \end{array} \times \begin{array}{c} \begin{array}{cc} d_0 & d_1 \end{array} \\ \begin{bmatrix} g & 0 \\ 0 & h \end{bmatrix} \end{array} = \begin{array}{c} \\ d_0 \\ d_1 \end{array}\begin{array}{c} \begin{array}{cc} d_0 & d_1 \end{array} \\ \begin{bmatrix} acg & adh \\ beg & bfh \end{bmatrix} \end{array}$$

Here, the first matrix is $W_0(q_0)$ as defined above, the second is $W(\pi)$, and the third is $W_f(q_n)$. This path is a witness to the corresponding string being accepted iff the final weight is non-$\overline{0}$, which happens whenever at least one of acq, adh, beg, and bfh is non-$\overline{0}$. ($\overline{0}$ in this domain is $\left[\begin{smallmatrix} 0 & 0 \\ 0 & 0 \end{smallmatrix}\right]$.) This condition is exactly the same condition as that in the previous paragraph.

The W_0 and W_f constructions above are not the only possible choices; for instance, setting the initial weight to $\left[\begin{smallmatrix} a & b \\ 0 & 0 \end{smallmatrix}\right]$ and/or the final weight to $\left[\begin{smallmatrix} g & 0 \\ h & 0 \end{smallmatrix}\right]$ would also work. However, if we are actually interested in what initial data values lead to the path being accepted, or what data values are actually reachable in the final state, then the original definition provides the most information. For example, adg is true iff it is possible for the XFA to start in (q_0, d_0), follow the given path, and finish in (q_n, d_1).

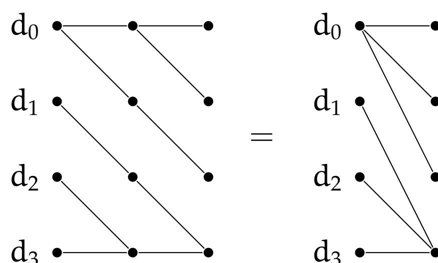
Finally, I wish to introduce a type of diagram that will be used later in the dissertation. The following diagram illustrates a binary relation on $D = \{d_0, d_1, d_2, d_3\}$:



Each dot represents an element from D. The dots in the left column are the *pre-state* and the dots in the right column are the *post-state*, and we will refer to those in the right with primed names. There is a line from d_i on the left to d'_j on the right iff (d_i, d'_j) is in the relation being represented. The above diagram then represents the relation $\{(d_0, d_0),$ $(d_0, d_1), (d_1, d_2), (d_2, d_3), (d_3, d_3)\}$. Call this relation R.

Visually computing the $\otimes$ of two or more relations is easy: simply concatenate the diagrams for each of the relations so that they share dots between the right-hand column of

the first relation and the left-hand column of the second, then see what paths exist between the left column of the first relation and the right column of the last. The following diagram illustrates $R \otimes R$:

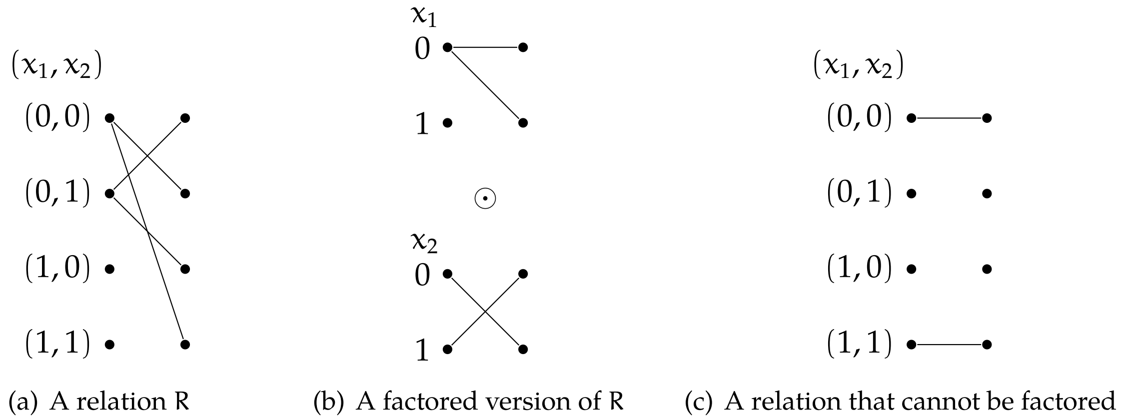


In Chapter 6, each data value will have some structure to it: it will be an assignment to a set of variables $\{x_1, x_2, \cdots, x_n\}$, where each variable takes on a value from the set $\{0, 1, 2, \cdots, 2^m - 1\}$. Thus a data value is some tuple like $(0, 1, 3, 2)$, and there are 2^{nm} possible data values. In order to visualize relations on the set of these values, we use the following convention. Suppose a relation τ can be factored into $\tau_1 \odot \tau_2 \odot \cdots \tau_k$, where each τ_i is on a subset of the variables and $\odot$ is the tensor product of the relations taken in the natural way, combining the variable subsets to match τ. (See Definition 3.5 in Section 3.5 and the subsequent text for more information about tensor products.) In this case, we will illustrate each of the τ_is instead. Figure 3.1 provides an example of factoring a relation in this way, along with a relation that cannot be factored. Finally, it will often be the case that all but one τ_i will be the identity relation on the variable in question; in this case, we will just display the single τ_i that is not.

We will draw XFAs showing their transformers illustrated using diagrams such as these, where there will be one diagram per transition. For states that are part of an accepting configuration, in the transformer diagrams on incoming transitions we will sometimes circle the data values that will accept in concert with that state. Figure 3.5 has an example of a very simple XFA with this convention.

3.4 Symbolic ε closure [new/adaption]

Smith et al. provide an algorithm for performing ε-closure on an XFA, starting from a specific configuration (q, d). This algorithm is shown in Listing 3.2.



(a) A relation R (b) A factored version of R (c) A relation that cannot be factored

Figure 3.1: Factoring a relation. There are two variables x_1 and x_2; each can be 0 or 1.

1 **ComputeEpsilonReachable(q, d)**
2 *result* $= \{(q, d)\}$;
3 **foreach** $(q_i, d_i) \in$ *result* **do**
4   **foreach** $q_f \in \{q \mid (q_i, \varepsilon, q) \in \delta\}$ **do**
5   *result* = *result* $\cup (\{q_f\} \times \{d_f \mid (d_i, d_f) \in U(q_i, \varepsilon, q_f)\})$;
6 **return** *result*;

Listing 3.2: Smith et al.'s ε-closure procedure [76, Algorithm 1]

However, Smith's algorithm is not well-suited for problems with large data sets. The reason is that, as can be seen on Line 3 of Listing 3.2, it iterates over each configuration of the ε closure explicitly. Our goal was to work with data sets that were very large (originally we had hoped that we could track multiple 32-bit values), and we looked for ways to perform this operation symbolically.

In this section, we describe two such symbolic techniques. First, we briefly mention an algorithm due to Mohri [58, §3.2]. Second, we describe a technique that is based on the Tarjan algorithm for solving the path-expression problem [80]:

Definition 3.4. Given a directed graph G and vertex s, the *single-source path-expression problem* asks us to find a regular expression $E(s, v)$ for each vertex v, such that $E(s, v)$ represents all paths from s to v. (The alphabet of the regular expression is the set of edges in G.)

The specific implementation we use realizes Tarjan's algorithm using fast weighted pushdown systems [47]. Our experience is that the Tarjan/FWPDS algorithm is significantly faster than Mohri's; see Section 3.4.1.

Mohri calls his algorithm a "shortest distance" algorithm over semirings. This may seem odd, but the more typical sense of shortest distances on a graph can be obtained by using the tropical semiring. (In fact, many common graph problems and algorithms can be viewed as a more general algorithm specialized to the tropical semiring.) Mohri claims that his algorithm is a non-trivial generalization of standard shortest-path algorithms; when applied with the tropical semiring, the result matches either the Bellman-Ford or the Dijkstra algorithm depending on the choice of queue. For more information, please see Mohri's description [58].

The second algorithm we considered applies Tarjan's algorithm and existing work on fast weighted pushdown systems (FWPDSs). What ε closure must do is, given a source node q, compute the net effect of the transformers along every ε path from q to each state q'. The "net effect" is simply the combine-over-all-paths value from q to q' using only ε transitions — and this sort of problem is exactly what WPDSs are designed to solve. The combine-over-all-ε-paths value from q to q' is defined as

$$\bigoplus_{\pi \text{ is an } \varepsilon \text{ path from } q \text{ to } q'} W(\pi).$$

What we do is convert the XFA's ε transitions into an (F)WPDS, then issue a *post** query from the configuration set corresponding to $\{q\}$. The answer to the ε-closure query can be read off from the *post** result, and is a mapping from states to the combine-over-all-paths weight to that state.

Readers who are familiar with the definition of WPDSs (see, for example, [72, Defs. 2 and 6]) can see Fig. 3.3 for details; others can feel free to skip it.

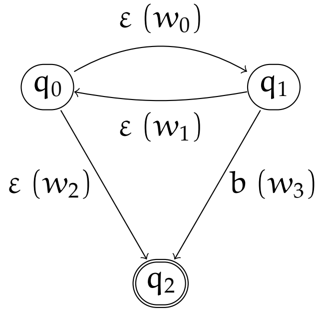
While in principle it is possible to use standard WPDS algorithms to solve *post**, we use the FWPDS *post** instead [47].[2] FWPDS reachability algorithms have their basis in Tarjan's algorithm for solving the path-expression problem (Definition 3.4).

[2]For general program analysis, Lal extended Tarjan's algorithm to handle interprocedural problems, which resulted in FWPDSs, but we do not actually take advantage of the interprocedural capabilities. Thus, FWPDS are in some sense an implementation detail of how we use Tarjan's algorithm.

Given a WFA $(Q, \Sigma, \delta, W, W_0, W_f)$, we create a WPDS $(\mathcal{P}, S, f)$ where $\mathcal{P} = (P, \Gamma, \Delta)$ is a PDS and:

- $P = \{p\}$
- $\Gamma = Q$
- $\Delta = \{\langle p, q\rangle \hookrightarrow \langle p, q'\rangle \mid q' \in \delta(q, \varepsilon)\}$
- S is the same as the WFA's semiring
- $f(\langle p, q\rangle \hookrightarrow \langle p, q'\rangle) = W(q, \varepsilon, q')$

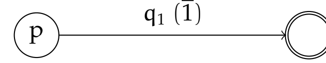
(a) Converting a WFA to a WPDS



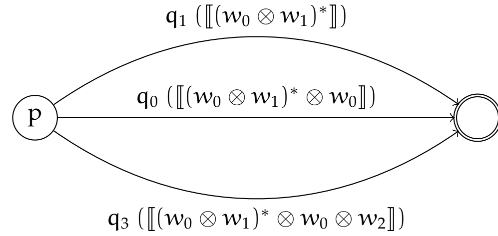
(b) An example WFA

- $\langle p, q_0\rangle \hookrightarrow \langle p, q_1\rangle$ with weight w_0
- $\langle p, q_1\rangle \hookrightarrow \langle p, q_0\rangle$ with weight w_1
- $\langle p, q_0\rangle \hookrightarrow \langle p, q_2\rangle$ with weight w_2
- $\langle p, q_1\rangle \hookrightarrow \langle p, q_2\rangle$ with weight w_3

(c) The rules in the WPDS created when converting the WFA in (b) to a WPDS.



(d) The query constructed for $\varepsilon\mathtt{close}(q_1)$



(e) Example *post** result. The edge weights are notated as $[\![e]\!]$ for some expression e to emphasize that the result has the *evaluation* of e rather than the expression itself. For a weight w, w^* is defined to be $\overline{1} \oplus w \oplus w \otimes w \oplus \cdots$.

Figure 3.3: Solving WFA ε closure via translation to an (F)WPDS. To compute $\varepsilon\mathtt{close}(q)$ on a WFA, the WFA is first converted into a WPDS using the procedure specified in (a). An example WFA is shown in (b) and the corresponding WPDS rules in (c). Then a *post** query is made on the resulting WPDS; for the example, the query and result are shown in (d) and (e). The result of $\varepsilon\mathtt{close}$ can be read directly off of the result automaton: in this case, $\varepsilon\mathtt{close}(q_1) = \{q_1 \mapsto (w_0 \otimes w_1)^*, q_2 \mapsto (w_0 \otimes w_1)^* \otimes w_1, q_3 \mapsto (w_0 \otimes w_1)^* \otimes w_1 \otimes w_2\}$.

Many graph problems can be answered by reinterpreting the leaves and operators in a regular expression appropriately [81]. We are interested in the following reinterpretation, which matches what is used for dataflow analysis:

- $\varnothing$ is reinterpreted as $\overline{0}$
- ε is reinterpreted as $\overline{1}$
- A leaf e names an edge (q, q'); we reinterpret it as $W(q, \varepsilon, q')$
- Concatenation is reinterpreted as $\otimes$
- Alternation is reinterpreted as $\oplus$
- Kleene star of a weight w is reinterpreted as $w^* = \overline{1} \oplus w \oplus (w \otimes w) \oplus (w \otimes w \otimes w) \oplus \cdots$ (but see below for more discussion)

The result of this reinterpretation is the combine-over-all-paths value over the paths that the regular expression represents. In our case, it is the ε closure from a source state to a target state.

One benefit from using Tarjan's algorithm is the ability to compute w^* more directly. A naïve approach would compute this value by computing w^k for successively larger k until $\bigoplus_{i=0}^{k} w^i$ does not change.[3] If the semiring has height h, this will potentially require $\mathcal{O}(h)$ extend and combine operations.

However, w^* can be computed more efficiently. Let $f(k) = \bigoplus_{i=0}^{k} w^i$ and let k_0 be the value of k at which the procedure given above stabilizes. (In other words, $f(k_0 - 1) \neq f(k_0) = f(k_0 + 1)$.) Note that $f(k_0) = f(k)$ for all $k \geqslant k_0$. Now, $(\overline{1} \oplus w)^k = f(k)$ because we are assuming that the semiring is idempotent. The expression $(1 \oplus w)^k$ can be evaluated using repeated squaring: computing $(\overline{1} \oplus w)$, then $(\overline{1} \oplus w)^2$, then $(\overline{1} \oplus w)^4$, etc. until convergence. This procedure only needs one combine operation at the start plus $\mathcal{O}(\log_2 h)$ extend operations, which is improvement that is exponential in the semiring height.

There is a caveat with this, which is that these complexity measurements assume that the $\otimes$ operations are constant time. It is possible that repeated squaring leads to each operating being more expensive and this reduces or eliminates the benefit from repeated squaring [47, Footnote 3].

[3] Exponentiation is repeated extends: $w^0 \triangleq \overline{1}$ and $w^i \triangleq w^{i-1} \otimes w$.

Using Tarjan's algorithm also brings other benefits. One benefit is that using Tarjan's algorithm eliminates the need to choose a good order in which to visit states in the automaton. (Or viewed another way, Tarjan's algorithm chooses a good order.) A second benefit is that Tarjan's algorithm can better take advantage of the semiring's associativity and distributivity properties. Suppose we have a sequence of states $q_1, q_2, \cdots, q_n$ where each state until q_n has the following state as its sole successor (and all transitions between them are εs). Using Tarjan's algorithm will let us compute the extend of the path from q_1 to q_n *once*, and then use the resulting value in the future to propagate new information from q_1 directly to q_n; this behavior falls naturally out of Tarjan's algorithm as the regular expression is reinterpreted. Algorithms based on propagating weights around the graph (e.g., the traditional WPDS algorithm or Mohri's) would need to take special steps to avoid propagating a new weight across *every transition* between q_1 and q_n every time the weight on q_1 changes.

Tarjan's algorithm computes the path regular expression in time $\mathcal{O}(m \log n)$ for a reducible graph with m edges and n vertexes. Reinterpreting the regular expression takes $\mathcal{O}(m \log n \log h)$ semiring operations, where h is the semiring's height [47, 80]. (As mentioned above, the $\log h$ factor may be worse in practice in terms of actual time if the extend operations become expensive.) For irreducible graphs the running time can degrade to $\mathcal{O}(n^3)$; however, Lal et al. did not find much performance degradation due to irreduciblity.

3.4.1 Performance comparison of ε closure methods

In this section, we briefly describe some empirical measurements of the performance of various options for ε closure. We compare the following choices of algorithms:

- Mohri's algorithm
- WPDS (iterated propagation)
- FWPDS (Tarjan's)

	Mohri			WPDS			FWPDS		
Benchmark	Sec.	# ⊗s	# ⊕s	Sec.	# ⊗s	# ⊕s	Sec.	# ⊗ s	# ⊕ s
32-cycle/6-bits	5.98	5,378	9,362	6.61	5,444	9,202	2.04	1,570	1,170
32-cycle/8-bits	43.7	17,666	33,938	41.6	17,732	33,778	6.76	1,634	1,170
png2ico/1-bit	17.5	307,393	250,513	15.1	319,845	263,214	15.5	237,709	134,561
png2ico/2-bit	120.0	345,041	315,990	97.9	378,179	379,882	96.5	237,721	134,562

Table 3.4: ε-closure algorithm performance. "#⊗s" and "#⊕s" give the number of times the operation was called.

The benchmarks we measured are as follows:

- A 32-cycle version of the benchmark described in Section 6.3.1, with 6 bits per logical variable (*32-cycle/6-bits*)
- The same 32-cycle benchmark with 8 bits per logical variable (*32-cycle/8-bits*)
- An XFA inferred from the program *png2ico* with 1 bit per logical variable (*png2ico/1-bit*)
- The *png2ico* XFA with 2 bits per logical variable (*png2ico/2-bits*)

All tests measure the time to read in the given automaton, build the transformers, and determinize it. In all cases, the optimization described in Section 6.3.1 is active. The time reported is the mean of 5 runs.

As can be seen by the synthetic benchmark (the *32-cycle* tests), there can be a significant difference in performance between the ε closure methods.

3.5 State-Determinization [adaption/background]

To be fully deterministic, an XFA needs to satisfy two requirements:

- For all $q \in Q$ and $\sigma \in \Sigma$, $|\delta(q, \sigma)| \leqslant 1$. (We use $\leqslant$ here, but it would be equally reasonable to use $=$.)
- For every transition $t \in \delta$ and data value $d \in D$, $|U(t)(d)| \leqslant 1$.

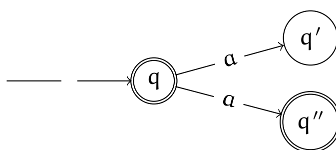
In this section, we describe how to achieve the first of these requirements; such an XFA we call *state-deterministic*. (Mohri uses the term *subsequential* for the equivalent notion on WFAs [57].) We will not create an XFA that meets the second requirement explicitly; we compensate using other means during the containment check (Section 3.6).

State-determinizing an XFA consists of two steps:

First, in preparation for the following steps, the data set of the XFA is lifted from D to $Q \times D$ and transitions are adjusted accordingly

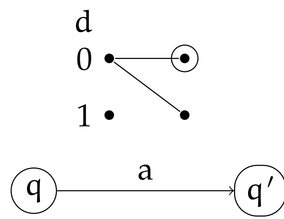
Second, the state and transition portion of the XFA are determinized using the standard algorithm; transformers are computed as described below. Again, we use the term *cell* to refer to a state in the determinized XFA, to distinguish it from states in the original nondeterministic automaton. Thus a cell is a set of states.

The reason producing a fully-deterministic XFA would be attractive can be illustrated as follows. Consider *why* we might want to determinize standard FAs. Determinization is useful for two reasons: first, it makes checking a concrete input string faster ($\mathcal{O}(|s|)$ instead of potentially $\mathcal{O}(|s| \cdot |Q|)$ or even higher), and second, it makes it possible to check universality and containment. Both reasons apply equally well in the data domain. First, when reading a concrete string, the XFA tracks the current set of possible configurations. If there is nondeterminism in the data value, then the XFA needs to track each of the possibilities.[4] Second, the same problems that arise in standard FAs when performing universality and containment testing can arise. In a standard FA, the difficulty with deciding universality is that one needs to look at *all* possible paths for a string. It is not possible to look at the following automaton and determine that it is not universal simply by the existence of the non-accepting path $q \xrightarrow{a} q'$:



Similarly, nondeterminism in the data value means that, even if the path through the state portion of the XFA is fixed, different choices of data value successors could reach some

[4]It is only necessary to track a single *weight*, but doing so is, in context, as expensive as tracking a set of states when interpreting an NFA.



- $Q = \{q, q'\}$, $\Sigma = \{a\}$, $\delta = \{(q, a, q')\}$
- $D = \{0, 1\}$
- $U(q, a, q') = \{(0, 0), (0, 1)\}$
- $QD_0 = \{(q, 0), (q, 1)\}$
- $F = \{(q', 0)\}$

Figure 3.5: Example XFA with nondeterminism in the data value. For completeness, the definition of the XFA is shown to the right.

accepting and some rejecting configurations. Figure 3.5 shows an example XFA where the configurations $(q', 0)$ and $(q', 1)$ are both reachable and $(q', 0)$ is accepting (i.e., it is a member of F) and $(q', 1)$ is not. This is essentially exactly the same situation as the FA illustrated a moment ago. Later, in Section 3.6.1, Fig. 3.7 illustrates this point from another angle.

Figure 3.6 gives an example XFA and the result of both determinization steps.

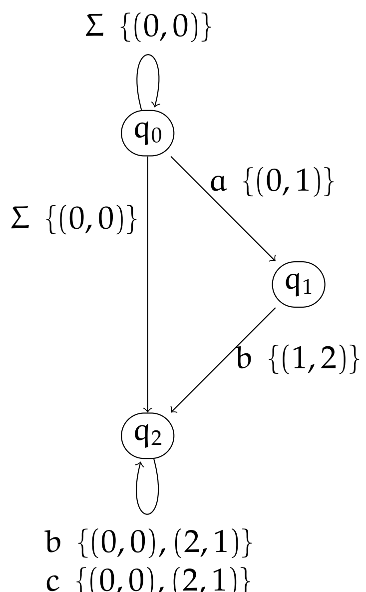
Like ε closure, Smith et al. give an algorithm for determinizing XFAs [76, Algs. 2 and 3], but we considered it unsuited for this application as-is, for the same reasons: we ddid not want to materialize $\mathcal{P}(D)$. (Note that the first two steps are combined in Smith's version.) Here we do a reformulation of the algorithm, and generalize it to WFAs for which we can provide an appropriate tensor product operation.

Definition 3.5. Suppose that we are given three semirings S_1, S_2, and S_3 (where components are subscripted to correspond). The *tensor product* operation is a function $\odot : S_1 \times S_2 \to S_3$ that satisfies the following requirements:

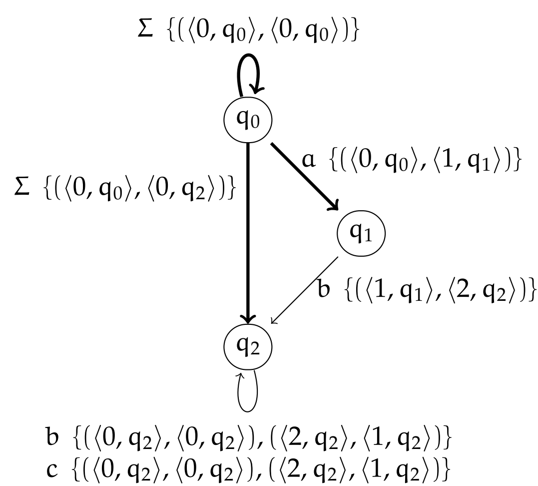
- $(a \odot b) \otimes_3 (a' \odot b') = (a \otimes_1 a') \odot (b \otimes_2 b')$ for each $a, a' \in S_1$ and $b, b' \in S_2$
- $a \odot b = \overline{0}_3$ iff $a = \overline{0}_1$ or $b = \overline{0}_2$

Let S_Q be the semiring of binary relations over Q. (The semiring's components are defined the same as the semiring of binary relations over D defined in Section 3.3.) We will sometimes use the notation $q \to q'$ instead of (q, q') for tuples in S_Q.

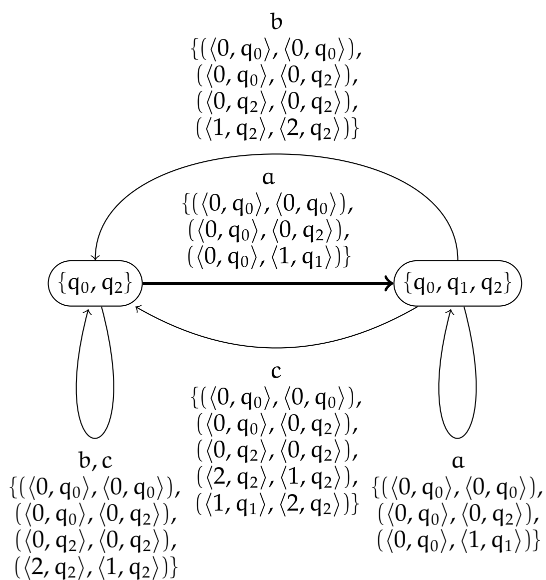
Suppose that we are given a WFA $A = (Q, \Sigma, \delta, W, W_0, W_f)$ over semiring S. To determinize A, we need to be given need two additional constructs: we need a semiring S_T and



(a) Example XFA with $D = \{0, 1, 2\}$. Edges are labeled with their symbol and relation.



(b) The result of the first determinization step, which lifts each weight to encode the state transfers as well. (The bold transitions are presentation indications for the following step.)



(c) The result of the second determinization step, which determinizes transitions. The bold transition on a results from the three bold transitions in (b), and its weight is the $\oplus$ of the weights of those three transitions; for XFAs, the $\oplus$ is simply the set union.

Figure 3.6: Example XFA and determinization [76, Fig. 8].

a tensor product operation $\odot : S_Q \times S \to S_T$. In general, these can be any semiring and operation that obey the requirements above.

In the context of XFAs, S_T is the semiring of binary relations over $Q \times D$ (we will use $\langle q, d \rangle$ for the tuple (q, d)), and $\odot$ is defined as follows:

$$t \odot u = \{ (\langle q, d \rangle, \langle q', d' \rangle) \mid (q, q') \in t \text{ and } (d, d') \in u \}$$

If we interpret transformers as $|D| \times |D|$ Boolean matrices and an element of S_Q as $|Q| \times |Q|$ Boolean matrices, then $\odot$ matches the Kronecker product of the matrices. For example, if $D = \{d_1, d_2\}$ and $Q = \{q_1, q_2\}$, then:

$$t \odot u = \begin{array}{c} \\ q_1 \\ q_2 \end{array} \begin{array}{c} \begin{array}{cc} q_1 & q_2 \end{array} \\ \begin{bmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{bmatrix} \end{array} \odot \begin{array}{c} \\ d_1 \\ d_2 \end{array} \begin{array}{c} \begin{array}{cc} d_1 & d_2 \end{array} \\ \begin{bmatrix} u_{11} & u_{12} \\ u_{21} & u_{22} \end{bmatrix} \end{array}$$

$$= \left[\begin{array}{c|c} t_{11} \cdot u & t_{12} \cdot u \\ \hline t_{21} \cdot u & t_{22} \cdot u \end{array} \right]$$

$$= \begin{array}{c} \\ \langle q_1,d_1 \rangle \\ \langle q_1,d_2 \rangle \\ \langle q_2,d_1 \rangle \\ \langle q_2,d_2 \rangle \end{array} \begin{array}{c} \begin{array}{cccc} \langle q_1,d_1 \rangle & \langle q_1,d_2 \rangle & \langle q_2,d_1 \rangle & \langle q_2,d_2 \rangle \end{array} \\ \left[\begin{array}{cc|cc} t_{11}u_{11} & t_{11}u_{12} & t_{12}u_{11} & t_{12}u_{12} \\ t_{11}u_{21} & t_{11}u_{22} & t_{12}u_{21} & t_{12}u_{22} \\ \hline t_{21}u_{11} & t_{21}u_{12} & t_{22}u_{11} & t_{22}u_{12} \\ t_{21}u_{21} & t_{21}u_{22} & t_{22}u_{21} & t_{22}u_{22} \end{array} \right] \end{array}$$

This definition of $\odot$ has the tensor product property:

$$\begin{aligned} &(A \odot B) \otimes (A' \odot B') \\ &\quad = \left\{ (\langle a_1, b_1 \rangle, \langle a_3, b_3 \rangle) \;\middle|\; \begin{array}{l} \exists \langle a_2, b_2 \rangle : (\langle a_1, b_1 \rangle, \langle a_2, b_2 \rangle) \in (A \odot B) \\ \qquad \text{and } (\langle a_2, b_2 \rangle, \langle a_3, b_3 \rangle) \in (A' \odot B') \end{array} \right\} \\ &\quad = \left\{ (\langle a_1, b_1 \rangle, \langle a_3, b_3 \rangle) \;\middle|\; \begin{array}{l} \exists \langle a_2, b_2 \rangle : ((a_1, a_2) \in A \text{ and } (b_1, b_2) \in B) \\ \qquad \text{and } ((a_2, a_3) \in A' \text{ and } (b_2, b_3) \in B') \} \end{array} \right\} \end{aligned}$$

$$= \left\{ (\langle a_1, b_1\rangle, \langle a_3, b_3\rangle) \;\middle|\; \begin{array}{r} \exists\langle a_2, b_2\rangle :((a_1, a_2) \in A \text{ and } (a_2, a_3) \in A') \\ \text{and } ((b_1, b_2) \in B \text{ and } (b_2, b_3) \in B') \end{array} \right\}$$

$$= \left\{ (\langle a_1, b_1\rangle, \langle a_3, b_3\rangle) \;\middle|\; \begin{array}{l} (\exists a_2 : (a_1, a_2) \in A \text{ and } (a_2, a_3) \in A') \\ \text{and } (\exists b_2 : (b_1, b_2) \in B \text{ and } (b_2, b_3) \in B') \end{array} \right\}$$

$$= \left\{ (\langle a_1, b_1\rangle, \langle a_3, b_3\rangle) \mid (a_1, a_3) \in (A \otimes A') \text{ and } (b_1, b_3) \in (B \otimes B') \right\}$$

$$= (A \otimes A') \odot (B \otimes B')$$

Finally, before we get to the determinization algorithm, I will briefly mention a different algorithm for determinization described by Mohri [57, §3.3]. Mohri's algorithm differs substantially from the one presented here. In particular, it imposes a constraint on the WFA's semiring that it have multiplicative inverse (i.e. $\forall a \neq \overline{0} \in S : \exists a^{-1} \in S : a \otimes a^{-1} = \overline{1}$). Many of the weights that we create in this dissertation do not have inverses, and so we cannot use Mohri's algorithm. However, it is interesting to note that Mohri also has to use a new domain which is $Q \times S$ in the determinized automaton, which is essentially the same as the first step described here.

The following two subsections will walk through the first two determinization steps. Suppose that we are given an WFA $A = (Q, \Sigma, \delta, W, W_0, W_f)$ over semiring S and a tensor product operation from $S \times S_Q$ to S_T. We will define new automata A^L and A^{SD}, reflecting results of the first two steps, respectively. The components of each automaton will be superscripted to match its name, e.g. $A^L = (Q^L, D^L, \Sigma, \delta^L, W^L, W_0^L, W_f^L)$, except that Σ remains constant throughout. The third step we handle in a somewhat different way during the inclusion test, which is discussed in Section 3.6.

3.5.1 Lifting the data set from D to $Q \times D$ and adjusting transformers

It is possible for an XFA that is reading a string s to be in a set of configurations such as $\{(q, 0), (q, 1), (q', 0)\}$; see Fig. 3.6. The determinized machine must respect that same set of configurations (or at least something isomorphic to it). However, if the XFA simply tracked that states q and q' are reachable and data values 0 and 1 are reachable, it will lose the fact that $(B, 1)$ is not reachable. The second determinization step will collapse A and B into a single cell as in the standard algorithm, which means that the original three configurations

need to be tracked in the data value. Keeping these separate will be the job of S_T (with support from S_Q and $\odot$).

We define A^L (the L for "lifted") in terms of A:

- Q and δ are unchanged: $Q^L = Q$ and $\delta^L = \delta$.
- W^L is over the semiring S_T instead of S
- Every weight is "lifted" to S_T by taking the tensor product with a natural element from S_Q:
  - $W^L(q, \sigma, q') = W(q, \sigma, q') \odot \{q \to q'\}$
  - $W_0^L(q) = W_0(q) \odot \{q \to q\}$
  - $W_f^L(q) = W_f(q) \odot \{q \to q\}$

This construction encodes endpoints of each transition in its weight. This extra information is important for the next step of the determinization process, which will compute the $\oplus$ of the weights over multiple edges. For that process to be correct, we will need it to be the case that, for transitions t_1 and t_2, $W(t_1) \otimes W(t_2) = \overline{0}$ if the target of t_1 does not equal the source of t_2.

To show that the language of A^L is the same as that of A, we need the following simple fact:

Lemma 3.6. Given an idempotent semiring S and a set of elements $\{w_i \mid i \in I\}$, the following holds:

$$\bigoplus_{i \in I} w_i = \overline{0} \quad \text{if and only if} \quad \forall i \in I : w_i = \overline{0}$$

Proof. Recall that a semiring is idempotent if $a \oplus a = a$. Assume $\bigoplus_{i \in I} w_i = \overline{0}$. Then for each $i \in I$:

$$w_i = w_i \oplus \overline{0} = w_i \oplus \bigoplus_{j \in I} w_j = (w_i \oplus w_i) \oplus \bigoplus_{\substack{i \in I \\ j \neq i}} w_j = w_i \oplus \bigoplus_{\substack{i \in I \\ i \neq j}} w_j = \bigoplus_{j \in I} w_j = \overline{0}$$

□

Now we can show that $L(A^L) = L(A)$. Let s be a string; we will show that $W^L(s) = \overline{0}$ iff $W(s) = \overline{0}$, which means that the languages are the same.

$$
\begin{aligned}
W^L(s) = & \bigoplus_{\pi = t_1 \cdots t_n \in \mathrm{paths}(s)} \underbrace{\left[W_0^L(\mathtt{src}(t_1) \otimes \bigotimes_{t \in \pi} W^L(t) \otimes W_f^L(\mathtt{tgt}(t_n)) \right]}_{\neq \overline{0} \text{ when } \pi \text{ witnesses } s \in L(A)} \\
= & \bigoplus_{\pi \in \mathrm{paths}(s)} \Big[\big(W_0(\mathtt{src}(t_1)) \odot \{\mathtt{src}(t_1) \to \mathtt{src}(t_1)\} \big) \\
& \otimes \bigotimes_{t \in \pi} (W(t) \odot \{\mathtt{src}(t) \to \mathtt{tgt}(t)\}) \\
& \otimes \big(W_f(\mathtt{tgt}(t_n)) \odot \{\mathtt{tgt}(t_n) \to \mathtt{tgt}(t_n)\} \big) \Big] \\
= & \bigoplus_{\pi \in \mathrm{paths}(s)} \Big[\big(W_0(\mathtt{src}(t_1)) \otimes \bigotimes_{t \in \pi} W(t) \otimes W_f(\mathtt{tgt}(t_1)) \big) \\
& \odot \big(\{\mathtt{src}(t_1) \to \mathtt{src}(t_1)\} \otimes \bigotimes_{t \in \pi} \{\mathtt{src}(t) \to \mathtt{tgt}(t)\} \otimes \{\mathtt{tgt}(t_n) \to \mathtt{tgt}(t_n)\} \big) \Big] \\
= & \bigoplus_{\pi \in \mathrm{paths}(s)} \underbrace{\Big[\Big(W_0(\mathtt{src}(t_1)) \otimes W(\pi) \otimes W_f(\mathtt{tgt}(t_1)) \Big) \odot \{\mathtt{src}(t_1) \to \mathtt{tgt}(t_n)\} \Big]}_{\neq \overline{0} \text{ when } \pi \text{ witnesses } s \in L(A^L)}
\end{aligned}
$$

(For space reasons, here we use $\mathtt{src}$ for a transition's source and $\mathtt{tgt}$ for its target.) In the final line, $\{\mathtt{src}(t_1) \to \mathtt{tgt}(t_n)\}$ is the relation with a single tuple that captures the fact that π is a path from the source of t_1 to the target of t_n.

Suppose that $s \in L(A^L)$; then $W^L(s) \neq \overline{0}$. By Lemma 3.6, there exists a path π for which $[W_0(\mathtt{src}(t_1)) \otimes W(\pi) \otimes W_f(\mathtt{tgt}(t_1))] \odot \{\mathtt{src}(t_1) \to \mathtt{tgt}(t_n)\}$ is non-$\overline{0}$. By the second tensor-product property and the fact that $\{\mathtt{src}(t_1) \to \mathtt{tgt}(t_n)\} \neq \varnothing = \overline{0}$, we know that $W_0(\mathtt{src}(t_1)) \otimes W(\pi) \otimes W_f(\mathtt{tgt}(t_1)) \neq \overline{0}$. Because $\mathtt{paths}(s)$ in A and A^L are the same, π is also valid in A, and that means that π witnesses that $s \in L(W)$.

Now suppose that $s \notin L(A^L)$ and hence $W^L(s) = \overline{0}$. By a similar argument, we know that $W_0(\mathtt{src}(t_1)) \otimes W(\pi) \otimes W_f(\mathtt{tgt}(t_1)) = \overline{0}$ for every path π in A^L. But that means that the combine of the paths in A is also $\overline{0}$, so s is not accepted by A.

3.5.2 Determinize the state portion of the WFA

If one ignores the data portion of the WFA, this step proceeds exactly as it does for standard finite automata. The question then becomes how do we deal with the weights? It turns out that working in S_T makes this quite simple: we simply take the $\oplus$ of the weights on each of the transitions or states that are put together.

We define A^{SD} ("SD" for state-deterministic) in terms of A^L:

- In agreement with standard FAs, $Q^{SD} = \mathcal{P}(Q^L)$ and, for all $c \in Q^{SD}$ and $\sigma \in \Sigma$, $\delta^{SD}(c, \sigma) = \bigcup_{q \in c} \delta(q, \sigma)$.
- The semiring of A^{SD} is S_T
- Transformers and final weights are the $\oplus$ of the constituents:
  - $W^{SD}(c, \sigma, c') = \bigoplus_{q \in c} \bigoplus_{q' \in c'} W(q, \sigma, q')$
  - $W_f^{SD}(c) = \bigoplus_{q \in c} W_f^L(q)$
- Exactly one cell has nonzero initial weight. Let $C_0 = \{q \in Q^L \mid W_0^L(q) \neq \overline{0}\}$. We define:
  - $W_0^{SD}(C_0) = \bigoplus_{q \in C_0} W_0(q)$
  - $W_0^{SD}(c) = \overline{0}$ for all $c \neq C_0$.

It would also be possible to define *all* initial weights as $W_0^{SD}(c) = \bigoplus_{q \in c} W_0^L(q)$, which is more analogous to the way W^{SD} and W_f^{SD} are defined. However, the following proof is simpler the way it is defined above.

The correctness of the above construction depends on using S_T weights. In particular, we will use the following property:

Property 3.7. Suppose that we are given two transitions t_1 and t_2 in A^L, and that neither $W^L(t_1)$ nor $W^L(t_2)$ is $\overline{0}$. Then:

$$\text{If } \mathtt{target}(t_1) \neq \mathtt{source}(t_2), \text{ then } W^L(t_1) \otimes W^L(t_2) = \overline{0}$$

In other words, if transitions t_1 and t_2 do not form a path, then pretending that they *do* and taking their extend will not give a non-$\overline{0}$ weight when it should not.

Weights in S_T have this property. Suppose $W(t_1) = w_1 \odot \{p \to p'\}$ and $W(t_2) = w_2 \odot \{q \to q'\}$ where $p' \neq q$. Then $W(t_1) \otimes W(t_2) = (w_1 \odot \{p \to p'\}) \otimes (w_2 \odot \{q \to q'\}) = (w_1 \otimes w_2) \odot (\{q \to q'\} \otimes \{p \to p'\}) = (w_1 \otimes w_2) \odot \overline{0} = \overline{0}$.

Now we prove that the determinization step does not affect the language of the WFA. We give a sequence of equalities that prove that the weight of every string s is equal in A^L and A^{SD}. Some of the equalities require further justification and there is new notation; these are covered afterward.

$$
\begin{aligned}
W^{SD}(s) &= \bigoplus_{\pi \in paths^{SD}(s)} \bigotimes_{t \in \pi} W^{SD}(t) && (3.1)\\
&= \bigotimes_{(c,\sigma,c') \in \Pi(s)} W^{SD}(c,\sigma,c') && (3.2)\\
&= \bigotimes_{(c,\sigma,c') \in \Pi(s)} \bigoplus_{q \in c} \bigoplus_{q' \in c'} W^{L}(q,\sigma,q') && (3.3)\\
&= \bigoplus_{{}^{?}\pi \in T(s)} \bigotimes_{t \in {}^{?}\pi} W(t) && (3.4)\\
&= \bigoplus_{\pi \in paths^{L}(s)} \bigotimes_{t \in \pi} W(t) && (3.5)\\
&= W^{L}(s) && (3.6)
\end{aligned}
$$

We now give line-by-line explanations, as well as definitions of $\Pi(s)$, ${}^{?}\pi$, and $T(s)$.

Equation (3.1): This applies the definition of the weight of a string to A^{SD}.

Equation (3.2): The automaton A^{SD} has only one state with a non-$\overline{0}$ initial weight, and from that state (and every other) the transition function δ^{SD} is deterministic. As a result, $paths^{SD}(s)$ is a singleton set; we denote the sole element $\Pi(s)$. (Rather, $\Pi(s)$ is the only path that will contribute a non-$\overline{0}$ weight to the $\bigoplus$.)

Equation (3.3): This simply expands the definition of $W^{SD}(c,\sigma,c')$ given above.

Equation (3.4): This step is only applying the distributive rule, but deserves some explanation. We also use a function $T(s)$ to denote a sequence of transitions (not necessarily a path) defined as follows. For each $1 \leqslant i \leqslant |s|$, let $t_i^{SD} = (c_i, s_i, c_i')$ be the ith transition

of $\Pi(s)$; then let $T_i = \{ (q, s_i, q') \in \delta^L \mid q \in c_i \text{ and } q' \in c_i' \}$. It is possible to view T_i in two ways. The first way is that T_i is all transitions in A^L that were "collapsed" into the ith transition of A^{SD} when reading s. The second way is that T_i is all i^{th} transitions that you can take on any path in A^L matching s.

Finally, let $T(s)$ be the set containing lists of transitions drawn from successive T_i: $T(s) = \{ u_1 u_2 \cdots u_{|s|} \mid u_1 \in T_1, u_2 \in T_2, \cdots, \text{and } u_{|s|} \in T_{|s|} \}$. One way to look at $T(s)$ is each transition list is something that you can obtain by following $\Pi(s)$, and at each step i choosing a transition from A^L that was collapsed into t_i and then forgetting which state in the cell you were in.

We use ${}^?\pi$ in Equation (3.4) to mark the fact that elements of $T(s)$ are not necessarily paths, but just lists of transitions.

Equation (3.5): Each element ${}^?\pi$ of $T(s)$ corresponds to either a legitimate path in A^L, and is thus an element of $\mathtt{paths}^L(s)$, or to a non-path transition list. For ${}^?\pi$ that are not paths, by Property 3.7 we know that $\bigotimes_{t \in {}^?\pi} W(t) = \overline{0}$ and thus dropping it from the $\bigoplus$ will not change the result. For ${}^?\pi$ than are paths, ${}^?\pi \in \mathtt{paths}^L(s)$ because all transition labels match and it starts from an initial state. Finally, every $\pi \in \mathtt{paths}^L(s)$ appears in $T(s)$ because that is what determinization does.

Equation (3.6): This applies the definition of the weight of a string to A^L.

3.6 Language containment

To determine language containment of standard finite automata A and B, the traditional algorithm determinizes and complements B, intersects the result with A, and checks the intersection against emptiness: $A \subseteq B$ iff $A \cap \neg B = \varnothing$. (De Wulf et al. provide an alternative inclusion procedure that uses antichains [25]; we discuss antichains later.)

The inclusion testing process for XFAs works along the same broad arc, but in some ways it is rather different. The root cause of the difference is that we logically group the powerset construction of the data domain with the test itself, rather than saying "do the subset construction followed by a search."

We start by defining powerset semirings, as well a lifting of an automaton's weight domain to the powerset semiring (Section 3.6.1). Because universality testing raises in a simpler way most of the same complications and solving techniques as inclusion testing,

we first describe a basic universality test for XFAs (Section 3.6.2).[5] Following that, we move back to inclusion testing, defining our cross product construction for WFAs (Section 3.6.3), a basic version of the inclusion test (Section 3.6.4), and finally describe how to use antichains to improve the efficiency of the operations in practice (Section 3.6.5).

3.6.1 The powerset semiring [background and new]

Just as the subset construction tracks sets of states, to obtain a fully-deterministic WFA we must track sets of weights; why is explained in the context of XFAs in the introduction to Section 3.5. This means that the data set must now be $\mathcal{P}(Q \times D)$, where D is the original (not state-deterministic) data set.

We do this by using the powerset semiring, $\mathcal{P}(S_T)$:

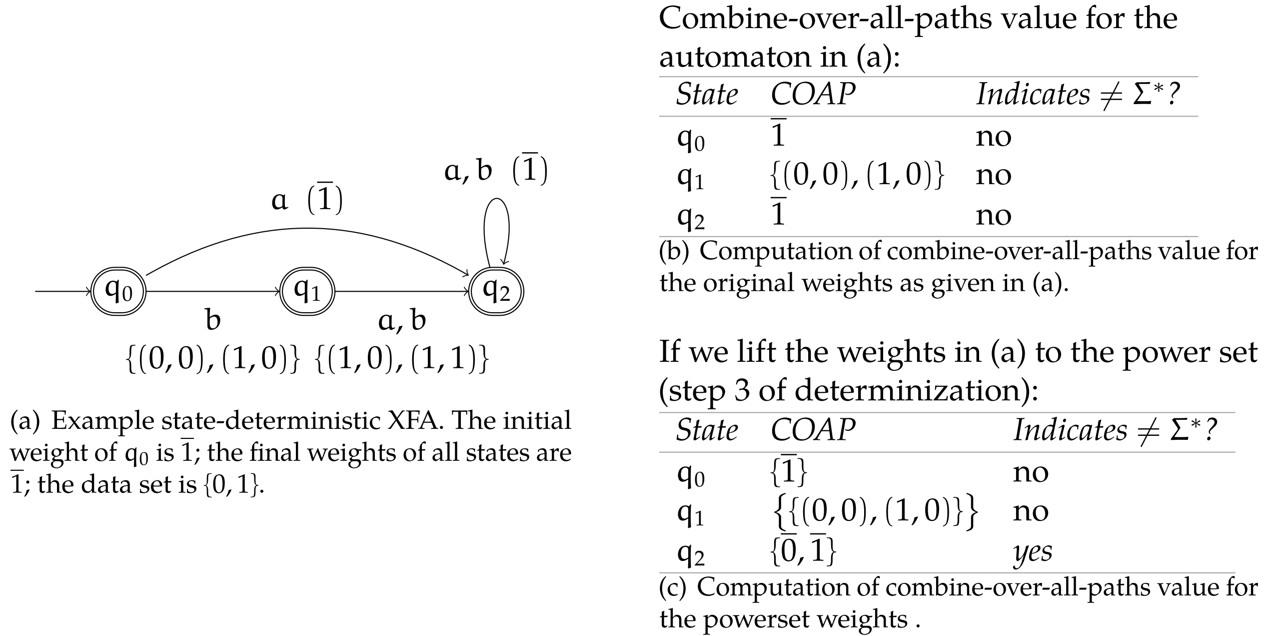
- The set is $\mathcal{P}(S_T)$
- $\overline{0} = \varnothing$
- $\overline{1}_{\mathcal{P}(S_T)} = \{\overline{1}_{S_T}\}$
- $\oplus$ is set union, $\cup$
- $\otimes$ is pairwise extend: $X \otimes_{\mathcal{P}(S_T)} Y = \{x \otimes_{S_T} y \mid x \in X \text{ and } y \in Y\}$

where we subscript components of the semirings for clarity.

The fully-deterministic WFA A^D is defined as follows:

- The state portion is the same as W^{SD}; $Q^D = Q^{SD}$ and $\delta^D = \delta^{SD}$
- The semiring is $\mathcal{P}(S_T)$
- All weights are singleton sets of the weights in W^{SD}:
  - $W^D(c, \sigma, c') = \{W^{SD}(c, \sigma, c')\}$
  - $W_0^D(c) = \{W_0^{SD}(c)\}$
  - $W_f^D(c) = \{W_f^{SD}(c)\}$

[5] Recall that an automaton A is universal iff $L(A) = \Sigma^*$. Universality and inclusion are related because answering both requires something that can basically turn into the subset construction; both problems are PSPACE complete.



(a) Example state-deterministic XFA. The initial weight of q_0 is $\overline{1}$; the final weights of all states are $\overline{1}$; the data set is $\{0, 1\}$.

Combine-over-all-paths value for the automaton in (a):

State	*COAP*	*Indicates* $\neq \Sigma^*$*?*
q_0	$\overline{1}$	no
q_1	$\{(0,0),(1,0)\}$	no
q_2	$\overline{1}$	no

(b) Computation of combine-over-all-paths value for the original weights as given in (a).

If we lift the weights in (a) to the power set (step 3 of determinization):

State	*COAP*	*Indicates* $\neq \Sigma^*$*?*
q_0	$\{\overline{1}\}$	no
q_1	$\big\{\{(0,0),(1,0)\}\big\}$	no
q_2	$\{\overline{0},\overline{1}\}$	*yes*

(c) Computation of combine-over-all-paths value for the powerset weights .

Figure 3.7: Illustration of why the third step of determinization, lifting transformers to the power set, is necessary. Doing so avoids combining the weights of a and ba

It may appear at first glance that this step does not actually do anything, because $W^D(s) = \{W^{SD}(s)\}$ for every string s. However, if we want to start looking at multiple strings at once, the extra powerset is necessary to avoid combining information from different strings, because $W^D(s_1) \oplus W^D(s_2) = \{W^{SD}(s_1), W^{SD}(s_2)\} \neq \{W^{SD}(s_1) \oplus W^{SD}(s_2)\}$ in all cases where s_1 and s_2 have different weights.

Fig. 3.7 gives an example automaton that is not universal, and shows that if we perform the same process on W^{SD} instead of W^D, the result is incorrect.

3.6.2 WFA universality [new]

Suppose we are given a state-deterministic XFA A. Let the single state with a non-$\overline{0}$ initial weight be q_0. We will use W^{SD} for weights in A to emphasize the state-deterministic aspect. We want to determine whether there exists a string s such that $W^{SD}(s) = \overline{0}$, in which case A is not universal. A conceptual, non-effective way of determining the existence of s is to materialize the entire mapping W^{SD} of strings to weight, and then look through W^{SD} for an entry with weight $\overline{0}$. However, it is not necessary to enumerate the entire mapping: if

it was possible to list every weight that could arise (i.e., the projection of W^{SD} onto the second element of each pair), then looking at that list would be sufficient: if $\overline{0}$ was present, that would mean that *some* string was not accepted (although there would not be enough information to say what string). When the semiring is of finite size, it is actually possible to compute the set of all possible weights that can arise, and it can be effectively (if potentially slowly) generated; this section describes how. (Later we talk about how to use antichains to not have to list all possible weights.)

Note that we do not quite need to compute the full weight of each string, including the initial and final weights. If we know that there is a path π from the initial state q_0 (i.e., the only state with a non-$\overline{0}$ initial weight) to q, then we know that there is a *string* with weight $W_0^{SD}(q_0) \otimes W^{SD}(\pi) \otimes W_f^{SD}(q)$. It is thus sufficient to compute the weight of all paths from q_0 to each node, without regard to the initial and final weights.

With that introduction, our goal is to compute, for each state q, the set of weights that can reach that state. That is, if we define $\mathrm{paths}(q_0, q)$ to be the set of paths from q_0 to q regardless of their length or what string labels them, we want to compute the following value:

$$\{W^{SD}(\pi) \mid \pi \in \mathrm{paths}(q_0, q)\}$$

We can do this by first lifting A to A^D to use the powerset semiring as described in Section 3.6.1 and then computing the *combine-over-all-paths* (COAP) value for each state, which is defined as follows:

$$\mathrm{COAP}(q) = \bigoplus_{\pi \in \mathrm{paths}(q_0, q)} W^D(\pi) \tag{3.7}$$

These two goals are equivalent. (Note that paths is the same in A and A^D.) That is:

$$\mathrm{COAP}(q) = \{W^{SD}(\pi) \mid \pi \in \mathrm{paths}(q_0, q)\} \tag{3.8}$$

We will discuss how to compute $\mathrm{COAP}(q)$ and prove the equivalence claimed by Equation (3.8) at the end of the section. Define $\mathrm{COAP}'(q) = \{W_0^{SD}(q_0) \otimes w \otimes W_f^{SD}(q) \mid w \in \mathrm{COAP}(q)\}$.

Once the value $\mathrm{COAP}(q)$ is computed for each q, determining whether the language is universal is very simple: the language is not universal iff there is a q such that $\overline{0} \in \mathrm{COAP}'(q)$. First we prove the forwards direction: if the language is not universal, then there is a q such

that $\overline{0} \in \mathsf{COAP}'(q)$. Suppose s is a string for which $s \notin L(A)$. Let $q_s = \delta(q_0, s)$. Because s is not in A's language, we know that $\overline{0} = W^{SD}(s) = W_0^{SD}(q_0) \otimes W^{SD}(\Pi(s)) \otimes W_f^{SD}(q_s)$, where $\Pi(s)$ is the only path starting in q_0 matching s. By Equation (3.8), $W^{SD}(\pi) \in \mathsf{COAP}(q_s)$, and thus $\overline{0} \in \mathsf{COAP}'(q_s)$. The steps in this proof can be reversed to get the backwards direction.

Remark 3.8. There is a curious fact about this algorithm, which is that $\mathsf{COAP}(q) = \varnothing$ is not a counterexample to universality, whereas $\mathsf{COAP}(q) = \{\overline{0}\}$ is. This may be surprising to readers who have a mental model of a weight $\overline{0}$ being equivalent to the absence of a path or transition. The explanation for this apparent contradiction can be explained by way of analogy to standard FA. Suppose that we are given an FA that is deterministic in that there is never a choice between two transitions, but that may be incomplete in the sense that $\delta(q, \sigma)$ is undefined (or $\varnothing$) for certain inputs. If every state that is reachable from the initial state (via δ) is accepting, this does *not* necessarily mean that the machine is universal: if there is some string s for which $\delta(q_0, s)$ is undefined, then that string will be rejected and the machine is not universal. Conversely, just because some state q is not accepting does not necessarily mean that the automaton is not universal: q may not be reachable from the initial state. Only if every state is reachable and the automaton is complete (and deterministic) does "the automaton is universal" correspond to "every state accepts." In the case of WFAs, $\mathsf{COAP}(q) = \varnothing$ corresponds to the unreachable case, and tracking $\overline{0} \in \mathsf{COAP}(q)$ corresponds to making the WFA "complete." If we took care to only look at the reachable portion of the XFA and checked for completeness via other means, we could use a different inclusion test.

Computing $\mathsf{COAP}(q)$ can be done using standard techniques, which nearly-exactly matches those for computing ε closure, except that we include all transitions instead of just ε transitions. We perform the computation by way of translation to a WPDS followed by a post* query, as shown in Fig. 3.3(a) except that $\Delta = \{\langle p, q\rangle \hookrightarrow \langle p, q'\rangle \mid \exists\sigma : q' \in \delta(q, \sigma)\}$.

Finally, we prove the equivalence stated in Equation (3.8), that $\mathsf{COAP}(q) = \{W^{SD}(\pi) \mid \pi \in \mathsf{paths}(q_0, q)\}$:

$$\begin{aligned}
\mathsf{COAP}(q) &= \bigoplus_{\pi \in \mathsf{paths}(q_0, q)} W^D(\pi) && \text{By Equation (3.7)} \\
&= \bigcup_{\pi \in \mathsf{paths}(q_0, q)} W^D(\pi) && \text{Def. of } \bigoplus \text{ in powerset semiring} \\
&= \bigcup_{\pi \in \mathsf{paths}(q_0, q)} \{W^{SD}(\pi)\} && \text{Def. of } W^D \text{ (Section 3.6.1)} \\
&= \{W^{SD}(\pi) \mid \pi \in \mathsf{paths}(q_0, q)\}
\end{aligned}$$

3.6.3 WFA cross product [adaption/new]

For standard FAs, taking the cross product of two automata means that we take the Cartesian product of the two machines' sets of states, initial states, and accepting states, and that we take the Kronecker product of the transition relations.

For WFAs, we do the same. How we treat the weights can be viewed three ways. From the point of view of XFAs, we take the Kronecker product of the transformers as well. More generally, we can use a paired semiring. Given S_A and S_B, we define the semiring $\langle S_A, S_B \rangle$ as:

- The set is the Cartesian product $S_A \times S_B$
- $\overline{0}_{\langle\rangle} = \langle \overline{0}_A, \overline{0}_B \rangle$
- $\overline{1}_{\langle\rangle} = \langle \overline{1}_A, \overline{1}_B \rangle$
- Extend and combine are pointwise: $\langle a, b \rangle \otimes_{\langle\rangle} \langle a', b' \rangle = \langle a \otimes_A a', b \otimes_B b' \rangle$ and similarly for $\oplus$

It would also be possible to define the cross product in terms of another tensor product, but it would complicate the description of the inclusion test and we have not found that it leads to significant additional insight.

For WFAs, we define the cross product of $A = (Q^A, \Sigma, \delta^A, W^L, W^L_0, W^L_f)$ and $B = (Q^B, \Sigma, \delta^B, W^B, W^B_0, W^B_f)$ as $A \times B = (Q^\times, \Sigma, \delta^\times, W^\times, W^\times_0, W^\times_f)$ where:

- The semiring of $A \times B$ is the paired semiring $\langle S_A, S_B \rangle$

- $Q^{\times} = Q^{A} \times Q^{B}$
- $\delta^{\times} = \delta^{A} \times_{\text{Kronecker}} \delta^{B}$ (i.e., $\delta^{\times}((q_a, q_b), \sigma) = \delta^{A}(q_a, \sigma) \times \delta^{B}(q_b, \sigma)$)
- $W^{\times}((q_a, q_b), \sigma, (q'_a, q'_b)) = \langle W(q_a, \sigma, q'_a), W(q_b, \sigma, q'_b)\rangle$
- $W_0^{\times}((q_a, q_b)) = \langle W_0(q_a), W_0(q_b)\rangle$
- $W_f^{\times}((q_a, q_b)) = \langle W_f(q_a), W_f(q_b)\rangle$

3.6.4 Basic inclusion test [new]

The basic inclusion test proceeds much the same as for universality checking. Given two WFAs A and B, to determine whether $L(A) \subseteq L(B)$, we perform the following steps:

1. Compute A^{SD} and B^{SD} as described in Sections 3.5.1 and 3.5.2 (only B^{SD} should really be necessary, but we perform both)
2. Construct $A^{SD} \times B^{SD}$ as described in Section 3.6.3
3. Lift the to the powerset domain to get $(A^{SD} \times B^{SD})^{D}$, as described in Section 3.6.1
4. Compute $COAP'(q)$ for each state q of $(A^{SD} \times B^{SD})^{D}$
5. Look for a state q for which $COAP'(q)$ contains an element $\langle a, b\rangle$ where $a \neq \bar{0}$ and $b = \bar{0}$.

The proof that this procedure is correct proceeds much as the proof for universality did, except that we have to look at each component of the pairs. We need $a \neq \bar{0}$ to hold so we make sure that the strings we consider are a member of $L(A)$ — finding a string that is not in $L(B)$ is not a counterexample to inclusion if it is also not in $L(A)$.

3.6.5 Speeding up operations with antichains [as indicated]

We can speed up the inclusion test — often significantly — by using an antichain algorithm [25]. In this section, we describe how antichain algorithms work on standard finite automata and then how we can use them for XFAs.

```
1  isUniversal(Q, Σ, δ, q0, F)
2    visited = {};
3    bfsQueue.enqueue(q0);
4    while bfsQueue is not empty do
5      currentState = bfsQueue.dequeue();
6      for σ ∈ Σ do
7        next = getSuccessors(currentState, δ, σ);
         // next is a set of states --- i.e., a cell
8        if next ∩ F = ∅ then
           // A nonaccepting state is reachable!
9          return false;
10       if next ∉ visited then
           // We found a new state!
11         visited.insert(next);
12         bfsQueue.enqueue(next);
   // We have fully explored the state space reachable from q0, and
      nothing rejects
13 return true;
```

Listing 3.8: The non-antichains, early-cutoff universality algorithm for FAs.

Antichains for FA universality checking [background]

Consider universality checking on a standard, nondeterministic FA A. The simplest algorithm for universality checking would determinize A using the subset construction, then do a forward search over the resulting transition graph from the initial cell.[6] If a non-accepting cell is reachable from the initial cell, then A is not universal; otherwise it is universal. A better algorithm can be obtained by weaving together the subset construction with the search, stopping the subset construction and returning "not universal" if a non-accepting cell is ever generated. For purposes of this discussion, consider this algorithm to primarily track a *set of cells* that have been visited — in other words, a set of sets of states in the nondeterministic automaton. Listing 3.8 shows code for such an algorithm, which we will call the "early-cutoff" algorithm.

De Wulf et al.'s antichain algorithm for FA universality checking [25] adapts the interwoven algorithm to get significant gains in practice.[7] It still tracks a set of visited cells, but

[6] Recall that a "cell" is just the term for a state generated by the subset construction.

[7] All three algorithms have the same worst-case complexity, which arises when the constructions generate cells in a bad order.

maintains that this visited set is an *antichain*. An antichain of states is a set of cells such that no two cells in the antichain are comparable via $\subseteq$.

When a new cell c is discovered, the non-antichain algorithm would always add c to its visited set. The antichain algorithm will perform one of three actions. The algorithm will *ignore* c if there is an existing c' in its visited set such that $c' \supset c$. The algorithm will *replace* with c one or more existing cells $c'_1, c'_2, \cdots$ in the visited set, for which $c'_i \subsetneq c$. The algorithm will simply *add* c to the visited set if it is incomparable to all existing elements.

The effect of the antichain universality algorithm is that it needs to explore less of the determinized state space. Figure 3.9 shows an example automaton and its determinization, indicating the portion that is not explored because of the use of antichains.

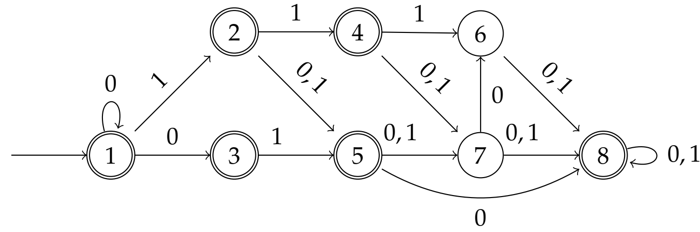
The reason that the antichain universality algorithm is correct can be expressed as follows. Suppose we have a cell c that is reachable from the initial cell. If there is a string s from which all paths reject, then the automaton is not universal. But if all paths from c reject, then so do all paths from any cell c' that is a subset of c. (The opposite is not true of course.) What this means is that every time that we can use c to help prove non-universality, we could also have used c', provided that c' is reachable from the initial cell. Thus keeping track of c is not helpful for purposes of universality checking, so we can drop it.

Antichains for WFA universality testing [new]

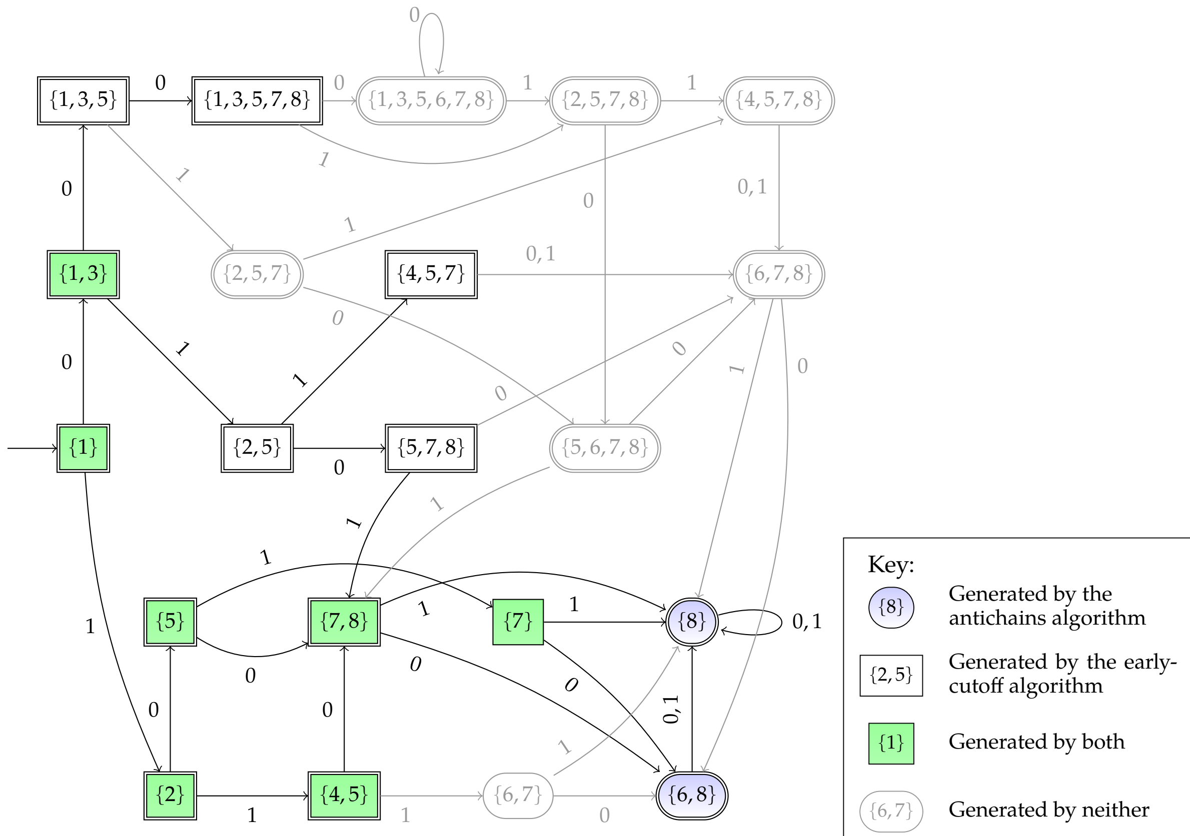
We can use a similar algorithm to improve WFA universality and inclusion testing. First we broaden the definition of antichains to semirings, then describe how we can use antichains to help with universality checking. In the next subsubsection, we describe how antichains can help with inclusion checking.

Suppose we are given a semiring S. An *antichain* of S is a set of elements $\{s_1, s_2, \cdots, s_n\}$ such that no two elements are comparable by $\sqsubseteq$; i.e., for all s_i and s_j, we have $s_i \not\sqsubseteq s_j$ and $s_j \not\sqsubseteq s_i$. For an arbitrary subset $T \subseteq S$, let $\lceil T \rceil$ be the antichain that contains all maximal elements of T; that is, $\lceil T \rceil = \{s \in T \mid \forall s' \neq s \in T : s \not\sqsubseteq s'\}$. Dually, let $\lfloor T \rfloor$ be the antichain that contains all minimal elements of T, i.e., $\{s \in T \mid \forall s' \neq s \in T : s \not\sqsupseteq s'\}$.

There are two alternatives to the powerset semiring that we can define: one is the semiring made up of antichains of maximal elements, and the other is the semiring made up of antichains of minimal elements. Antichains of maximal elements are used if we compute $\mathsf{COAP}(q)$ in a backwards direction (pre*), and antichains of minimal elements



(a) An example NFA designed to illustrate the benefits of the antichain algorithm for universality checking, from [26].



(b) The result of applying the subset construction to the automaton in (a). Note that the set of states generated by both algorithms is the same as the set of states generated by the algorithm with both optimizations.

Figure 3.9: Illustration of the benefit of antichains in universality checking as well as cutting off the search early. As can be seen, both optimizations can improve the search by quite a bit. Furthermore, the benefits are cumulative: changing from the early-cutoff, non-antichains algorithm to the early-cutoff, antichains algorithm still reduces the size of the explored state space from 13 states to 7 states. In both cases, the indicated states are what arise when exploring the state space in a breadth-first manner, following 0 edges before 1 edges.

are if we compute $\mathsf{COAP}(q)$ in a forwards direction (post*). Here we will assume that we are working in a forwards direction, but the two cases are duals of each other.[8] In each case, the construction is the same as the powerset semiring except that when taking $\otimes$ or $\oplus$, we add a $\lfloor\cdot\rfloor$ or $\lceil\cdot\rceil$ as appropriate to the result.

To make use of antichains, we simply replace the powerset semiring with the appropriate antichain semiring when performing the construction described in Section 3.6.1.

Suppose we have automaton A that uses the powerset semiring and automaton B that uses the semiring of minimal antichains. We now prove that A is universal iff B is universal. Let $\mathsf{COAP}^A(q)$ and $\mathsf{COAP}^B(q)$ be the combine-over-all-paths solutions in A and B respectively.

Lemma 3.9. For each state q:

- $\lfloor \mathsf{COAP}^A(q) \rfloor = \mathsf{COAP}^B(q)$.
- $\mathsf{COAP}^A(q) \supseteq \mathsf{COAP}^B(q)$.

Proof. The second fact follows immediately from the first.

The first fact follows the proof at the end of Section 3.6.2, with minor changes to insert $\lfloor\cdot\rfloor$ operators:

$$\begin{aligned}
\mathsf{COAP}(q) &= \bigoplus_{\pi \in paths(q_0,q)} W^D(\pi) \\
&= \left\lfloor \bigcup_{\pi \in paths(q_0,q)} W^D(\pi) \right\rfloor \\
&= \left\lfloor \bigcup_{\pi \in paths(q_0,q)} \{W^{SD}(\pi)\} \right\rfloor \\
&= \lfloor \{W^{SD}(\pi) \mid \pi \in paths(q_0, q)\}
\end{aligned}$$

[8]It is also possible to describe FA universality checking via a backwards search from the accepting states, in which case we would keep maximal cells instead of minimal cells. De Wulf et al.'s original explanation actually proceeds in this direction.)

The second equality is justified by the fact that generating the entire set $\mathtt{bigcup}_\pi W^D(\pi)$ and taking the minimum is the same as computing the minimum "along the way"; that is:

$$\lfloor W^D(\pi_1) \cup W^D(\pi_2) \cup W^D(\pi_3) \cup \cdots \rfloor = \lfloor \cdots \lfloor \lfloor W^D(\pi_1) \cup W^D(\pi_2) \rfloor \cup W^D(\pi_3) \rfloor \cup \cdots \rfloor.$$

This property falls out naturally from the definition of $\lfloor \cdot \rfloor$. ☐

> **Theorem 3.10.** If WFA A is constructed using the powerset semiring and B is constructed using the semiring of minimal antichains, then A is universal iff B is universal.

Proof. From Theorem 3.10, we know immediately that if B is not universal, neither is A. This is because if A is not universal, there is a state q for which $\bar{0} \in \mathsf{COAP}^{A'}(q)$. Because $\mathsf{COAP}^B(q) \subseteq \mathsf{COAP}^A(q)$ and the initial and final weights are the same in the two automata, $\bar{0} \in \mathsf{COAP}^B(q)$ as well. Thus B is not universal.

For the other direction (if A is not universal, neither is B), suppose we know q and π (where $\pi \in \mathtt{paths}(q_0, q)$) such that $W_0^A(q_0) \otimes W^A(\pi) \otimes W_f^A(q) = \bar{0}$. There must be some $w \in \mathsf{COAP}^B(q)$ for which $w \sqsubseteq W^A(\pi)$. By the definition of $\sqsubseteq$, $W(\pi) + w = W(\pi)$, which means that

$$\begin{aligned} \bar{0} &= W_0^A(q_0) \otimes W^A(\pi) \otimes W_f^A(q) \\ &= W_0^A(q_0) \otimes (W^A(\pi) \oplus w) \otimes W_f^A(q) \\ &= \left(W_0^A(q_0) \otimes W^A(\pi) \otimes W_f^A(q)\right) \oplus \left(W_0^A(q_0) \otimes w \otimes W_f^A(q)\right) \end{aligned}$$

which, by Lemma 3.6, can only be true if $W_0^A(q_0) \otimes w \otimes W_f^A(q) = \bar{0}$. That means that B is not universal either. ☐

Antichains for WFA inclusion testing [new]

In this section, we describe how we use the idea of antichains to improve inclusion testing. We use a different ordering $a \preccurlyeq b$ than $a \oplus b = b$. Recall that semiring elements of the cross product are pairs; we define $\langle a, b \rangle \preccurlyeq \langle a', b' \rangle$ iff $a \sqsupseteq a'$ and $b \sqsubseteq b'$ in their respective domains. Keeping minimal elements according to $\preccurlyeq$ corresponds to keeping maximal elements from the first automaton and minimal elements from the second.

3.6.6 Complexity of XFA universality and inclusion testing [new]

What we would like to demonstrate is that the (potentially exponentially-expensive) process described earlier in this section is necessary to perform universality and inclusion testing.

Suppose that we are given a state-deterministic XFA A, and wish to determine whether it is universal. Lifting A to the powerset semiring is linear in the size of A because it does not really do anything, but the weight propagation process takes time $\mathcal{O}(h|Q|)$ where h is the height of the semiring. If A's semiring is S, then A^D's semiring is $\mathcal{P}(S)$,[9] whose height h is $|S|$. If A's data set is D, then $|S| = 2^{|D|\cdot|D|}$. This means that universality checking a state-deterministic XFA takes $\mathcal{O}(|Q|2^{|D|\cdot|D|})$ time.

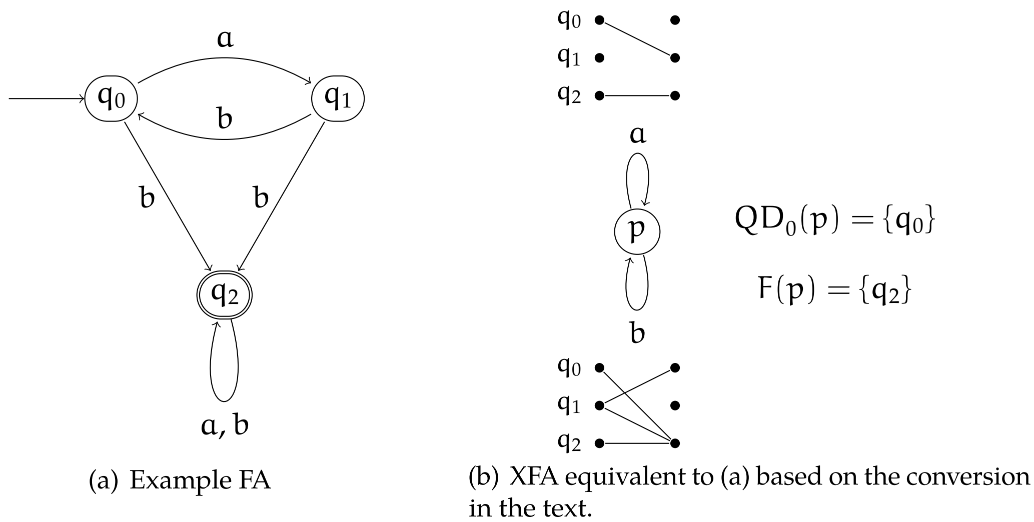
Now, suppose that we are given a standard FA $B = (Q_B, \Sigma, \delta_B, Q_{0,B}, F_B)$. In linear time in the size of B we can create a single-state, state-deterministic XFA C for which $L(C) = L(B)$. Let $\delta_B|_\sigma = \{(q, q') \mid q' \in \delta_B(q, \sigma)\}$, and create the following automaton:

- $Q = \{p\}$, $\delta(p, \sigma) = \{p\}$ for all σ
- $D = Q_B$
- $U(p, \sigma, p) = \delta_B|_\sigma$
- $QD_0(p) = Q_{0,B}$
- $F(p) = F_B$

In other words, we have taken B's state transitions and moved them entirely into C's data transformers. Fig. 3.10 gives an example of this transformation.

Determining whether B is universal requires $\mathcal{O}(2^{|Q_B|})$ in the worst case, while solving via an XFA is $\mathcal{O}(2^{|Q_B|\cdot|Q_B|})$. So the XFA route has a somewhat higher complexity, but it is only a small polynomial increase in an exponential time algorithm.

[9] I am ignoring the S-to-S_T transformation done in the first step of determinization, as it is not required when starting from a state-deterministic XFA.



(a) Example FA

(b) XFA equivalent to (a) based on the conversion in the text.

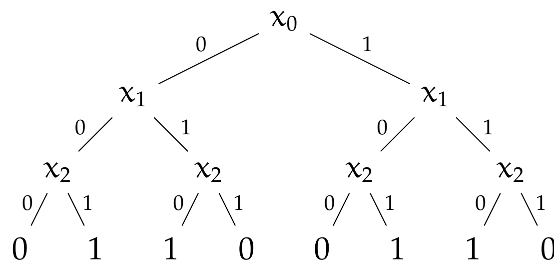
Figure 3.10: Transforming an FA into an XFA.

3.7 Representing relations as BDDs [background]

This section describes the principle way we represent relations, using *binary decision diagrams* (BDDs).[10] BDDs are a representation of Boolean functions that are often compact in practice [16]. We give an introduction to BDDs with an informal definition, then proceed to describe how it is possible to represent *non*-binary functions and relations.

3.7.1 A brief introduction to BDDs

One way to define BDDs is as follows. Consider a Boolean function $f(b_1, b_2, \cdots, b_n)$, and a decision tree that specifies the result of the function for each input. For example, this is a decision tree for the function $f(x_0, x_1, x_2) = x_0 \text{ xor } x_1 \text{ xor } x_2$:



[10]Technically, we are talking about what are called "reduced, ordered" BDDs, or ROBDDs, but we refer to them as just BDDs here.

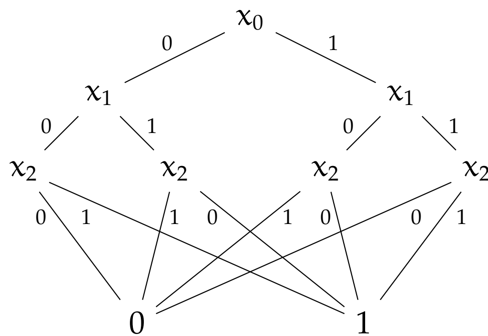
A BDD for a function f can be thought of as a representation of the decision tree, compressed through the following rules:

1. There are always exactly two leaf nodes, one 0 and one 1. (An exception is that the constant functions $f(\vec{b}) = 0$ and $f(\vec{b}) = 1$ will be represented by a one-node tree.)

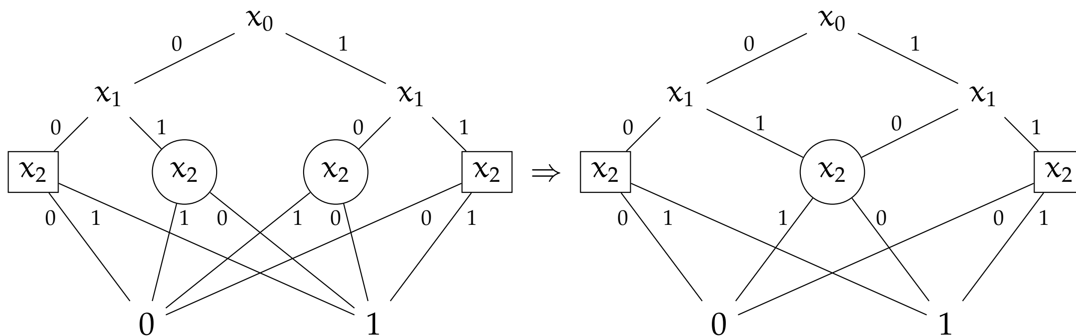
2. If two nodes n_1 and n_2 are at the same level (i.e., are labeled with the same variable), and they have the same left child and the same right child (and hence behave the same), then n_2 is collapsed into n_1. That is, n_2 is removed and any incoming edges are redirected to n_1.

3. If both children of a node n point to the same node, then n is removed and any incoming edges are redirected to its child.

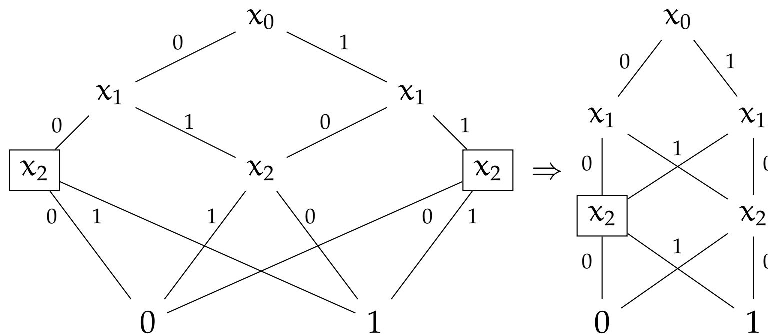
For example, we can apply these rules to the function specified above (note that the left child in the picture is not necessarily the 0 edge any more in these diagrams). First we apply the first rule:



Then we note that there are two "kinds" of x_2 nodes, marked with a circle and a square. We collapse the circle nodes:



Then the square nodes:



There are no more nodes that meet the criteria of the second compression step, and there are no nodes at all that meet the criteria of the third, which means that the above is the canonical BDD for the given function. The nodes along the left path represent the case where an even number of 1s have been seen so far, and the nodes along the right path represent the case where an odd number of 1s have been seen. We could easily enough extend this idea to the *xor* of any number of variables and the size of the BDD will only grow linearly, instead of the exponential growth seen by a complete representation of the decision tree itself.[11]

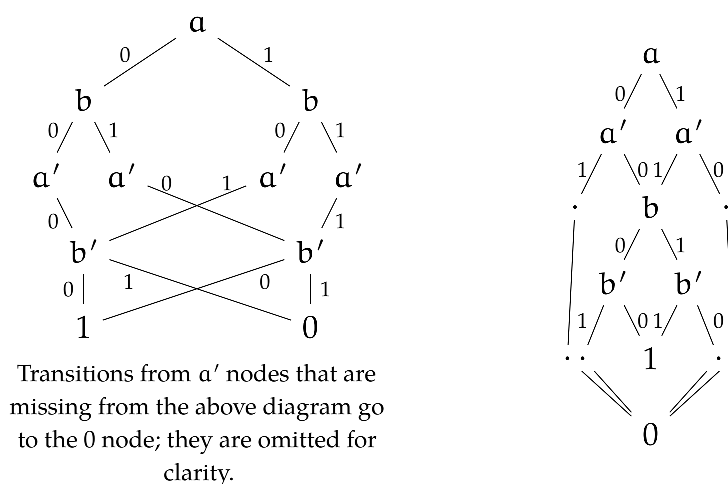
There is no guarantee, of course, that BDDs necessarily remain small; in the worst case, the compression steps may not do anything and it may be necessary to track the entire decision tree unexpanded. However, many groups have had a lot of success using BDDs in practice.

There are algorithms for manipulating BDDs that apply logical operations to their functions. For instance, given BDDs for $f_1(\vec{b})$ and $f_2(\vec{b})$, it is possible to construct a BDD for $f_3(\vec{b}) = f_1(\vec{b}) \wedge f_2(\vec{b})$ in time that is at most quadratic in the sizes of the two input BDDs. Quantification is also possible. Given a BDD for function $f_1(b_1, b_2, \cdots, b_n)$, it is possible to create a BDD for $f_3(b_1, b_2, \cdots, b_k) = \exists b_{k+1}, b_{k+2}, \cdots b_n : f_1(b_1, b_2, \cdots, b_n)$ and similarly for $\forall$. (The quantified variables need not be at the end.) Quantification, unfortunately, is potentially exponentially expensive in the number of variables being quantified out ($n - k$ above).

[11]This suggests an alternative interpretation of BDDs, which is that BDDs act like automata that read the arguments to the function as a fixed-length string but that can potentially skip over variables. The 1 node is the sole accepting state, and the first two compression rules correspond exactly to automaton minimization.

One very useful aspect of BDDs is that the representation of a function is uniquely determined (up to the variable order, discussed momentarily). Actual implementations of BDDs go one step further and make sure that the exact same object in memory is produced when the BDDs are equal. This invariant takes relatively little effort to maintain, improves memory usage in practice (because there are not redundant objects in memory), and it also makes checking equality of two functions represented as BDDs extremely fast.

Finally, it is important to address the *variable order* of a BDD, which is the order in which variables are consulted as the BDD is traversed from root to leaf. Because each node of the BDD is labeled with the variable to be consulted, it is not necessary for the variable order to agree with the order of the function's arguments, and the variable order can be freely changed without disrupting the meaning of the BDD. For example, the following are two BDDs for the function $f(a, b, a', b') = (a = a') \wedge (b = b')$ using different variable orders:



Transitions from a' nodes that are missing from the above diagram go to the 0 node; they are omitted for clarity.

With only two pairs of variables that are compared ($a = a'$ and $b = b'$) the trend may not be obvious, but if the number of pairs is increased the left order is far less efficient than the right order. Suppose that $f(a_1, \cdots, a_k, a'_1, \cdots, a'_k) = (a_1 = a'_1) \wedge \cdots \wedge (a_k = a'_k)$. Using the variable order $a_1, a_2, \cdots, a_k, a'_1, \cdots, a'_k$ (as on the left) will cause the BDD to have a size exponential in k, while using the variable order $a_1, a'_1, a_2, a'_2, \cdots, a_k, a'_k$ (as on the right) will cause the BDD to have a size that is only linear in k.

As this example illustrates, using a good variable order is very important to using BDDs efficiently. Unfortunately, *finding* the best variable order to use is an NP-complete problem [12]. For our application, we use domain knowledge to select a variable order that is likely to be reasonable; Section 6.2.3 describes this order.

3.7.2 Representing non-Boolean functions and relations with a BDD

As described above, BDDs are representations of *Boolean* functions. To represent non-Boolean functions and relations, there are two steps: first, we encode the domain (and co-domain, if any) using Boolean variables, and second, we create a BDD for the characteristic function of the object of interest.

For instance, suppose we have an XFA where the data set is $D = \{0, 1, \cdots, 7\}$. We need three Boolean variables b_1, b_2, b_3 to represent the value from D that is the first element of each tuple in the relation, which we will do by treating $b_1 b_2 b_3$ as the binary representation, and another three variables b_1', b_2', b_3' to represent the second element in each tuple. The simple relation $\{(0, 5)\}$ is then represented with a BDD for the function:

$$f(b_1, b_2, b_3, b_1', b_2', b_3') = (b_1 = 0 \wedge b_2 = 0 \wedge b_3 = 0 \wedge b_1' = 1 \wedge b_2' = 0 \wedge b_3' = 1)$$

As mentioned previously, the application of XFAs discussed in Chapter 6 will use data values with some structure. Each value will be an assignment to a set of logical variables $\{x_1, x_2, \cdots, x_n\}$, where each variable takes on a numeric value, and thus each data value is a tuple like $(0, 1, 3, 2)$. We assign a separate group of Boolean variables to each logical variables, and then write the characteristic function as $f(\vec{x_1}, \vec{x_2}, \cdots, \vec{x_n}, \vec{x_1}', \vec{x_2}', \cdots, \vec{x_n}')$.

We will switch between using $f(b_1, \cdots, b_n)$, $f(\vec{b})$, $f(\vec{b}, \vec{b}')$ (where the b variables can cover "logical variables"), and $f(\vec{x_1}, \cdots, \vec{x_n}, \vec{x_1}', \cdots, \vec{x_n}')$ depending on how much detail we need.

The empty relation $\overline{0} = \varnothing$ is just the BDD with a single 0 node. The identity relation $\overline{1}$ is the BDD for $f(\vec{b}, \vec{b}') = \bigwedge_{i=1}^{i \leqslant |\vec{b}|}(b_i = b_i')$, which is small if the b and b' variables are interleaved in the variable ordering. The $\oplus$ of two relations represented by BDDs B_1 and B_2 just corresponds to taking the disjunction: $(B_1 \oplus B_2)(\vec{b}) = (B_1(\vec{b})) \vee (B_2(\vec{b}))$. The composition of B_1 and B_2 is expressed as follows:

$$(B_1 \otimes B_2)(\vec{b}, \vec{b}') = \exists \vec{b}'' : (B_1(\vec{b}, \vec{b}'') \wedge B_2(\vec{b}'', \vec{b}'))$$

The above can be computed on BDDs in the following manner:

- The $\vec{b}'$ variables in B_1 are renamed to $\vec{b}''$
- The $\vec{b}$ variables in B_2 are renamed to $\vec{b}''$

- The conjunction of these two BDDs is done [16, §4.3]
- The $\vec{b}''$ variables are existentially quantified away

In actual implementations, the final two steps are typically woven together to obtain a somewhat more efficient algorithm [55, "The *AndExists* algorithm", §2.3].

3.8 XFA implementation in WALi [new]

We have implemented the operations described in this chapter in an add-on library in WALi, the weighted automaton library [44]. WALi already provided an implementation of WFAs, but it was essentially only geared towards using WFAs as a part of program analysis using WPDSs. We extended it with the generic WFA algorithms described in this chapter (epsilon closure, determinization, and language inclusion) as well as provided an XFA wrapper class that knows that its weights are relations and knows how to take the tensor product during determinization. (The generic WFA determinization procedure takes essentially a callback procedure to perform the tensor product; the XFA class supplies the one for relations.)

4 A Binary Front End for Daikon

We need to find invariants of programs in order to create XFA models of them using the techniques described in Chapter 6.There are a number of ways it would be possible to do this, ranging from static analysis to a fully-dynamic approach. Our implementation uses the latter, using a tool called Daikon [32] to obtain invariants. This chapter describes a new front end for Daikon we wrote that operates on binaries and outputs different information from the standard Daikon front ends.

4.1 Daikon background

Daikon is a tool for dynamically finding invariants, developed by Ernst et al. At a high level, it monitors the execution of a program over many runs, and reports invariants (such as "at line 217, $x = y$") that held over all of the runs. (It is also capable of finding "near invariants," which hold only during most of the runs, and other similar facts; we do not use these features.) Figure 4.1 shows an example program and the invariants that the stock version of Daikon produces.

There is a reasonably extensive list of the form of invariants that Daikon will find, including equalities (e.g., $x = y$), inequalities (e.g., $x \leqslant y$), maximum and minimum values for variables, the ordering of values in an array, etc. It is capable of finding both "one vocabulary" invariants that hold at a particular program point (e.g., $x = y$ for program variables x and y) or "two vocabulary" invariants that relate the program state before and after a procedure call (e.g., $x = x_0 + 1$ if a procedure increments x).

Because Daikon operates dynamically, its results are only as good as the suite of tests it is run with. There is no guarantee that the "invariants" it produces are truly invariants — a reported invariant could be violated in situations for which there is no test, and hence Daikon never saw a case where the non-invariant failed to hold. (Section 6.4 discusses we chose Daikon in light of this limitation.) Nevertheless, Daikon is often useful in practice.

The primary use case of Daikon as it was first presented was as an aid to developers to help program understanding. Suppose that a developer wonders whether x always equals y at some point in the program so they could decide how to make a change to the program; Daikon was billed as a system that would tell them. Since then, Daikon has found uses in areas such as modifying programs to enforce Daikon-produced invariants in order to improve security [11, 66], reducing the number of mutants needed in mutation testing to characterize the quality of a test suite [74], characterizing the memory usage of procedures [15], and many other applications. (See the publications page of the Daikon web site [67] for a bibliography.)

Daikon's architecture is split into two parts; a block diagram illustration is shown in Fig. 4.1(a). There are several language-specific front ends that monitor the execution of the target program and output a *trace file* of the program's execution. (Daikon ships with front ends for Java, C and C++, and Perl. Other front ends are available for .NET, Eiffel, and IOA.[1]) The front ends also output information to the *declarations file* about what variables and program points are present in the program. One or more traces are then read by a universal back end that actually finds and reports the invariants. Figure 4.2 shows the declaration and trace file that are produced for the example program shown in Fig. 4.1.

Ordinarily a program needs to be run many times with different inputs, producing many different trace files. The back end aggregates information across all of the traces, looking for invariants that hold over all of them. In Figs. 4.1 and 4.2, for brevity we use an example that does not read input, and hence only needs one run.[2]

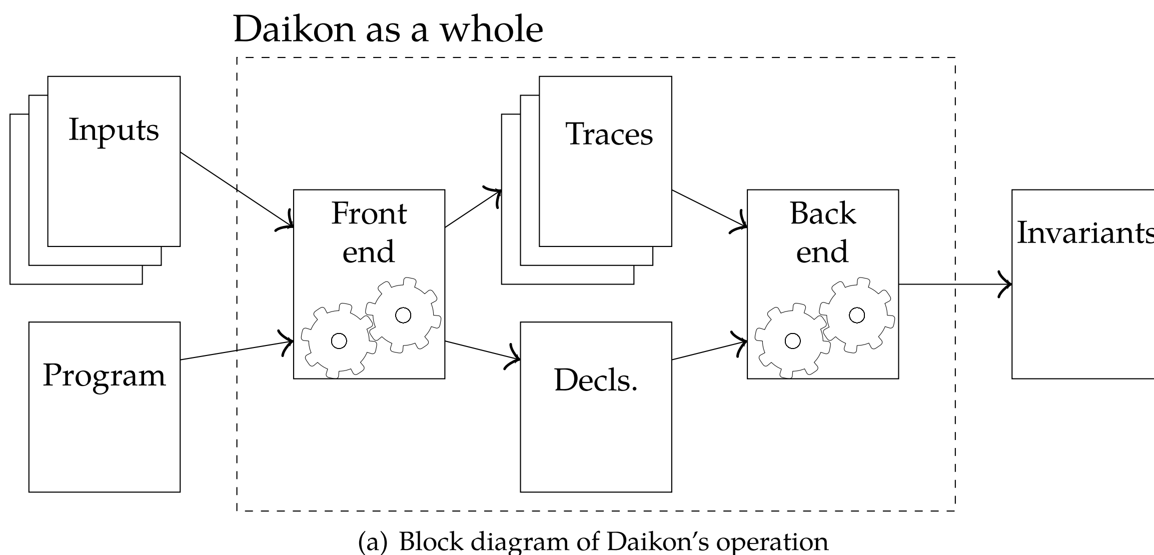
4.2 Snotra: a new Daikon front end

This section discusses our front end for Daikon, called Snotra.[3] Snotra operates on x86 or x86-64 binary code with debugging information, which in practice means it has been

[1] IOA is a formal modeling language for describing I/O automata [83].

[2] In general, even a program that has no apparent input could still have random or pseudo-random behavior, but the example in question does not.

[3] We continue a naming theme based on Norse mythology. The C/C++ front end that ships with Daikon is named Kvasir, after a Norse god of wisdom and knowledge. Kvasir is built using the dynamic-analysis framework Fjalar, named after one of two dwarves who killed Kvasir. (Also, one goal of Fjalar is to give easy access to the debugging information in an executable, which is stored in DWARF format.) In turn, Fjalar is built atop Valgrind, named for the main gate to Valhalla. Finally, Snotra is a goddess described as wise.



(a) Block diagram of Daikon's operation

```
int x = 0;

void inc_x() {
    ++x;
}

int main() {
    inc_x();
    inc_x();
    inc_x();
    return 0;
}
```

(b) An example program

```
..inc_x():::ENTER
::x one of { 0, 1, 2 }
======================
..inc_x():::EXIT
::x one of { 1, 2, 3 }
======================
..main():::ENTER
::x == 0
======================
..main():::EXIT
return == orig(::x)
::x == 3
return == 0
```

(c) Invariants

Figure 4.1: Diagram of Daikon's operation, including example program and invariants. The invariants were gathered using the stock Daikon front end Kvasir.

```
input-language C/C++
decl-version 2.0
var-comparability none

ppt ..inc_x():::ENTER
  ppt-type enter
  variable ::x
    var-kind variable
    rep-type int
    dec-type int

ppt ..inc_x():::EXIT0
  ppt-type subexit
  variable ::x
    var-kind variable
    rep-type int
    dec-type int

ppt ..main():::ENTER
  ppt-type enter
  variable ::x
    var-kind variable
    rep-type int
    dec-type int

ppt ..main():::EXIT0
  ppt-type subexit
  variable ::x
    var-kind variable
    rep-type int
    dec-type int
  variable return
    var-kind variable
    rep-type int
    dec-type int
```

(a) The declarations file

```
input-language C/C++
decl-version 2.0
var-comparability none

..main():::ENTER
this_invocation_nonce
0
::x
0      <--- x's value
1

..inc_x():::ENTER
this_invocation_nonce
1
::x
0      <--- x's value
1

..inc_x():::EXIT0
this_invocation_nonce
1
::x
1      <--- x's value
1

..inc_x():::ENTER
this_invocation_nonce
2
::x
1     <--- x's value
1
```

```
..inc_x():::EXIT0
this_invocation_nonce
2
::x
2      <--- x's value
1

..inc_x():::ENTER
this_invocation_nonce
3
::x
2      <--- x's value
1
..inc_x():::EXIT0
this_invocation_nonce
3
::x
3      <--- x's value
1

..main():::EXIT0
this_invocation_nonce
0
::x
3      <--- x's value
1
return
0
1
```

(b) Trace file

Figure 4.2: The declarations file and trace file produced during the run of the program in Fig. 4.1, using the stock Daikon and C/C++ front end Kvasir. Note how, at each program point, it outputs the value of x, which is incremented between the entry and exit to `inc_x`.

compiled from C or C++ code. (However, the following section discusses ways in which this restriction could be loosened substantially.)

As discussed in Chapter 6, for compatibility checking we are interested in finding equalities between two values:

The value of a field in the message. In the producer, the value of each field manifests as a parameter to an output procedure (or a value pointed to by a parameter). In the consumer, the field value manifests as the return value of a procedure (or a "return value" passed via an output parameter).

The number of repetitions of a field or group of fields. If a program can read a variable number of repetitions of a field, there essentially must be some form of loop in the program that does it. Snotra's goal is to find the corresponding loop and count the number of iterations. The number of iteratons of a loop is called that loop's *trip count*; Snotra will add an instrumentation variable called a *loop trip counter* (or just trip counter) to track it. The actual field in the message will be called the *repetition count*. (Trip counters were introduced by Saxena et al. [73].)

We look for such equalities at the exits of loops (or "at loop exits"). Later in this section, we describe how Snotra instruments programs to obtain this information. (*Instrumenting* a program means that we modify the program to change its behavior slightly; usually it consists of adding code that tracks additional information, which is the instrumentation code.) Chapter 6 discusses how such equalities are actually used in format inference.

Section 4.2.1 provides motivation for creating our own front end, including discussions of how Snotra could be applied to other tasks. Section 4.2.2 discusses how Snotra works in general. Sections 4.2.3 and 4.2.4 describe how Snotra obtains field values and trip counts, respectively. Section 4.2.5 gives a full example program and the instrumentation added. Also relevant to this section is a discussion of the "quality" of our choice of instrumentation and instrumentation techniques; however, we defer this discussion until Section 6.4.

4.2.1 Motivation

The front ends that come with Daikon all have the same basic behavior, which is to instrument the entry and exit point of each procedure. At these points, it outputs the values of

variables that are either global (in Fig. 4.1, `x`) or parameters to the procedure; at exit points, it also outputs the value that is being returned (in Fig. 4.1, this shows up in the declarations file, trace file, and final invariants as `return`).

However, the stock front ends do not output information at other program points within a procedure, and (somewhat as a result) they do not output the values of local variables. As will be seen in Section 4.2.2, we need different information from different program points, so the stock front ends are insufficient.

Rather than modify one of the existing front ends, we choose to create our own, called Snotra. In addition to making it possible to output the information we need, Snotra brings the additional benefit that the design is potentially more flexible in terms of the target programs it analyzes. (As may become clear soon, this benefit is primarily applicable to applications other than format compatibility. The techniques described in Part II of the dissertation are primarily aimed at developers writing or maintaining a piece of software, at which point requiring debugging information is reasonable. The potential benefits of Snotra over the existing Daikon front ends are largest when analyzing programs that *others* have written, when no debugging information is available.) Additionally, Kvasir only works on Linux systems, and the alternative C/C++ front end (which runs on other systems) requires the non-free software Purify.

The preferred front end that ships with Daikon for finding invariants in C and C++ code is called Kvasir. Currently, Kvasir and Snotra both operate on executables with debugging information compiled from C and C++ code. However, the latter two assumptions (that debugging information is present and that the original language was C or C++) are very lightweight in Snotra. The only hard dependence on the C/C++ requirement in Snotra is the presence of the debugging information, while Kvasir's dependence runs deeper.

Furthermore, Snotra's use of debugging information was designed to be relatively lightweight and replaceable by information recovered from other analyses such as those in CodeSurfer/x86 [10]. Snotra uses debugging information for three purposes:

1. Finding procedures,

2. Finding the variables (locals, parameters, and globals) that are visible at each program point of interest, and

3. Finding the types of those variables.

If an alternative source of these three pieces of information is provided, Snotra would be able to replace debugging information with that source.

The first and second pieces of information can be recovered from CodeSurfer/x86 analyses. In particular, variable information can be replaced by the *abstract locations* (a-locs) recovered by CodeSurfer/x86's value-set analysis (VSA). That is, instead of printing out the value of each variable at a point of interest, Snotra would print the value of each a-loc.

The types of variables could be addressed via two means. CodeSurfer/x86's version of aggregate-structure identification (ASI), originally developed by Ramalingam, Field, and Tip [70], would recover some type information; in particular, ASI would split aggregate structures into their components and find the sizes of most integer and pointer types. Determining additional information, such as the signed-ness of integer values or distinguishing numeric integers from pointers would require a new analysis that would look at how they are used. However, the additional information is unnecessary for many applications: simply knowing some candidate invariant about the raw value of a memory address could be enough to help *other* analysis tools, such as McVeto [82, §3.2], that can make use of candidate, but unproven, invariants.

While Snotra has not really been used in such a fashion yet, Junghee Lim briefly used it for a machine-code model-checking tool called McTreeIC3. She described Snotra as being promising and useful, but did not have time to integrate it fully into McTreeIC3 before leaving the University of Wisconsin, and the project has not been picked back up yet. In any case, the additional flexibility and potential for future use served as an additional reason we created our own front end instead of modifying Kvasir.

In the interest of full disclosure, Snotra also falls short of Kvasir in some areas. One obvious area is just the engineering effort that has gone into the respective tools; Daikon and Kvasir are mature projects, while Snotra is in its infancy. However, there is a more fundamental limitation. Kvasir is ultimately built on top of the dynamic-analysis tool Valgrind, which is primarily associated with a tool called *Memcheck* (distributed with Valgrind) that detects memory errors. Many of the same capabilities that allow Memcheck to find errors are also used by Kvasir. For instance, Kvasir knows about the bounds of heap blocks through these mechanisms, so if the target program allocates an array on the heap, Kvasir can print out the value of the array. Kvasir is also aware of what memory is uninitialized, and uninitialized variables are printed with a special indicator of that

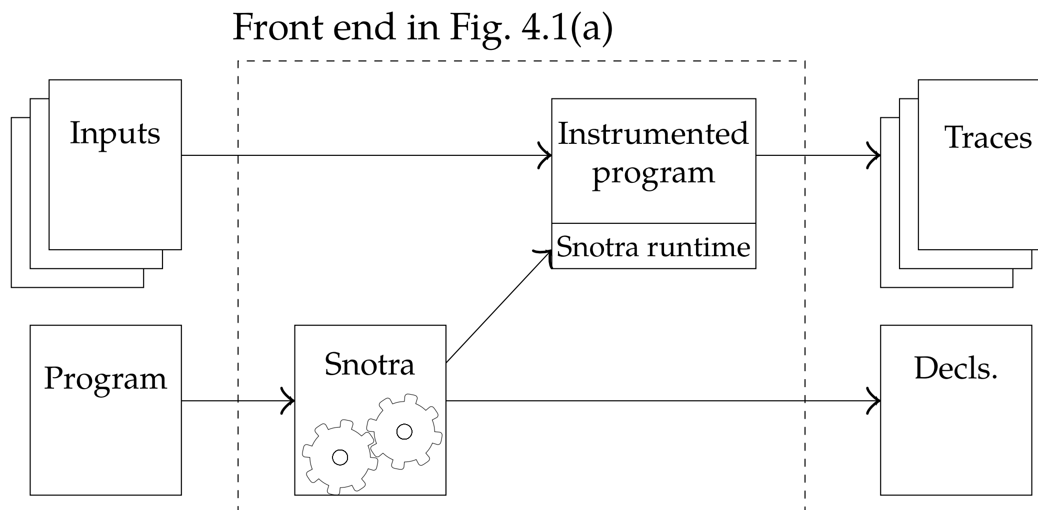


Figure 4.3: A block diagram of Snotra's operation.

fact (rather than just printing whatever value happens to be at that location). It would be difficult (or impossible, in some cases) to make Snotra behave similarly. These limitations do not affect Snotra's application to format analysis described in Chapter 6, however.

Finally, as part of their ClearView system for automatically patching errors in deployed software, Perkins et al. created a different x86 front end for Daikon [66]. However, it outputs far different information than Snotra does, and Perkins et al.'s tool does not add any additional instrumentation like what we need to find equality invariants between field values and repetition counts. (Their tool also does not seem to be publicly available.)

4.2.2 How Snotra works

Snotra primarily consists of an instrumentation engine that modifies the target program to output the values of variables at program points of interest. (There is a portion of Snotra that is loaded into the instrumented program and assists with this task.) Snotra also outputs the declarations file containing information about what the program points and variables of interest are. A diagram showing how Snotra works is given in Fig. 4.3.

To perform the instrumentation, Snotra makes use of the Dyninst library [68]. Dyninst is a library that makes it relatively easy to modify and analyze a target binary program. Modifications take the form of injecting additional code; the injected code is able to access

memory of the target program. Dyninst takes great care to not affect the execution of the program under test other than what is explicitly requested — e.g., inserting instrumentation code that does nothing should have no effect on the program's behavior. For analysis, Dyninst provides access to the procedures and variables in the target program, produces control flow graphs for procedures, and gives access to information such as the loops in a program (see Section 4.2.4).

Snotra has two modes: *Kvasir-emulation* and *compatibility-instrumentation*. In the Kvasir-emulation mode, Snotra outputs the same instrumentation as Kvasir (subject to the limitations discussed at the end of Section 4.2.1 and others); this mode is primarily used as a debugging tool, so we can compare the outputs of the two tools and make sure that they agree. We will assume Snotra is operating in the compatibility-instrumentation mode.

4.2.3 Instrumentation for field values

Before we describe what Snotra does, a brief word on what Snotra does *not* do is in order. For simplicity, suppose that we are instrumenting the consumer, that there is a single procedure, `readInt`, in which we are interested, and that `readInt` returns the value read. (Later in the section we discuss alternatives.)

The simplest instrumentation scheme that may come to mind would be to output information about the return value from `readInt` each time it is called:

```
1 int* readArray()            int* readArray()
2   int n := readInt()          int n := readInt()
3                               OUTPUT_INFO_ABOUT(n)
4   int* a := new int[n]        int* a := new int[n]
5   for i := 1 to n:       →    for i := 1 to n:
6     a[i] := readInt()           a[i] = readInt()
7                                 OUTPUT_INFO_ABOUT(a[i])
8                               OUTPUT_INFO_ABOUT_LOOP()
9   return a                    return a
```

where `OUTPUT_INFO_ABOUT(n)` prints information about the line it is on along with the value of n, and `OUTPUT_INFO_ABOUT_LOOP()` prints information about the previous loop (see Section 4.2.4). (Snotra works on binaries, as said before, but we show examples in pseudocode for expository purposes.)

Unfortunately, this simple approach does not work. Recall that we are ultimately interested in knowing about relationships between n and the trip count of the loop on Lines 5–7, though the latter information is unavailable on Line 3. With the exception of procedure-entry to procedure-exit, the Daikon back end does not look at the relationships between variable values across program points. Splitting the information we need across two program points (Lines 3 and 8) will thus not work.[4]

Furthermore, we cannot simply delay outputting information about n until the end of the loop, because there is no guarantee that it will remain unchanged:

```
1 int* readArray()              int* readArray()
2    int n := readInt()            int n := readInt()
3    int* a := new int[n]          int* a := new int[n]
4    while n > 0:                  while n > 0:
5       a[i] := readInt()    →        a[i] := readInt()
6       n := n - 1                    n := n - 1
7                                  OUTPUT_INFO_ABOUT_LOOP_AND(n, a[i])
8    return a                      return a
```

This version of the program is effectively the same as before, but because n will always have the value 0 after the loop's exit, we no longer have the information we need.

What we do instead is to follow an approach similar to the latter example, but we first add a new variable, n_0, to the program and set it to the value read on Line 2. After the loop, we output information about n_0 rather than n:

```
1 int* readArray()              int* readArray()
2    int n := readInt()            int n := readInt()
3                                  int n_0 := n     <-- instrumentation
4    int* a := new int[n]          int* a := new int[n]
5    while n > 0:                  while n > 0:
6       a[i] := readInt()    →        a[i] := readInt()
7                                     a_i0 := a[i]
8       n := n - 1                    n := n - 1
9                                  OUTPUT_INFO_ABOUT_LOOP_AND(n_0, a_i0)
10   return a                      return a
```

[4]We also cannot pretend that Lines 3 and 8 are one program point or are procedure-entry and procedure-exit, because we do not know a priori which input fields are associated with which loops.

We will refer to the instrumentation variables that are added during this step as *I/O variables*.

A new I/O variable is always added, regardless of whether the original is overwritten or not, which saves Snotra from having to determine whether or not to add a variable.

The instrumentation works as follows:

1. Find call sites that call I/O procedures
2. For each call site:
    a) Add one or more new global variables[5] for each of the values written or read
    b) Set each variable to the value read or written (see below)

Adding the I/O variables to the list of variables also ensures that it will be printed at loop exit points, as described in the following section.

The one item that remains to describe is how the value read or written is determined for the instrumentation described in (2b). There are four cases that Snotra is able to handle:

Case 1: Input procedures that return the value in question explicitly. For example, `readInt()` in the above examples behaves this way. For this case, the I/O variable corresponding to the call site is set to the return value, as shown above, after the procedure returns.

Case 2: Input procedures that "return" the value in question via an output parameter. An example is `scanf`; the approximate equivalent of `n := readInt()` is `scanf("%d", &n)`. For this case, the user of Snotra specifies which parameter(s) receive the input values, and Snotra adds instrumentation that, after the procedure returns, dereferences those arguments and places the values into their I/O variables. For the case of `scanf("%d",&n)`, the instrumentation essentially takes the form of $\mathtt{n}_0$ `:= *(&n)`. (The full capabilities of `scanf` are beyond the capabilities of Snotra at present, but because we have a library that can interpret format strings, it would be possible to add a "`scanf` model" that would support cases where the format string is constant.)

Case 3: Output procedures that are passed the value as a parameter. For example, `putchar(n)` writes `n` as a byte to standard output. For this case, the user of Snotra specifies what

[5]Globals allow the statement that reads a field value to be in a separate procedure from any loops with a trip count that will be found to equal the field value.

parameter(s) receive the output values, and Snotra adds instrumentation that simply copies the values of those parameters into their I/O variables.

Case 4: Output procedures that are passed a pointer to the value as a parameter. For example, `write(fd, &n, sizeof(n))` writes the contents of `n` to the stream specified by `fd`. Similar to the analogous case for input variables, the user specifies what parameter(s) receive the pointers and Snotra adds instrumentation that copies the values at those addresses to their I/O variables. (Like `scanf`, Snotra is currently unable to distinguish between different "kinds" of uses of `write` when, for example, the size of the write potentially differs. It would be possible to add a model for `write`-like calls that would support its behavior.)

4.2.4 Instrumentation for loop trip counts

Finally, we discuss how Snotra instruments programs to discover loop trip counts. As for the case of I/O variables, we cannot depend on the program to track the right thing. Many loops will have a program variable that counts iterations, but they are not necessarily used as we need. For instance, program variables could count up by a number other than 1 (especially after the compiler's optimizer goes to work), could count down instead of up (the situation in the second two examples in the previous subsection), or may be entirely absent if a loop is controlled by something other than an explicit count.

To instrument loops, we first use Dyninst to detect them. The combination of Dyninst and Snotra recovers three pieces of information: control-flow graph (CFG) edges that start outside the loop and end inside the loop (*enter* edges), CFG edges that start inside the loop and end outside the loop (*exit* edges), and CFG edges that, informally, go from the end of the loop back to the start (*back* edges). It is possible for there to be more than one edge per loop of any of the three kinds of edges. Dyninst makes it possible to add instrumentation that executes whenever a particular CFG edge is taken, and we add instrumentation for all three kinds of loop edges.

In addition to adding code to track loop trip counts, Snotra must add the code that actually outputs the trace file. This is done on loop exit. We are interested in equalities between I/O variables and trip counters, and in most cases the exit of a loop is the right place to find such equalities. By that point the loop is done executing, so the trip count

will not change any more. Because we are assuming the programs write to or read from a stream, as long as the message field holding the repetition count appears earlier in the stream than the repeated fields, the program also cannot read the repetition count after that point. Thus checking for equalities that hold at loop exit is sufficient, and it is easy to do from an implementation standpoint.

Snotra's loop instrumentation proceeds as follows:

1. Add instrumentation code that, when the program starts, outputs some metadata about the program
2. Obtain the list of loops from Dyninst
3. For each loop L:
   a) Add a new global variable k_L that will act as the loop trip count
   b) To each enter edge of L, add the instrumentation $k_L := 0$
   c) To each back edge of L, add the instrumentation $k_L := k_L + 1$. (Per the next section, there will always be exactly one back edge in the current version of Snotra.)
   d) For each exit edge e of L, perform the following:
      i. Add instrumentation to e that outputs some metadata about the current loop L
      ii. For each global instrumentation variable x, add instrumentation to e that reports the value of x (along with metadata about x)

Loop detection

Snotra relies on Dyninst's loop-detection algorithms. Dyninst, in turn, finds *natural loops* in the control-flow graph (CFG) of the target program. A natural loop is a maximal region of the CFG that meets the following requirements:

1. There is a single node, called the loop's *header*, that *dominates* all nodes in the region. That is, any path from outside the region to inside the region must pass through the header.
2. There is a path from every node in the region back to the header.

3. Furthermore, there is a single edge (the loop's back edge) for which:
   a) the target of the back edge is the loop's header, and
   b) from every node n, every path from n to the header that stays within the region traverses the back edge.

(This is not the usual way these terms are defined, but it is equivalent and simpler to describe.) There are well-known algorithms for finding natural loops; see, e.g., Aho et al. [1, §9.6]. Dyninst uses such an algorithm. The back edges referred to in the previous definition are the same back edges to which instrumentation is applied.

Ideally the natural loops of a program would correspond to the programmer's notion of the program's loops; unfortunately, the correspondence is not perfect, especially in binary programs. We will show two potential ways in which the correspondence can be violated, and argue that the consequences are small for the application of compatibility checking. The second of these violations is discussed in the next section, on irreducible graphs.

The first way in which the correspondence can be violated is if the compiler (in particular, the optimizer) performs a transformation that "splits" a single source-level loop into multiple natural loops. Figure 4.4 illustrates this possibility, showing original source code that has a single `while` loop, a lowered form that only uses conditional `gotos`,[6] the CFG of the lowered version, and the CFG of an optimized version that has two natural loops. The reason that the optimizer may perform the transformation illustrated by Fig. 4.4 is because it would subsequently be able to collapse Node 5 into Node 3, removing a jump instruction that just targets another jump, reducing the number of instructions that need to be executed when the `if` statement in Line 2 is false.

As described so far and as implemented in Snotra, each natural loop receives its own trip counter. In an example such as that in Fig. 4.4, this splitting is almost certainly undesirable. It may be better to merge any loops that have the same header, and add a trip counter for each loop header that gets incremented on all of that larger loop's back edges.[7]

Incorrectly-detected loops (e.g., using a split natural loop) will only have the effect of coarsening the overapproximation of the program models we build. There are two potential

[6] The `end_of_if` label would be present in case there was code after the `if` statement.

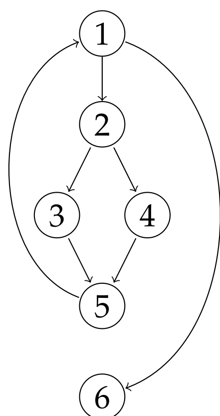
[7] Actually this proposal is not quite ideal either, because if a nested loop shared the same header as an outer loop we would like to keep that separate. What we would want to do is merge any natural loops that share the same header but for which none is a subgraph of another.

```
gcd(a, b):
    while a != b:
        if a > b:
            a := a - b
        else:
            b := b - a
    return a
```

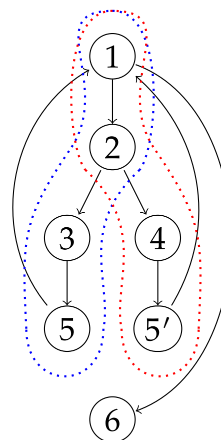
(a) Example program

```
  gcd(a, b):

  loop_head:
1   if a != b goto out

2   if a > b  goto a_bigger

    (* false branch of 'if' *)
3   b := b - a
    goto end_of_if

  a_bigger:
    (* true branch of 'if' *)
4   a := a - b

  end_of_if:
5   goto loop_head

  out:
6   return a
```

(b) A lowered form of the program in (a) that only uses branches instead of loops



(c) CFG for the example program. The numbers in the nodes correspond to the line numbers in (b).



(d) A plausible transformation of the CFG shown in (c), potentially created by a compiler's optimizer. The two dotted outlines show the two natural loops.

Figure 4.4: Splitting a single source-level `while` loop into two natural loops.

effects of incorrect loop detection on the final result. The more likely scenario is that using natural loops will cause fewer invariants to be reported. The invariants are used to generate semantic constraints, and thus fewer invariants lead to fewer semantic constraints. The result is that the model is a coarser overapproximation of the target program's format. Because both models are overapproximations, the reduction in precision can leave the answer unchanged, make a true report into a false one, or make a false report into a true one.

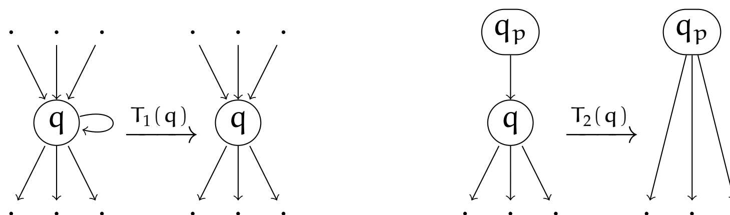
The less likely scenario that can be caused by using natural loops is that an additional constraint is *added*. Suppose that a program reads a number n and then executes a loop with the structure of Fig. 4.4(d), exactly one branch of which performs an I/O operation. Furthermore, suppose that the number of times *that branch* of the `if` statement in the loop is executed equals n, but there are be additional executions that take the other branch. In this case, the loop trip counter k for the natural loop corresponding to the branch with the I/O will be found to equal n. And, assuming that $n = k$ is a true invariant rather than one only produced because of inadequate tests, the result will actually be a model that is *more precise* than what would have been achieved had the two natural loops been merged. (Again, the additional precision can turn a false report into a true one ore vice versa.)

Because suboptimal loop detection processes do not have a clear negative effect on the result if they do not happen frequently, we do not take extraordinary steps to try to control them. The loop merging is something that we would like to attempt, but it is not implemented at this point.

Irreducible graphs

The second way in which the correspondence between program loops and natural loops can be broken is if the control-flow graph is irreducible. An *irreducible graph* is one that can only be directly formed out of unstructured control flow constructs — essentially, unrestricted use of `gotos`; in a moment, we provide a formal definition. However, like the loop-splitting example, the optimizer could potentially introduce unstructured control flow.

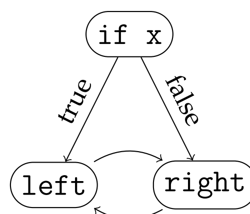
There are several equivalent definitions of irreducability; we give one. Define two graph transformations $T_1(q)$ and $T_2(q)$ as follows. If a node q has a self loop, $T_1(q)$ removes the self loop. If a node q has a single predecessor q_p, $T_2(q)$ collapses q into its predecessor (i.e., redirect outgoing edges from q so they originate from q_p and remove q). A graph G



(a) Illustrations of the T_1 and T_2 transformations.

```
void f(bool x) {
    if x:
        goto right
    else:
        goto left
 left:
    goto right
 right:
    goto left
}
```

(b) Procedure with irreducible control flow



(c) Control flow graph of `f()`

Figure 4.5: Irreducible control flow

contains an irreducible region iff G cannot be reduced to a single node using a sequence of T_1 and T_2 transformations. Figure 4.5(a) illustrates the two transformations, and Figures 4.5(b) and 4.5(c) illustrate the stereotypical example of irreducible control flow.

The reducibility of the control-flow graph is important when it comes to loop detection because the back edge of a loop only has an unambiguous definition for reducible graphs (or reducible portions of an irreducible graph). In fact, even the definition of a "loop" arguably becomes muddled. The irreducible components of a graph, such as is shown in Fig. 4.5(c), do not have any back edges, despite there being a cycle present.[8]

[8]Our terminology is that used by Aho et al. [1, §9.6.4], which distinguishes between back edges and a more general kind of edge called a *retreating edge*, where an edge $q \to p$ is retreating if an arbitrarily-chosen depth-first search visits p before q. Not everyone makes such a distinction, and "back edges" are sometimes defined to match this definition of retreating edges. For irreducible graphs, retreating edges are not uniquely determined — different depth-first search orders can produce a different set of retreating edges. However, in a reducible graph all depth-first search orders result in the same set of retreating edges, which provides an alternative definition of *reducible* than the one we give: a graph is reducible if every retreating edge is a back edge (by our definition).

Snotra will not detect cycles in the CFG that are present only because of an irreducible section of the graph. (There are no back edges in such a cycle, and thus the algorithm for finding natural loops, which is what Dyninst and, in turn, Snotra use will not report any back edges.) Like the more likely scenario for the consequences of loop splitting, ignoring irreducible cycles could potentially lead to a coarser overapproximation. However, the same reasoning given before applies here: if irreducible cycles are infrequent enough, using a more sophisticated technique to handle them is not a high priority. While we do not measure the frequency, common wisdom is that irreducible regions of a graph are very uncommon.

There are two potential approaches we could use if we did want to handle irreducible cycles better. The first is that we could simply designate an arbitrary (or heuristically-chosen) edge as the "back edge." This would sometimes lead to picking up any semantic constraints that would otherwise be missed. The second approach is a more standard technique for addressing irreducibility, which is to clone part of the CFG (see Aho et al. [1, §9.7.6]). By duplicating portions of the graph and redirecting some edges to the copies, it is possible to transform an irreducible graph into a reducible one that has the same behavior.

4.2.5 Example instrumentation

This section provides an example program along with its instrumentation. (The instrumentation is inserted by hand as an illustration, as Snotra does not operate at the source level. The names of the instrumentation variables are chosen to aid the reader.) Figure 4.6 shows an example program, and Fig. 4.7 shows an version with the instrumentation that we add.

Note that this example illustrates one of the tradeoffs of our instrumentation strategy that is discussed in Section 6.4, which is that at any given point, only the most-recently read (or written) value at an I/O statement is remembered. For instance, after the loop on Lines 10–20 exits, only the latest value read on 12, 15, and 18 are available.

```
int nrows, ncols;
nrows = read_int();
ncols = read_int();
pixel ** image = malloc(sizeof(*image) * nrows);
for (int r=0; r<rows; ++r) {
    image[r] = malloc(sizeof(**image) * ncols);
    for (int c=0; c<cols; ++c) {
        read_byte(&image[r][c].red);
        read_byte(&image[r][c].green);
        read_byte(&image[r][c].blue);
    }
}
```

Figure 4.6: Original program

```
1   int N0_nrows, N1_ncols, N2_red, N3_green, N4_blue,
        K0_outer, K1_inner, *TEMP;
2   int nrows, ncols;
3   nrows = N0_nrows = read_int();
4   ncols = N1_ncols = read_int();
5   pixel ** image = malloc(sizeof(*image) * nrows);
6   K0_outer = 0;
7   for (int r=0; r<rows; ++r, ++K0_outer) {
8       image[r] = malloc(sizeof(**image) * ncols);
9       K1_inner = 0;
10      for (int c=0; c<cols; ++c, ++K1_inner) {
11          TEMP = &image[r][c].red;
12          read_byte(TEMP);
13          N2_red = *TEMP;
14          TEMP = &image[r][c].green;
15          read_byte(TEMP);
16          N3_green = *TEMP;
17          TEMP = &image[r][c].blue;
18          read_byte(TEMP);
19          N4_blue = *TEMP;
20      }
21      OUTPUT_PROGRAM_POINT_HEADER();
22      OUTPUT_INT("N0_nrows", N0_nrows);
23      OUTPUT_INT("N1_ncols", N1_ncols);
24      OUTPUT_INT("N2_red",   N2_red);
25      OUTPUT_INT("N3_green", N3_green);
26      OUTPUT_INT("N4_blue",  N4_blue);
27      OUTPUT_INT("K1_inner", K1_inner);
28  }
29  OUTPUT_PROGRAM_POINT_HEADER();
30  OUTPUT_INT("N0_nrows", N0_nrows);
31  OUTPUT_INT("N1_ncols", N1_ncols);
33  OUTPUT_INT("N2_red",   N2_red);
34  OUTPUT_INT("N3_green", N3_green);
35  OUTPUT_INT("N4_blue",  N4_blue);
36  OUTPUT_INT("K1_outer", K0_outer);
```

Figure 4.7: Instrumented program. (The `TEMP` variable is really just whatever memory location the argument is stored at and is not explicitly materialized.)

Part II

Application Compatibilty Checking

5 Control-Flow Format Compatibility

We can now discuss how we use automata to perform format-compatibility checking. Recall that the goal is stated as follows:

> Given a program P that writes a stream of output and a program C that reads it, determine whether every message that P can write is understandable by C.

We address this question by modeling P and C as automata and checking language containment.

In this chapter, we describe how to create regular and nested-word models of programs and how to use the models to perform format-compatibility checking. We begin with an informal overview of the technique, and then move into specifics. Chapter 6 describes how we use extended finite automata to model semantic constraints on the input, which is where Snotra and Daikon enter the picture. The techniques presented in this chapter are implemented in a tool called the Producer-Consumer Conformance Analyzer (PCCA).

5.1 Overview

Consider an example system made up of the producer and consumer shown in Listings 5.1 and 5.2, respectively. The producer is a program that monitors a sensor, and periodically sends a "packet" of data to the consumer.[1] The system uses an abbreviated protocol: if the sensor data has not changed since the last message, then only the Boolean literal `false` is sent. Line 2 in Listing 5.1 makes this decision.

As presented, these components are correct: both "speak" the same protocol. However, consider a buggy version of the consumer that does not account for the possibility that the producer sends an abbreviated message, and instead always expects the full packet. Buggy

[1] We use *packet* to refer to the data that the components communicate each time through their "loop", but packets are not a first-class notion in either the example programs or our analysis technique.

```
1  sendReading(Sensor* device, int prev)
2    if device→setting == prev then
3      writeBool(false);
4    else
5      writeBool(true);
6      writeDouble(device→setting);
7      writeBool(device→valid);
8  loop(Sensor* device, int prev)
9    ... // update device with new readings
10   sendReading(device, prev);
11   if ... then
12     loop(device, device→setting);
13 main()
14   Sensor device;
15   loop(&device, -1);
```

Listing 5.1: Example producer

```
1  updateReading(int* setting, bool* valid)
2    *setting = readDouble();
3    *valid = readBool();
4  main()
5    int setting;
6    bool valid;
7    while ... do
8      if readBool() then
9        updateReading (&setting, &valid);
10     ... // do something with current readings
```

Listing 5.2: Example consumer

```
1  main()
2    while ... do
3      readBool();
4      updateReading (&setting, &valid);
5      ... // do something with current readings
```

Listing 5.3: Example buggy consumer. (`updateReading` is the same as in Listing 5.2.)

code that acts this way is shown in Listing 5.3. (Perhaps the specification of the format changed partway through the development, and the consumer was not updated.)

To find this incompatibility bug, we reason about the languages over which each component operates. In the consumer, we know that the `updateReading` procedure always reads a `double` and then a `bool`. Thus, each time through the loop in the buggy version of `main`, the consumer reads a `bool` then the `double`–`bool` sequence from `updateReading`. Thus we can determine that the input language of the buggy consumer, expressed as a regular expression over types that the consumer reads, is

`(bool double bool)*.` (buggy consumer)

Similarly, we can determine that the output language of the producer, expressed as a regular expression, is

`(bool | bool double bool)*.` (producer)

From these two language descriptions we can see that one of the components is buggy: the string `bool bool` is in the producer's language but not in the consumer's. The non-containment suggests that some execution of the producer could output two Boolean values, but *no* execution of the consumer would expect to read that message.

A similar analysis suggests that Listing 5.2's consumer is correct. The language it expects is

`(bool (double bool)?)*,` (correct consumer)

which is equivalent to what we inferred for the producer.

The description above is given in terms of regular expressions, but the technique described in this chapter operates instead on finite automata. Inferring the automata is done by constructing the control-flow graph of the program, which with suitable modifications is then essentially treated as an automaton. Procedure calls that perform input or output (as appropriate for whether the program being analyzed is the producer or consumer) are labeled with the type of datum that is written or read, and other transitions are labeled with ε. The language of the resulting automaton is then an overapproximation of the message format (as we define it) of the program in question.

Section 5.2 covers how we build an automaton that models the input or output behavior of a program. When we infer finite automata, the only step left is to check whether the

language of the producer's model is a subset of the language of the consumer's: if so, PCCA reports that the components are compatible, otherwise it reports that the components are incompatible. An alternative to standard FAs is to use NWAs. The NWA construction is also described in Section 5.2, and its benefits are described in Section 5.2.2. However, as mentioned back in Chapter 1, using NWAs makes the compatibility question more complicated; Section 5.3 describes the "Enrich" process, which effectively further approximates the consumer's model and allows us to perform the inclusion test. Section 5.4 describes ways in which the "message format" can be interpreted more broadly than it may initially appear, to gain additional precision or to be used for other applications.

The techniques described in this chapter are implemented in a tool called PCCA. Section 5.5 describes implementation-specific information about PCCA, including how the user tells PCCA about what procedures perform I/O. Finally, Section 5.6 presents experimental results.

5.2 Building FA and NWA Models of a Program

The first step in the compatibility-checking process is to infer an automaton that approximates the language of each component. In the case of the producer, we wish to infer the language of all possible outputs; in the case of the consumer, we wish to infer the language of all expected inputs.

Our technique creates automata that mimic the control-flow behavior of the source programs. Each automaton that PCCA generates has the same language as one created by transliterating the program's interprocedural control-flow graph (ICFG) into an automaton in the following manner:

1. There is one state $\tilde{c}$ for each ICFG node c.

2. If a call site c can call an I/O procedure that outputs or expects a value of type τ, we add a transition on τ from $\tilde{c}$ to its corresponding return node. In the NWA model, this is an internal transition.

3. If a call site c can call a non-I/O procedure f with entry node f_e and exit node f_x, we add one transition from $\tilde{c}$ to $\tilde{f_e}$ and a second transition from $\tilde{f_x}$ to the corresponding return site. In the FA model, both transitions are ε transitions. In the NWA model,

the first transition is a call transition that occurs on the symbol $\langle$, and the second is a return transition that occurs on the symbol $\rangle$ when $\tilde{c}$ is the call predecessor.

4. All other transitions in the ICFG become ε transitions. In the NWA version, all are internal transitions.

5. The entry node of main becomes the start state, and the exit node becomes the sole accepting state.

However, if we used this naive translation, the resulting automata would be extremely large, which would cause problems during the determinization phase of PCCA.[2]

Instead of treating the ICFG as a whole, PCCA works procedure-by-procedure through the program. For each procedure, it looks at the *intra*procedural CFG in place of the full *inter*procedural CFG in step 1 and carries out the above translation, except that steps 3–5 are replaced by the following:

3′. If a call site c can call a non-I/O procedure *f*, we add an internal transition from $\tilde{c}$ to the corresponding return, labeled with a generated symbol `call_f`.

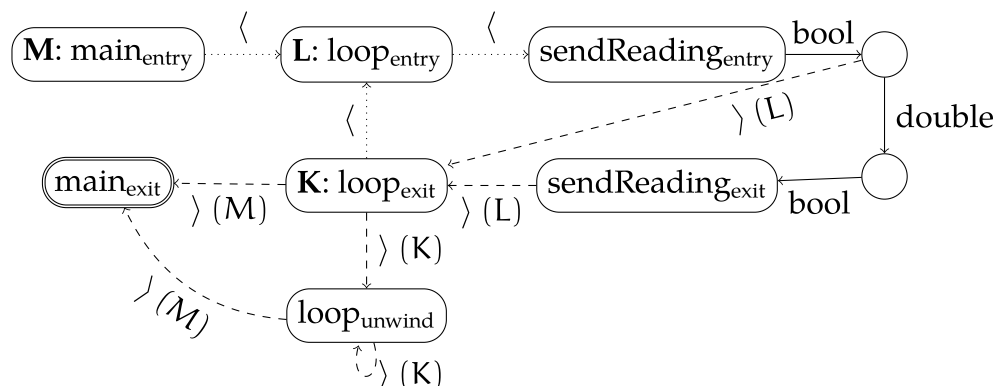
The automaton's starting and accepting states are the entry and exit nodes of that procedure.

At this point, we have a finite automaton for each procedure; even the NWA version can be interpreted as such because we never introduce call or return transitions. We then use standard algorithms to determinize and minimize each procedure's machine. (PCCA's implementation uses the OpenFST library for this purpose [3].) The efficiency upshot is that this technique turns what could be a multiplicative factor between procedures into an additive one, thus dramatically reducing the time spent in determinization.

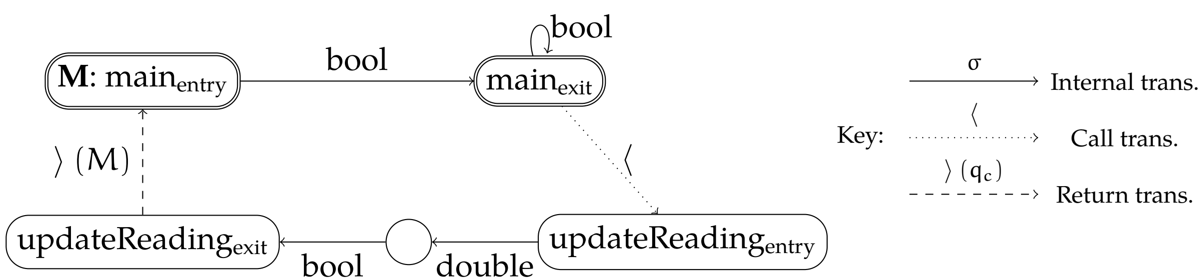
Once we have the collection of minimized automata, we combine all the automata into one and "restore" the call and return transitions. We replace each transition that moves from state $\tilde{c}$ to $\tilde{r}$ when reading a symbol `call_f` with a pair of transitions that match those in the original step 3:

6. We add a transition from $\tilde{c}$ to f's entry point. In the FA version, this is an ε transition; in the NWA version, it is a call transition on the symbol $\langle$.

[2]As shown in Section 5.6, determinization dominates execution time.



(a) The producer's NWA.



(b) The consumer's NWA.

Figure 5.4: The automata that are inferred for the producer and consumer shown in Listings 5.1 and 5.2. To reduce clutter, all transitions to the implicit "stuck state" are omitted, and ε transitions have been removed. (The latter transformation removes 7 states and a comparable number of transitions.) Return transitions are labeled with $\rangle\ (q_c)$, where q_c is the call predecessor. The finite-automaton models are the same except that $\langle$ and $\rangle$ transitions are labeled with ε instead.

7. We add a transition from the exit point of f to $\tilde{r}$. In the FA version, this is an ε transition; in the NWA version, it is a return transition labeled with the symbol $\rangle$ and with a call predecessor of $\tilde{c}$.

Finally, we have to perform one more determinization step in case connecting the procedures adds nondeterminism. This can happen if there is a loop or conditional where each branch calls a different procedure, as well as when there is an indirect call through a function pointer.

This translation essentially abstracts the program to its control flow only: data is not considered. One could envision a higher-fidelity translation that weaves selected data

elements (or abstractions of data elements) into the automata we infer, but of course there is a trade-off between precision and automaton size. Chapter 6 discussess attempts at addressing data using extended finite automata (XFAs).

Figure 5.4 shows the NWAs that are inferred from the code in Listings 5.1 and 5.2, respectively. Return transitions have labels of the form "$\rangle$/X", which means that the machine can make the transition only if state X is the call predecessor. The FA version is similar, except that all call and return transitions are replaced with ε transitions.

5.2.1 Knowledge about I/O procedures

PCCA needs information about what procedure calls can perform I/O. There are a number of ways the user can provide such information (see Section 5.5.1).

One important point is that there needs to be agreement between the producer and consumer regarding what types are used. The first, and easiest, issue related to this point is that the names of the types must agree.

The second issue is that the granularity of the I/O procedure specifications must agree. Consider our example. As written, both the producer and consumer have I/O operations expressed in terms of their constituent C types. It would also be possible to have the producer and consumer store values in a two-element structure `SensorData`, and do a "bulk read/write" with `fread()/fwrite()` to operate on the struct as a whole. In such a case, it would be reasonable to say that the type of that I/O operation was `SensorData`. This works fine, but the two approaches cannot be mixed: the consumer and producer need to agree on the granularity.

The need for agreement between the producer and consumer on the granularity of types is not a fundamental limitation: it would be possible to have the user specify that `SensorData` is a {`double`, `bool`} struct at either the format-inference stage or after the NWAs are constructed. It should even be possible to extract this information from struct definitions in the code. We have not investigated this avenue; however, with the current implementation the user has the ability to specify, for example, that a particular call to `fread/fwrite` operates on a `double` and then a `bool`.

5.2.2 Benefits of Using NWAs

The regular models described above are the most imprecise form that we consider. The first way we look to make models more precise is by making them context sensitive; for this purpose, we use NWAs.

The trade-offs between NWAs and FAs mirror trade-offs that one can make in traditional interprocedural dataflow analysis. The simplest way of performing an interprocedural analysis is to build the ICFG and run the analysis as if call and return edges were just normal intraprocedural control-flow edges. However, that approach loses precision because of suprious data flows from one call site c_1, into the called procedure `f`, and then out the return edge to a different call site c_2. No such program execution can actually occur, and so it is desirable to exclude such paths. A similar kind of imprecision can affect the FA version of PCCA. This imprecision is because the analysis is not context sensitive.

A *context-sensitive* analysis is an analysis that respects the proper control flow of procedure calls. A context-sensitive analysis will ignore properties that only arise because of *invalid paths*, which are paths that make a call from one call site but return to a different point.[3] A context-*in*sensitive analysis will incorrectly incorporate properties of invalid paths.

(A traditional dataflow analysis associates each program point with a fact that holds at that program point. The term "context sensitivity" comes from the fact that, for such an analysis, there can conceptually be multiple facts at each program point depending on the *context* in which the program point was called. For example, a program point `p` in procedure `f` could be associated with one fact that applies when `main()` calls `f()` directly and then execution reaches `p`, and a second fact that applies when `main()` calls another procedure `g()` that then calls `f()`. Even though execution has reached `p` in both cases, the calling context differs, and the presence of both facts means that the analysis is context-sensitive.)

Without special precautions to make the analysis outlined in the overview context sensitive, it will not be. In particular, if an invalid path π is the only possible path that

[3]Note that "valid" does not correspond to "feasible", where the latter means "can happen in an actual concrete execution of the program." A valid path can still be infeasible due to values of program variables. For example, a path that sets `x := 0` and then follows the true branch of `if(x>0)` is *infeasible*, but can still be *valid*. In this dissertation we consider all invalid paths to be infeasible, but even that is sometimes not true in real-world programs, where buffer overflows or other bugs can allow the return address to be overwritten and permit control flows that are otherwise infeasible.

witnesses the acceptance of a string s in the inferred automata, then s will be accepted by that automaton when it should not be. (That is, the model as described so far would accept s even though the program cannot write or read a message with the format specified by s).

For instance, Listing 5.5 shows a producer and consumer for which FAs and NWAs produce different results. The FA version of PCCA infers `int | char int char` for the language of the consumer, but `int | char int | int char | char int char` for the producer. The producer's language contains two words that are not in the language of the consumer, thus the FA version of PCCA reports that the components are incompatible.

Whether or not a path is invalid is a property that is context-free but not regular. Suppose that we are given a program and a candidate path that we wish to determine the validity of. What we do is follow the path through the program. When we reach a call site we push the name of the call site onto a stack. When we reach a return site, we make sure that the next state of the path corresponds to the call site that is at the top of the stack, and then pop it. However, if the call depth of a program can be arbitrarily high (which happens exactly when there is recursion), it is not possible to check validity of a path with finite storage and so we cannot use a regular model.

One way of getting around this problem is to perform procedure cloning: each call site `c` gets its own copy of the procedure `f`, which is only called from `c` and only returns to `c`. Cloning eliminates the suprious control flows, but at the cost of a model that is infinite in the presence of recursion and potentially exponentially larger than the original model even in the absence of recursion. It would be possible to do exactly the same thing in our domain — create a single FSM, but clone procedures — but the same drawbacks would apply.

Another approach would use pushdown automata (PDAs) to model the programs, using the PDAs' stacks to track calls and returns. PDAs would exclude from the models any strings that correspond only to invalid paths. However, it is not decidable whether two context-free languages (specified as PDAs) are equal or whether one is contained within the other. So while we could use PDAs for format *inference*, we would need to perform an approximation to determine compatibility between the two components.

Nested-word automata come to the rescue (to an extent, as we will see). NWAs are powerful enough to capture the properties that we need to determine whether a path is valid, as they have unbounded storage and a stack-like behavior. In fact, the process for

deciding whether a path is valid is exactly what a visibly-pushdown automaton (VPA) will do when reading a string, if that VPA was created by converting an NWA to a VPA [7, §4]. Furthermore, as explained before, NWAs are closed under complementation and intersection, and hence language inclusion is decidable.

What we do is inject ⟨ and ⟩ symbols, which correspond to calls and returns, into the inferred automata. The NWA's stacking behavior can then enforce, when reading a ⟩, that the return site matches the call site.

The example code in Listing 5.5 benefits from this increase in precision. The NWA's constraints on the return transitions from the exit node of `outputInt` to each of the two return sites restricts the data flow, and as a result the producer's language is inferred to be ⟨ `int` ⟩ | `char` ⟨ `int` ⟩ `char`. The NWA version of PCCA reports that the two components are compatible.

Using NWAs is somewhat similar to using context-free-language reachability techniques [71]. CFL reachability techniques mark each call/return edge pair with a distinct set of matched parentheses; possible executions of the program correspond only to strings with matched parentheses. The dataflow problem can be formulated so that only flows along such well-matched paths are considered. Our use of NWAs closely mirrors this approach for the producer.[4]

There is a problem with using NWAs in this way, however. Strings of the automata are supposed to correspond to (formats of) messages that the programs can write or read, but the injected ⟨ and ⟩ symbols do not appear in the message format. The presence of these symbols essentially puts artificial constraints on the producer and consumer that are easy to violate, in which case we would falsely report an incompatibility. To work around this problem, we apply an operation we call Enrich, which is described in Section 5.3.

Unfortunately, the context-sensitivity benefit only applies to the producer's model: the enrich operation we do to the consumer essentially makes a regular approximation out of the original. We have not investigated applying the ideas of cloning to obtain increased precision, although we think it would be possible.

In other words, using NWAs to model the components provides a way to obtain a context-sensitive analysis in one of the components without the problems of cloning.

[4]CFL-reachability distinguishes acceptable return edges from unacceptable ones by whether the brackets match; our NWAs distinguish them by whether the corresponding call site is on the NWA's stack.

```
1  outputInt()
2    writeInt();
3  producerMain()
4    if ... then
5      outputInt();
6    else
7      writeChar(); outputInt(); writeChar();
8  inputInt1()
9    readInt();
10 inputInt2()
11   readInt();
12 consumerMain()
13   if ... then
14     inputInt1();
15   else
16     readChar(); inputInt2(); readChar();
```

Listing 5.5: Components that illustrate the benefits of NWAs

5.3 Enriching NWAs for Compatibility

This section applies to the NWA version of PCCA only.

It would be too restrictive to demand that the producer and consumer perform calls and returns at corresponding moments during their executions. The NWAs that we infer from the producer and consumer follow the same call/return behavior as the corresponding original programs; thus the strings in the languages of the producer and consumer models contain internal call and return symbols that are not actually present in the messages between components. Checking containment of the languages of the inferred models would require that the components agree in this respect.

Our running example (Listings 5.1 and 5.2) illustrates the issue. Each "packet" consists of a Boolean, optionally followed by a double and a Boolean. The producer sends the entire packet within one procedure (`sendReading`), but the consumer reads the first Boolean, and then calls another procedure (`updateReading`) to read the remaining values of the packet.

As a consequence of the components having different calling structure, the strings that correspond to a single message differ in the producer's model and the consumer's model.

Example 5.1. Consider the string `bool double bool`, emitted by Listing 5.1's code when the producer performs just one iteration—hence the string contains just a single packet. For the producer's NWA (Section 5.2), the string would be $\langle$ $\langle$ `bool double bool` $\rangle$ $\rangle$, while for the consumer's NWA (Section 5.2), the corresponding string would be `bool` $\langle$ `double bool` $\rangle$. These strings have $\langle$ and $\rangle$ in different locations.

To accommodate the different nesting structures, we "enrich" the consumer's NWA so that it can use nondeterminism to guess when the producer makes an internal call or return and insert the corresponding symbol into its own strings.

Example 5.2. For Example 5.1, the language of the consumer's enriched NWA contains not just `bool` $\langle$ `double bool` $\rangle$ but also $\langle$ $\langle$ `bool double bool` $\rangle$ $\rangle$. The latter string is in the languages of both the producer's NWA and the consumer's enriched NWA.

Example 5.3. For the buggy consumer in Listing 5.3, the original language contains strings such as `bool` $\langle$ `double bool` $\rangle$, but not, for instance `bool bool` (which is in the producer's NWA's language). Denote by C_e the NWA inferred for the buggy consumer. After enriching the consumer's language, `bool bool` will still not be in $\text{Enrich}(C_e)$; `bool bool` will be a counterexample to language containment.

If an analyst knows that both components use the same call/return structure, he can omit the enrichment step to obtain a more precise comparison of the two languages. Without the approximation caused by Enrich, a "compatible" result is more credible; however, if there is uncertainty in the call/return assumption, an "incompatible" result is less credible.

In essence, Enrich allows the consumer's NWA to emulate the call/return structure of the producer's NWA. Enrich is defined as follows:

Definition 5.4. Given NWA $A = (Q, \Sigma, q_0, \delta, F)$, augment δ with the following transitions:

1. For every state p, introduce a call transition $\delta_c(p, \langle, p)$.
2. For every pair of states (p, q), introduce a return transition $\delta_r(p, q, \rangle, p)$.
3. For every call transition $\delta_c(p, \langle, q)$ in the original NWA, introduce an ε-transition $\delta_i(p, \varepsilon, q)$.
4. For every return transition $\delta_r(p, p', \rangle, q)$ in the original NWA, introduce an ε-transition $\delta_i(p, \varepsilon, q)$.

Items 1 and 2 allow the consumer's enriched NWA to perform extra call or return moves to emulate the producer NWA, while Items 3 and 4 allow the consumer's enriched NWA to omit calls or returns, in case the producer has fewer.

Example 5.5. Examples 5.1 and 5.2 require all four steps: to match the producer, the consumer needs to add two calls to the beginning of the input string, add two matching returns to the end of the input string, and remove the "extra" call between the first "bool" and "double" and its corresponding return.

While in theory it is possible either to enrich the consumer to match the producer or to enrich the producer to match the consumer, in practice only the former is reasonable. The goal of the containment check is to determine the emptiness of $L(P) \setminus L(C)$. Enriching an NWA enlarges its language, so this operation adds some error E to one of the operands, resulting in either $(L(P) \cup E) \setminus L(C)$ or $L(P) \setminus (L(C) \cup E)$. Unfortunately, the error introduced by enriching the producer's NWA invariably leads to false positives: for the consumer to accept everything that the enriched producer emits, the consumer would have to accept every possible call structure of every string the producer emits.

What we would really like to have is a property like the following:

> Given a language L, define forget(L) to be the set of strings in L with all $\langle$ and $\rangle$ symbolss removed. Then forget$(L) =$ forget(Enrich(L)).

Unfortunately, it is not possible to have this property. If it did hold, it would be possible to determine language inclusion of context-free languages using a technique similar to that described in Section 2.1.3 and Fig. 2.5. The call and return edges would be labeled with $\langle$ and $\rangle$ respectively instead of ε. Given grammars for L_1 and L_2, to test whether $L_1 \subseteq L_2$, one would create NWAs N_1 and N_2 using the modified construction and then test whether $N_1 \cap \text{Enrich}(N_2) = \varnothing$. (The format of a program being broken up into different procedures based on implementation artifacts corresponds very closely to how the grammar for a language is structured, which is like implementation details of the grammar.) Thus the Enrich process does more than simply add and remove $\langle$ and $\rangle$s to the language, and performs more approximations. We think an intuitive way of looking at what happens is that Enrich creates a regular-like approximation of the original NWA's language.

5.4 Using PCCA for more than types

In this section, we describe how it is possible to use PCCA's format inference capabilities to infer more than just a model of a program's input or output language. We start off describing how to make the format models more accurate by incorporating some information about the programs' behaviors into the language of the model (without changing the actual communication format), and then discuss possible uses of PCCA's techniques for tasks that do not involve the format of messages at all.

We first return to the example in Listings 5.1 and 5.2 to illustrate how the programmer could improve the results of the analysis. We start by describing a bug that PCCA would not be able to find, and then explain how to modify the buggy code — but without changing the actual protocol — so that the bug *is* found.

Suppose that the specification of the protocol changed during development: the final `bool` field was not originally needed, but was added later. Suppose that the implementation of the producer *was* changed to emit this field, but the consumer was not. (In other words, Line 7 in Listing 5.1 was added at the time the specification changed. The consumer *should* have been changed to include Line 3 in Listing 5.2, but that line was erroneously omitted.)

This situation would almost certainly signify a bug, but it would *not* be detected by our tool. The reason is that there is no association between the procedure call on Lines 3 and 5 in the producer, which writes the first `bool` in each packet, and Line 8 in the consumer, which reads it. Instead, the consumer could "use" the call on `readBool` on Line 3 to consume the final field of the *previous* packet, then not call `updateReading` during that iteration.

We can modify the source code of the producer and consumer to make it possible for our technique to detect the previous bug. What prevents our technique from detecting this bug is that the elements that the producer and consumer thought were packets got out of sync. By inserting a "phony" I/O call at the start or end of each loop (e.g., in the ellipsis on Line 9 of the producer and between Lines 7 and 8 of the consumer), we can make the packet divisions visible to PCCA, allowing it to check that the producer's and consumer's packets cannot get out of sync.

The phony calls would have a type that does not appear in the packet itself; in our experiments we have called it `SEP`. The key point to realize is that this "type" does not have to have any material presence in any of the communications, and in fact the procedure that performs the phony I/O can be completely empty.

This idea can be generalized to "hijack" the compatibility algorithm to ensure that *events* that should occur during the execution of the producer and consumer occur in the proper order. From this point of view, a write operation is essentially an event, during which the fact that the program communicates is only incidental.

One potential application for PCCA that is entirely different from checking format compatibility is inspired by work by Srivastava et al. [78]. There are three independently-developed implementations of the Java libraries (i.e., the contents of the `java.` package): the official Java Development Kit originally developed originally by Sun, GNU Classpath, and a defunct implementation by the Apache project (Apache Harmony). Srivastava et al. analyzed the three libraries against each other to find differences in behavior in how the libraries invoked security-sensitive operations.

Figure 5.6 gives an example from Srivastava et al.'s paper. It illustrates a vulnerability in the Apache Harmony library that arises because there is a missing `securityManager` call near Line 8 of Fig. 5.6(b) before calling the native `impl.connect` procedure on Line 12. (The check is present on Line 11 of the Sun JDK, Fig. 5.6(a).) PCCA should be able to find this vulnerability, perhaps with some postprocessing. PCCA could infer a language along the lines of

$$(\texttt{checkMulticast} \mid \texttt{checkConnect}\ \texttt{checkAccept})\ \texttt{JNI-connect} \qquad \text{(JDK)}$$

for the Sun JDK version, and

$$(\texttt{checkMulticast} \mid \texttt{checkConnect})\ \texttt{JNI-connect} \qquad \text{(Harmony)}$$

for the Apache Harmony version. Duplicating Srivastava's results would not be a direct application of PCCA, especially if the goal were to match Srivastava et al.'s analysis as closely as possible. However, it seems that there could be potential for PCCA to be useful in such an area.

5.5 PCCA Implementation

We now describe some implementation details of PCCA. The version of PCCA described in this chapter works on C and C++ source code; it would be possible to use the XFA version and ignore invariants to obtain a PCCA for x86 code, but we have not explored this avenue.

```
1   // JDK
2   public void connect(InetAddress address, int port) {
3     ... connectInternal (address, port ); ...
4   }
5   private synchronized void connectInternal(InetAddress address, int port) {
6     ...
7     if (address.isMulticastAddress()) {
8       securityManager.checkMulticast(address);
9     } else {
10      securityManager.checkConnect(address.getHostAddress(), port);
11      securityManager.checkAccept(address.getHostAddress(), port);
12    }
13    if (oldImpl) {
14      connectState = ST_CONNECTED_NO_IMPL;
15    } else {
16      ... getImpl().connect(address, port); ...
17    }
18    connectedAddress = address;
19    connectedPort = port;
20    ...
21  }
```

(a) JDK implementation of `DatagramSocket.connect`

```
1   // Harmony
2   public void connect(InetAddress anAddr, int aPort) {
3     synchronized ( lock ) {
4       ...
5       if (anAddr.isMulticastAddress()) {
6         securityManager.checkMulticast(anAddr);
7       } else {
8         securityManager.checkConnect(anAddr.getHostName(), aPort);
9         // MISSING CALL to checkAccept
10       }
11      ...
12      impl.connect(anAddr, aPort);
13      ...
14      address = anAddr ;
15      port = aPort ;
16      ...
17    }
18  }
```

(b) Apache Harmony implementation of `DatagramSocket.connect`

Figure 5.6: Illustration of a missing security check (`securityManager.checkAccept`) in the Apache Harmony library [78, Fig. 2]. Srivastava et al.'s analysis is able to find this vulnerability.

PCCA has two phases: inference and compatibility. During the inference phase, PCCA uses CodeSurfer/C [21] to perform pointer analysis and build an interprocedural control-flow graph (ICFG) and call graph for each component. It traverses the ICFG to create a list of all call sites that (directly) call an I/O procedure (see Section 5.5.1), then traverses the call graph to determine which procedures to prune because they cannot perform I/O (see Section 5.5.2). It then traverses the ICFG again to create the automaton for each procedure as described in Section 5.2, determinizes and minimizes each of them, and combines them into our model of the program.

During the compatibility phase, PCCA reads the automaton produced for each component and proceeds with the compatibility check according to PCCA's mode (FA or NWA). The NWA mode of PCCA is implemented with OpenNWA (Section 2.2).

5.5.1 Seeding the System with I/O Procedures

PCCA requires information about (i) what procedure calls of the producer can perform output, and (ii) what procedure calls of the consumer can perform input. There are a number of ways such information can be supplied to PCCA:

1. The user can provide a list of I/O procedures (e.g. `readBoolean`, `writeInt`, as in the example) and their associated types. For calls to standard procedures, such as `puts`, PCCA is already equipped with such mappings.

2. For calls to `printf`- or `scanf`-style procedures, if the format string is a constant in the code, PCCA will parse the string to determine the types being operated on.

   The implementation is flexible enough so that the producer or consumer can contain user-defined procedures with `printf`/`scanf`-like format-strings, provided that the format-string syntax is either the same as what is used by `printf` or what is used by `scanf`. PCCA just needs to know the name of the procedure and which formal parameter holds the format string.

3. If all else fails, the user can supply comments that annotate procedure-call sites to specify that a particular call site performs either input or output. The annotation includes the type that is operated on. This method also allows the user to selectively choose only some call sites to a particular procedure.

4. Finally, the list of procedure-call sites that the tool should consider to be I/O procedures is explicitly materialized in a text file, so the user can add, remove, or change call sites in that list, or even generate it by different means. (In fact, in the current version of PCCA, the techniques described in Items 1 and 2 are implemented by one program, and the technique described in Item 3 is implemented by a second program.)

5.5.2 Removing Irrelevant Procedures

To reduce the size of the inferred NWA, PCCA prunes procedures that cannot possibly participate in I/O operations. If there is no path from the entry of procedure P to the exit of procedure P along which an I/O procedure can be invoked, P can be discounted entirely. One of the first steps of PCCA is to traverse the call graph generated by CodeSurfer, determine which procedures can transitively call an I/O procedure, and ignore all others. As illustrated in columns 3 and 4 of Table 5.7 (see Section 5.6), the effect of pruning is substantial, reducing the number of procedures by as much as 90%.

5.6 Experiments

To test the capabilities of PCCA, we ran it on a small corpus of examples (whose characteristics are listed in columns 2 and 3 of Table 5.7). The experiments were run on a system with dual quad-core, 2.27GHz Xeon E5520s processors; however, PCCA is entirely single-threaded. The system has 12 GB of memory, and runs Red Hat Enterprise Linux 5.

The experiments were designed to test whether PCCA would detect bugs in producer-consumer pairs that were buggy, correctly identify (presumably) correct code as having the language-containment property, and scale to realistic programs. We also compared the results between the FA and NWA-based modes of operation to determine whether the potential benefits discussed in Section 5.2.2 arose.

Each example consisted of a pair of programs—a producer and a consumer. In several cases, we used the program as both the producer and the consumer, which makes sense for programs that read and write the same format.

Test	LOC	#Funcs		$\lvert Q\rvert$	#I/O	Infer	NWA version (sec.)			FA version (sec.)		
		Orig.	Pruned			aut.	¬C	Total	OK?	¬C	Total	OK?
ex-prod	43	11	3	9	4	2.12	0.35	4.90	Y	0.10	4.76	Y
ex-cons	26	7	2	5	3	2.24						
ex-prod	43	11	3	9	4	2.12	0.16	4.49	N	0.10	4.61	N
ex-cons-fig3	25	7	2	5	3	2.09						
ex-prod-§2.5	43	11	3	10	5	2.29	0.70	4.87	N	0.10	5.09	N
ex-cons-§2.5	25	7	2	5	3	2.40						
gzip-prod	4396	100	17	51	25	26.3	123	177	N*	101	157	N*
gzip-cons	4396	100	24	71	50	27.8						
gzip-prod	4396	100	17	51	25	26.3	583	646	Y	102	156	Y
gzip-fix-cons	4389	100	24	73	51	27.8						
gzip-prod	4396	100	17	51	25	26.3	191	239	Y	187	235	Y
pigz-cons	5001	162	19	88	60	21.8						
bzip2-prod	5772	121	15	32	8	26.3	47.7	102	Y	47.1	101	Y
bzip2-cons	5772	121	13	29	10	27.4						
png2ico-prod	806	39	1	22	29	9.48	14.4	33.1	Y	0.16	10.1	Y
ico-spec-cons	n/a	n/a	n/a	26	28	n/a						

Table 5.7: The experiments. "LOC" is lines of code, "orig." is the number of procedures in the program, "pruned" is that number after pruning. $\lvert Q\rvert$ is the number of states in the inferred automaton (equal between the two variants). "# I/O" is the (static) number of calls to I/O procedures. "Infer aut." is the time (sec.) to produce the automata for every procedure in the program. (The output of this step is used for both the NWA and FA versions.) For both the NWA and FA version, ¬C is the time (sec.) to determinize and complement the automaton. (Determinizing each procedure's FA is not included in this time, but takes a negligible amount of time in all experiments.) "Total" is the end-to-end time for analysis, including the inference step. "OK?" reports the output of PCCA. "N*" marks an erroneous report of incompatibility.

The examples are as follows:[5]

- *ex-prod*/*ex-cons* make up our running example (stubs for the I/O procedures are included in the count),
- *ex-prod*/*ex-cons-fig3* uses the buggy version of the consumer presented in Listing 5.3,
- *ex-prod-§2.5*/*ex-cons-§2.5* are buggy versions of the running example, modified as described at the end of Section 5.4 with the separator to mark the packets,
- *gzip* and *bzip2* are the common Unix compression/decompression utilities,
- *gzip-fix-cons* uses a modified version of *gzip* (discussed below) to eliminate an erroneous report,
- *pigz-cons* is an alternative implementation of the gzip algorithm, designed to run in parallel, and
- *png2ico* is an image-conversion program, which we compare to a hand-written specification (*ico-spec-cons*).

Reported times are the median of 5 runs. The numbers for the FA version use NWAs with no call or return transitions. This gives an apples-to-apples comparison with NWAs, but is slower than an alternative implementation that converts each NWA to an OpenFST acceptor, determinizes with OpenFST, and converts back. All times are less than 1 sec. with the latter approach. There *is* an intrinsic cost to using an NWA representation, but we feel that most of the difference between our FA numbers and OpenFST's indicates room for improving the WALi implementation. (That would improve the NWA version as well.)

Three of the tests, *gzip*, *pigz*, and *png2ico*, required relatively minor modifications. *gzip* uses input and output operations much like those in our running example, except implemented as macros. Because PCCA uses the control-flow graph generated by CodeSurfer/C, these macros are not visible, so we replaced the macro definitions with procedures. In addition, *gzip* calls the procedure that actually performs the compression or decompression through a function pointer. CodeSurfer/C performs points-to analysis, but PCCA does not take such indirect calls into account; thus we modified the source to call the procedure

[5] Our experiments can be found at `http://www.cs.wisc.edu/wpis/examples/pcca/`

ID1	ID2	CM	FLG	MTIME	XFL	OS	···

ID1, ID2	Fixed constants; *gzip*'s "magic number"
CM	Compression algorithm
FLG	Flags, as a bitmap
MTIME	The modification time of the original file
XFL	Compression-method-specific flags
OS	ID of the OS where the file was compressed

Figure 5.8: The specification of *gzip*'s header format. Each field is 1 byte except for MTIME, which is 4.

directly. (This is not a fundamental limitation of our technique, though imprecise pointer analysis could lead to further imprecision.) Similarly, for *pigz* we had to change macros into functions and replace a call to the actual I/O routines that used `pthread_create` with a direct call. A final modification that applies in a similar manner to both *gzip* and *png2ico* will be described in their respective sections.

As shown in Table 5.7, PCCA reports that some commonly-used programs operate in a correct manner with regard to their I/O behavior, regardless of the automaton model used. PCCA also detects synthetic programming errors in small examples, as shown by the second pair of examples.

As can be seen in the results, the potential NWA benefits did not appear to affect the results of the analysis. (PCCA does report different results for the example in Section 5.2.2, but we do not include that experiment in Table 5.7.)

We also performed an informal experiment using the NWA version without Enrich (as mentioned at the end of Section 5.3). We tested programs that read and write trees in infix and prefix notation. Both the standard NWA version of PCCA and the no-Enrich version reported that the infix components are compatible with each other, that the prefix components are compatible with each other, and that each is incompatible with the other. As discussed in Section 5.3, the compatibility results are more credible for the no-Enrich version; the incompatibility results are more credible for the standard NWA version.

Omitting the Enrich step also dramatically decreased determinization time; even *gzip-fix-cons* could be determinized in less than one second. Thus, it might be beneficial to try to combine enrichment and determinization.

gzip The analysis of *gzip* reported a erroneous incompatibility in the distributed version; we examine the issues more closely here. For *gzip*, the compressed data itself appears as just a sequence of bytes, so the compatibility check essentially is testing the compatibility of the code that reads and writes the header and footer. Figure 5.8 describes the header format of a *gzip* file. The code that writes this header (in `zip.c`) corresponds very closely to the header format:

```
put_byte(GZIP_MAGIC[0]); /* magic header */
put_byte(GZIP_MAGIC[1]);
put_byte(DEFLATED);      /* compression method */
...
put_byte(flags);         /* general flags */
put_long(time_stamp);
...
put_byte((uch)deflate_flags); /* extra flags */
put_byte(OS_CODE);
```

For this code, PCCA infers the format specified in Fig. 5.8.

However, the code that *reads* the header is reported to be incompatible; this is a false positive. Unlike the output procedures, input is always done one byte at a time:

```
stamp  = (ulg)get_byte();
stamp |= ((ulg)get_byte()) << 8;
stamp |= ((ulg)get_byte()) << 16;
stamp |= ((ulg)get_byte()) << 24;
```

Because the consumer reads the `time_stamp` field as four individual `bytes` instead of one `long`, it appears incompatible. This is similar to the issue of granularity of types discussed in Section 5.2.

To address this, we replaced this code (and similar code that reads `long` fields in the footer) with a call to a new `get_long` procedure, and added `get_long` to the list of I/O functions. (The actual implementation of `get_long` does not matter, so it can perform the same four bytewise reads without changing the program's behavior.) In addition to helping PCCA, we feel that the modified code is cleaner: by having the code for reading and writing a `long` in one place, it is easier for the programmer to see that those procedures

agree, for instance by reading and writing the bytes in the same order. It should even be possible to use our techniques to perform this check as well, by giving different types to each byte in the `long`.)

After making this change, PCCA reports that the programs are compatible. It is unclear why there is such a dramatic difference between the time it takes to determinize each version of the consumer in the NWA version. The input NWAs are of almost identical size and makeup, but it appears that the extra `long` alphabet symbol in the revised version causes the determinized NWA to be much bigger (176 states vs. 27). (Note that neither of these automata are minimal; it could be that the extra size in the revised version could be reduced to be more in line with the original version.) The sizes of the two automata in the FA version are much closer.

png2ico For *png2ico*, we demonstrate a slightly different application of our techniques. Instead of comparing a producer to a consumer, we compare a producer to a manually-crafted specification acting as the consumer. This checks that the producer emits only messages that are allowed by the specification. In the case of *png2ico*, we see that the program indeed appears to conform to the specification.

We manually crafted an automaton that describes the format of an icon file [41] and used that as the consumer. For the ICO format, this was reasonably straightforward and took less than two hours. Despite the process of creating the NWA for the ICO specification being straightforward, during the specification's creation, PCCA reported incompatibility with *png2ico* a couple of times. When looking into why, I discovered that I had made minor errors in the specification.

The automaton allows PCCA to check header information, similar to *gzip* but with a much richer format. An icon file can hold several different images. In addition to a global header (that mainly says how many images there are), there is a directory that gives the offset and other information about each image and a header for the image data itself. We can check all of this, leaving only the raw image data itself appearing as a "meaningless" byte stream. (We cannot check that the image headers actually appear at the correct offsets, however.)

While most of the output from *png2ico* is performed through the procedures `WriteByte`, `WriteWord`, and `WriteDWord`, there are three places where a raw write is done using `fwrite`. Two of these locations write a sequence of raw bytes of an image to the file. We could

reasonably infer just the regular expression `byte*` for those calls (similar to the regular expression PCCA infers for the compressed data in *gzip*); however, we decided to put in a bit of extra effort to obtain higher confidence in the result. The two `fwrite` calls correspond to the "xor mask" and "and mask" of the bitmap. We manually specified that the first `fwrite` call outputs the type "xor mask" bytes and the second call outputs the type "and mask" bytes (an application of the technique described in Section 5.4), and required that each bitmap in the icon file contains a sequence of "xor mask" bytes followed by a sequence of "and mask" bytes. However, there is one call to `WriteByte` amongst those writing the "xor mask", so we had to manually change the type of that call to match that of the preceding `fwrite`. (We repeated the experiment but just used `byte*` for both masks, and PCCA still reported compatibility.)

The third call to `fwrite` is used instead of a sequence of four `WriteByte` calls; the reason the author chose this is not clear. We replaced this `fwrite` with the four individual `WriteByte` calls.

We only report the results for the version with specific types. The other variants we tried did not have much effect.

6 Adding Loop Counters With XFAs

The inference techniques described in the preceding chapter ignore values of variables within the programs. In this section, we give an overview of how we pay attention to certain data values to obtain a more precise format model.

Our goals are to determine two ways in which data values can affect the format:

- For consumers, we would like to find ways in which a data value read early in the input can affect the format of subsequent portions of the message.
- For producers, we would like to find ways in which a single data value is both written to an early part of the output and also affects the format that is written later.

For example, consider the two programs shown in Fig. 6.1. In the consumer, the value read in Line 1 determines part of the format to come — in this case, the number of `doubles` that are read later in the format. In the producer, the value of n that is passed as a parameter is both written in Line 1 and affects the number of `doubles` that are written later.

The two ways that data values can affect the format are conceptually duals of each other, but in terms of what goes on inside the program they are rather different. In the consumer, there is a flow dependence from the value read early in the message to the control structure that controls the later format.[1] For example, in Fig. 6.1(b), there is a flow dependence from Line 1 to Line 3. Such a flow dependence would not make sense for the producer, however — it would correspond to the producer having written out the earlier data value then immediately reading it back in. Instead, the same external information (in Fig. 6.1(a), the value of the parameter n) determines both the value of the field early in the message and the format of the later fields. For example, in Fig. 6.1(a), there is a flow dependence from the parameter n to Line 1 and another from the parameter n to Line 3, but there is no dependence (flow or otherwise) from Line 1 to Line 3.

[1] A *dependence* is a relation between two statements S1 and S2 that indicates that the two statements must appear in a particular order. S2 is *flow dependent* on S1 if S1 writes information that S2 reads. (Flow dependences are sometimes called read-after-write dependences, or true dependences.)

```
   produce(n, a):
1    writeInt(n)
2
3    for i := 1 to n
4      writeDouble(a[i])
```

(a) Example producer

```
   consume():
1    n := readInt()
2    a := new double[n]
3    for i := 1 to n
4      a[i] = readDouble()
5    return n,a
```

(b) Example consumer

Figure 6.1: Example producer and consumer

```
nrows = read_int();
ncols = read_int();
pixel ** image = malloc(sizeof(*image) * nrows);
for (int row=0; row<nrows; ++row) {
    image[row] = malloc(sizeof(**image) * ncols);
    for (int col=0; col<ncols; ++col) {
        image[row][col].red = read_byte();
        image[row][col].green = read_byte();
        image[row][col].blue = read_byte();
    }
}
```

Figure 6.2: Example program to read a simple image format

The two goals above look at the behavior of each program in a very procedural manner. By taking a more declarative perspective, we can unify the two goals. For both components, what we want to find is an invariant that the value written or read at one point in the message corresponds to a particular format in a different part of the message. In particular:

- We would like to infer and check compatibility of invariants that the value of one field in the message *equals the number of repetitions* of a different field or group of fields.

6.1 Overview

Consider the program shown in Fig. 6.2, which operates on a hypothetical, very simple image format. PCCA infers a format equivalent to the regular expression

```
int int ((byte byte byte)*)*.
```

The first `int` corresponds to reading the number of rows, and the second the number of columns. In our presentation, each of the two *s correspond to a program loop. The inner * corresponds to reading the pixels belonging to a single row, and always iterates the number of times specified by the second input `int`. The outer * corresponds to iterating over the rows, and always executes number of times specified by the first input `int`. Chapter 5's techniques are unable to capture these semantic constraints, while this chapter will describe how we can.

In a nutshell, our inference technique begins the same way as PCCA, building a regular expression where * occurrences correspond to program loops. (This is not how PCCA was described in the previous chapter, but it is a simplified view.) It will then determine equality invariants between the number of times each loop executes and values read at input points (or written at output points). When such an equality is detected, the * is replaced by a counted exponent referring to the input variable. In the case of Fig. 6.2, the regular expression given above will be transformed into

$$\texttt{r:int c:int ((byte byte byte)}^{\texttt{c}}\texttt{)}^{\texttt{r}} \tag{6.1}$$

(where `r` and `c` are dummy variables to which we give more descriptive names).

The expression in Equation (6.1) is never actually materialized; instead, an equivalent XFA is created.[2] However, the intuitive description of replacing a * with a counter that gets its value from another field is a useful way to think about extensions to the previous chapter's techniques.

Our approach finds equality invariants (for example, n equals the number of `doubles`) directly, without worrying about the actual data flow within the program. To do so, we first add instrumentation to the program and then look for invariants (in our case, equalities) between instrumentation variables. The instrumentation is described in detail in Chapter 4, but the key piece of instrumentation is the trip counter for each loop, which counts the number of times the loop executes.

In the example in Fig. 6.1, suppose that the trip count for the consumer's loop is k_c and the trip count for the producer's loop is k_p. Let n_c and n_p respectively be the I/O

[2]It is possible to formally define the extension to regular expressions used in Equation (6.1), an extension we call "message regular expressions" (MREs), which helps justify what we mean by "equivalent." However, because MREs are not used in the version of PCCA presented in this chapter, except in this intuitive description, we omit the formalities.

variables added because of Line 1 of the consumer and producer respectively. The invariant detection will find that $k_c = n_c$ and that $k_p = n_p$.

Once we have found invariants of this form, we incorporate them into the respective automata. We use XFAs, putting information about the values read and the trip counts into the XFAs' data values. Each XFA will only accept strings that respect the invariants found earlier. Strings that only label paths that ignore the invariants (e.g., 2 followed by 3 `doubles` for Fig. 6.1) will be rejected, in a somewhat analogous way to how NWAs can reject invalid paths (Section 5.2.2).

6.2 Inferring Format Models

Constructing an XFA model of a program's I/O format has three steps: build a model of the program's control flow (Section 6.2.1), we infer relationships between I/O values and loop trip counts (Section 6.2.2), and finally we incorporate the relationships into the XFA's data transformers (Section 6.2.3).

6.2.1 Modeling control flow

The control flow of the program is modeled by the state portion of the XFA. This is much the same as the technique discussed in the previous chapter. We begin by producing an interprocedural control flow graph (ICFG) for the program. For the most part, there is a 1-to-1 correspondence between nodes in the ICFG and states in the XFA, and between transitions in the ICFG and transitions in the XFA. Most transitions are labeled with ε because most programs only perform I/O at a few points.

The exception to the 1-to-1 correspondence comes into play when the subject program makes a call to one of a designated set of I/O procedures. In Chapter 5, we added a transition from the call site to the return site labeled with the type of the type of the value written or read (and did not have edges corresponding to the actual call or return). However, we need a way for information in the input to move into the XFA's data value (this point will be expanded near the end of Fig. 6.3). We do this by viewing each alphabet symbol as a sequence of bits, rather than an indivisible symbol. We insert the gadget shown in Fig. 6.3 at every node that corresponds to a call to an I/O procedure. For now, we only point out the structure of the states and transitions, and the data transformers will be explained

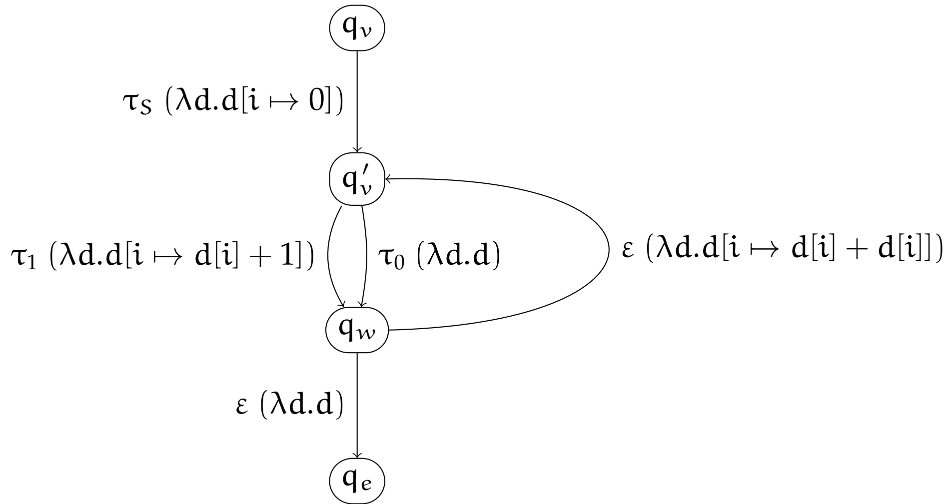


Figure 6.3: The XFA read gadget

in Section 6.2.3. For a given type τ, this gadget accepts a delimiter τ_S (so that bits from adjacent fields cannot blend together) and then a sequence of typed bits (τ_0 and τ_1).

For purposes of this section, we will assume that each I/O procedure call only reads or writes a single value, but this restriction is easily removed.

Let the ICFG be (V, E) of nodes and edges. Then we define the following components of the XFA as follows:

- $\Sigma = \{\tau_S, \tau_0, \tau_1 \mid \text{for each type } \tau\} \cup \{\$\}$
- $Q = \{q_v \mid v \in V\} \cup \{q'_v \mid v \text{ is a call to an IO procedure}\} \cup \{\textit{accept}\}$
- $QD_0 = \{e_{\textit{main}}\} \times D$ for the entry point of main (D is defined below) starting in any data value.
- $F = \{(accept, d) \mid d \in D = 0\}$

The state-transition relation δ and data-set transformers U are defined in Fig. 6.4. The data set is defined in Section 6.2.3.

CFG edge		Transitions	Data transformers
(v, e_f)	v call site to I/O func with variable i; w corresponding return	$(q_v, \tau_S, q_v{}')$ (q'_v, τ_0, q_w) (q'_v, τ_1, q_w) $(q_w, \varepsilon, q_v{}')$	$\lambda d.d[i \mapsto 0]$ id $\lambda d.d[i \mapsto d[i] + 1]$ $\lambda d.d[i \mapsto d[i] + d[i]]$
(v, w)	Edge goes from outside of a loop with trip count k to inside	(q_v, ε, q_w)	$\lambda d.d[k \mapsto 0]$
(v, w)	Edge is back edge in loop with trip counter k	(q_v, ε, q_w)	$\lambda d.d[k \mapsto d[k] + 1]$
(v, w)	Edge goes from inside of a loop with trip counter k to outside and there is an I/O relation $k = i$	(q_v, ε, q_w)	$\lambda d.$if $d[k] = d[i]$ then d else $\varnothing$)
(v, w)	Otherwise	(q_v, ε, q_w)	$\lambda d.d$
—	—	$(x_{main}, \$, acc)$	id

Figure 6.4: XFA state transition procedure δ and data relations U

6.2.2 Finding I/O relations

The goal of this step is to determine equality invariants between I/O values and loop trip counts. To do this, we instrument the program in question using Snotra and then find invariants with Daikon, as discussed in Chapter 4.

6.2.3 Modeling I/O relations

Now we would like to incorporate the I/O relationships found using Snotra and Daikon into the control flow model described in Section 6.2.1. For this, we will use the XFA's data set and transformers to enforce that the I/O relationships hold.

Let $N = \{0, 1, \cdots, n-1\}$ for some integer n. This set will be the domain over which the variables count; specifically, the equalities found by Snotra and Daikon will be enforced modulo n. Let $I = \{i_0, i_1, \cdots\}$ be the I/O variables found to participate in an equality, and $K = \{k_0, k_1, \cdots\}$ be the trip counters found to participate in an equality. For simplicity, assume that every trip counter is associated with at most one I/O variable and vice versa, so that we can say $i_0 = k_0$, $i_1 = k_1$, etc.

We will refer to both the instrumentation variables in Snotra's instrumentation (Section 4.2.2) as well as in the XFA's variable set as "trip counters" and "I/O variables", but

make no mistake: they are different. The variables in the instrumented version of the program *observe*, and provide a means for Daikon's back end to find emperically-supported invariants. The corresponding variables in the XFA's data set are used to *enforce* that the observed invariants actually hold in a particular run of the automaton.

An individual data value d will map each variable to a value, so the data set is the set of functions $D = (V \to N)$. (However, remember that a nondeterministic XFA can have several such valuations in hand at once.) We will write $d[i]$ for the value of variable i in d, and $d[i \mapsto a]$ to denote the function that acts like d except that i is mapped to a.

Each transformer $U(t)$ is a subset of $(V \to N) \times (V \to N)$. All of the transformers in the original, nondeterministic XFA will have deterministic data transformers, but they can become nondeterministic through automaton operations. Thus to simplify the notation in Fig. 6.4, we treat $U(t)$ as a function $D \to D$ rather than a relation, with one exception: it is possible for a data value d to have no successor, which we will denote by $\varnothing$. (In the relation view, this says that there is no d' for which $(d, d') \in U(t)$.)

Most transitions in the XFA are simply the identity relation; that is, $j' = j$ for all $j \in I \cup K$. The exceptions are for read statements and loops. For a read statement with an instrumentation variable i, the τ_S transition initializes i to 0, the τ_0 and τ_1 transitions increase i by 0 and 1, respectively (thus simulating reading from the most significant bit to the least significant) and the ε back edge doubles the value because another bit will be read. For a loop with instrumentation variable k, incoming transitions set k to 0, back edges increment k, and outgoing edges require that k equals the corresponding input variable. (The latter requirement is enforced by not providing any reachable next data value when the condition does not hold. This feature also means that the XFA cannot always determine the next data value of k by only looking at the current value, as the variables are not always independent.)

BDD variable order. Recall from Section 3.7.1 that the performance of BDDs is sensitive to the variable order. The variable order we use interleaves any variables that are compared at a loop exit. For example, if we have checks that $i_0 = k_0$, $i_1 = k_1$, etc., then the variables for i_0 and k_0 will be interleaved with each other and the variables for i_1 and k_1 will be interleaved with each other, but the two groups will be concatenated.

As described in Section 3.5.1, the first step of determinization multiplies the XFAs' weights (using $\odot$) with relations that represent the possible state transitions (e.g., $\{p \to q\}$).

Because the Q portion of the new $D \times Q$ data value needs to be stored in BDDs as well, we also need BDD variables for the current state. These variables are put at the beginning of the BDD. This is not necessarily an ideal location, because the value of the current state is not necessarily unrelated to the value of the counter variables, but it may not be possible to order the variables so that BDDs remain small in practice for this application. (We also tried the current state variables at the end of the BDD, as well as interleaving all variables.)

Finally, each logical variable is really two variables: the first holds the value in the left element of each pair, and the second holds the value in the right element of each pair. We denote the righthand variables with primed names.

Suppose that we have an XFA with the following characteristics:

- There are two I/O variables i_0 and i_1
- There are three loop trip counters k_0, k_1, and k_2
- There are two invariants $i_0 = k_0$ and $i_1 = k_1 = k_2$
- The current state variable is q
- All variables are 2 bits

then the variable order will be as follows (using $[x]_i$ for the ith bit of x):

$$[q]_0\,[q']_0\,[q]_1\,[q']_1\,[i_0]_0\,[k_0]_0\,[i_0']_0\,[k_0']_0\,[i_0]_1\,[k_0]_1\,[i_0']_1\,[k_0']_1$$
$$[i_1]_0\,[k_1]_0\,[k_2]_0\,[i_1']_0\,[k_1']_0\,[k_2']_0\,[i_1]_1\,[k_1]_1\,[k_2]_1\,[i_1']_1\,[k_1']_1\,[k_2']_1.$$

Why is the read gadget necessary? The fact that we need something like the read gadget, and need to break up each input symbol into its constituent bits, may seem a bit strange. To motivate the need for read gadgets, consider what would happen without them. We could have a transition labeled with a type like `int`. However, `int` is not *actually* an input symbol, only an abstract alphabet symbol; the concrete symbols are things like `0`, `1`, `42`, etc. Furthermore, suppose that the `int` transition is associated with an I/O variable i; then the result of reading the integer must be that i takes on (only) the concrete value. In the XFA formalism, being able to set i to any of m different values requires m different transitions, because each has to do something different to the data value. This means that the number

of transitions corresponding to a read would be exponential in the number of bits, and we would lose the benefits of thinking about the input in terms of abstract symbols.

Reading bit-by-bit means that we can progressively modify the data value in the way that is specified by the transformers in Figs. 6.3 and 6.4, and we keep the read gadget small.

Further, it is possibe to think about what the read gadget accomplishes in terms of a string homomorphism. A string homomorphism is a function $f : \Sigma^* \to \Sigma^*$ that replaces each symbol with a string. Thinking in terms of the concrete messages, a homomorphism could replace, for instance, 5 by $\texttt{int}_S\texttt{int}_1\texttt{int}_0\texttt{int}_1$, 11 by $\texttt{int}_S\texttt{int}_1\texttt{int}_0\texttt{int}_1\texttt{int}_1$, etc. The image of a regular language under a homomorphism is still regular, and while we are not quite doing the same thing, thinking about our technique in terms of homomorphisms may shed some light on what is going on.

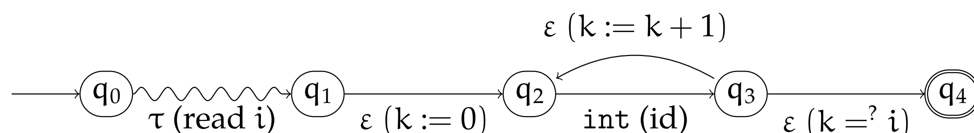
Readers who are used to thinking in program-analysis terms may wonder: if any value can arise as the net result of a read gadget, why does the read gadget buy anything at all? Why is it not just equivalent to a transformer that sets the corresponding I/O variable to unknown? The reason is that each possible value arises on a different string, and the analysis is sufficiently path sensitive to not combine the values for different strings.

6.3 Optimizations

This section describes two additional optimizations to the XFA compatibility process that we investigated.

6.3.1 Setting killed variables to a single value

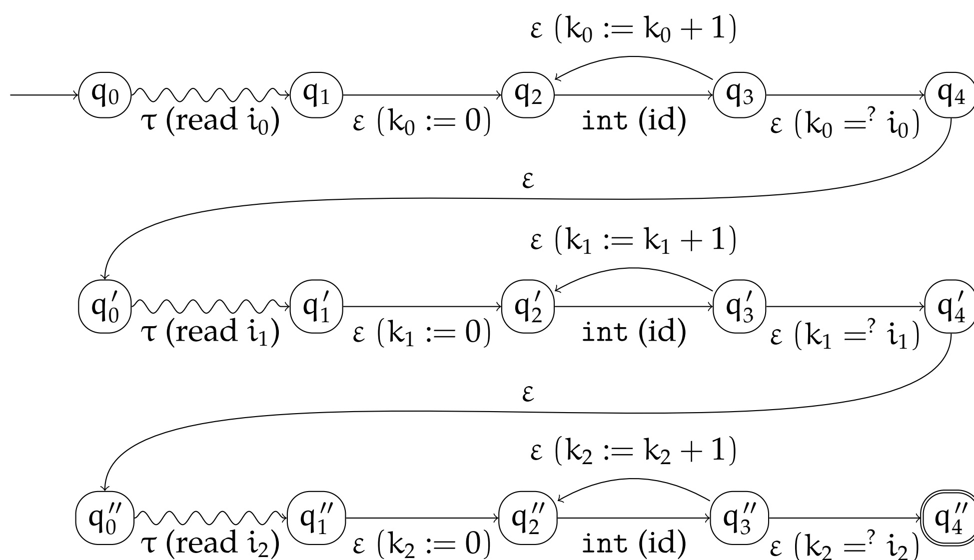
Consider the following XFA:



The XFA inclusion test (Section 3.6) computes the set of weights (or, for the antichains version, maximal or minimal weights) that it is possible to reach each state with. Because it is possible for the loop to execute any number of times, and each execution will have a

distinct weight (a transformer that sets the trip counter to the number of executions; in the antichains algorithm, these weights are incomparable and thus will all have to be stored), the number of weights that will be computed and stored for nodes within the loop (e.g., q_ι) is exponential in the number of bits we are interested in. (That is, it is linear in $|N|$.) Unfortunately, this is not just an upper bound; that many nodes really will need to be created.

However, there *is* an aspect that we can improve. The exit condition of the loop is $i = k$, and any value of i can arise. (This is always the case for the way we build XFAs, because for any value there is a sequence of τ_1 and τ_0 bits that will result in i taking that value via the read gadget.) The exit constraint cuts down on some weights — e.g., ones that set i to 2 and k to 3 — but there are still an exponential number of them (in the bit size). Now suppose that we concatenate several of the above automata in a row, as in the following:



In this version, the exponential effect multiplies. After the first loop, there will be 2^n possible weights. After the second loop, there will be $2^n \cdot 2^n$, because each assignment to i_0 and k_0 gets paired with each assignment of i_1 and k_1. In general, after m loops, there will be $(2^n)^m$ possible weights generated at the end.

We are being rather silly by tracking all this information. After all, the value of i_0 and k_0 will not have any effect after the end of the first loop. On the exit transition from the first loop, we can take the extra step of setting i_0 and k_0 to 0. (Any value would do of course;

there are other choices as well.) There will still be 2^n possible weights generated at q_{h1}, but only 1 weight at q_{e1}. Thus there will be 2^n weights at q_{h2}, but only 1 weight at q_{e2}. For this example, we have removed an exponential factor.

The experimental results presented in Section 6.5 bear out the analysis above: this optimization speeds up this example by an exponential factor.

How do we actually perform this optimization? In fact, figuring out whether a logical variable can affect the weight in the future is performing a well-known analysis called *live-variable analysis* (see, e.g., Aho [1, §9.2.5]). Live-variable analysis determines what variables are live at each program point. A variable is *live* if it can be referenced later before being overwritten. (In our construction, if it can be referenced, then it can affect the weight.)

Suppose that we are given an XFA. We define five functions all of type $\delta \to \mathcal{P}(V)$:

$$gen(t) = \{x \in V \mid \text{the subexpression } \cdot [x] \text{ appears in the transformer for t in Fig. 6.4}\}$$

$$kill(t) = \{x \in V \mid \text{the subexpression } \cdot [x \mapsto \cdot] \text{ appears in the transformer for t in Fig. 6.4}\}$$

$$live_{in}(t) = gen(t) \cup (live_{out}(t) - kill(t))$$

$$live_{out}(t) = \bigcup_{t' \in succ(t)} live_{in}(t')$$

$$killed\text{-}by(t) = live_{in}(t) - live_{out}(t)$$

Note that the first two procedures are closed form, while $live_{in}$ and $live_{out}$ are defined in terms of each other. By finding the minimal solution to these equations, we gain the information we need.[3]

For each transition t, $killed\text{-}by(t)$ tells us the variables that t's transformer needs access to, but that will never be referenced again in the future. We modify the transformer so that, after performing its usual execution, all variables in $killed\text{-}by(t)$ are set to 0. This can be done by extending the existing transformer with $\bigotimes_{x \in killed\text{-}by(t)} \lambda d.d[x \mapsto 0]$.

6.3.2 Collapsing ε sequences

Another optimization that we tried that does not appear to have a significant effect is to collapse sequences of ε transitions. Suppose that we have a sequence of ε transitions

[3]Finding the minimial solution can be done using standard techniques. We do it by treating the standard technique as a semiring, converting the XFA into a WPDS, and performing a pre*query.

$(q_0, \sigma, q_1)(q_1, \varepsilon, q_2)(q_2, \varepsilon, q_3) \cdots (q_{n-1}, \varepsilon, q_n)$ where (q_1, ε, q_2) is the only outgoing transition from q_1, (q_2, ε, q_3) is the only outgoing transition from q_2, etc. We collapse this sequence by removing all of the listed transitions and inserting a new transition (q_0, σ, q_n). The weight of the transition is $U(q_0, \sigma, q_1) \otimes \bigotimes_{i=1}^{n-1} U(q_i, \varepsilon, q_{i+1})$. This transformation trivially leaves the automaton's language unchanged (it represents the same formal power series).

The reason this optimization might be expected to improve the runtime is that it could help prevent an analysis from repeatedly having to propagate information down the chain of states. This would not help when we use FWPDSs as the backing algorithm, but could potentially help with normal WPDSs.

At the same time, it is simpler and less computationally-intensive than a full ε-removal process. The reason is that there can be no cycles that the algorithm has to deal with, because cycles are split at the loop head, which has at least two outgoing transitions. (One exception is the pathological case that there is an ε cycle with no exit.) We conjecture that cycles are where most of the computational effort comes in, because the weight has to "climb up the lattace" until it reaches a fixed point no matter what algorithm is used.

6.4 Tradeoffs with our instrumentation strategy

We now discuss what forms of semantic constraints can be captured by the instrumentation strategy we follow. The information that is kept by the XFA's data state and by the instrumentation that Snotra inserts into the target program is finite. At any point, it only stores the latest I/O value that was written or read, and it only stores the trip count of the latest time each loop is executed. In some cases, this is not sufficient to capture the semantic constraints within a format.

The ICO format discussed in Chapter 1 and Fig. 1.1 provides an example of how our approach can fall short: the height and width of each image is stored in each image's icon directory, but all ⟨*ico-directory*⟩s come at the start of each file. A program that writes or reads ICO files would almost certainly have a single static occurrence of the call that writes/reads the ⟨*ico-directory*⟩'s height field — but the instrumentation for that call would overwrite the height of the first image when it reads the height of the second, and it would overwrite the height of the second when it reads the height of the third.

I have a plan (which I call "progressive limiting") that could mitigate the effects of this limitation, but because of the performance shortcomings we have experienced I have not explored how well it behaves. Progressive limiting would limit certain looping variables to 1 (so that "the first call" is "the only call") and explore what new constraints arise under that limitation. Progressive limiting could be implemented automatically.

Consider the ICO format. Suppose we had a large test suite that includes ICO files with multiple images. It is likely that the only semantic constraint that would be inferred is that the count field in the header, i_h, matches the number of ⟨*ico-directory*⟩s, k_d, and the number of ⟨*ico-image*⟩s, k_i. For the reasons discussed above, it would *not* be able to infer anything about the height or width fields, the number of color planes, etc. Progressive limiting would see the constraint $i_h = k_d = k_i$, and would find all the test cases where all three values were 1. It would then re-run the inference process using just those test cases. Because there would only be one directory and image in each of those tests, the next round *would* be able to find relationships between the height, width, etc. when they exist.

6.5 Experiments

We conducted several experiments using the XFA version of PCCA. The language-inclusion tests did not perform well enough to do a straight comparison of the producer's model with the consumer's, but they did allow us to try an experiment using several runs to test inclusion of the automata that contain a subset of the variables in each. (We call such a subset a variable *group*.) This experiment has two purposes. First, it was designed to determine whether the extra repetition counters have the potential to gain precision over the NWA and FA versions of PCCA — if iterations of one loop could "overlap" with the iterations of another loop (similar to the problem described at the start of Section 5.4), then doing a full run would not likely be significantly better than just using ordinary FAs.

Second, running variable-group against variable-group provides an alternative, lower-cost way of getting an answer. It would be nicer if we could run the full version with all variables, but it still provides a potentially better answer than the NWA and FA versions of PCCA. The way PCCA would be used in this manner is as follows. Suppose that the producer's model has m variable groups and the consumer's model has n groups. The user would perform $m \cdot n$ containment checks, comparing every producer variable group

with every consumer variable group. If there is some consumer variable group G for which *every* containment check produces an "incompatible" result, then the overall answer should be considered "incompatible", otherwise the overall answer should be considered "compatible." Furthermore, as explained in Section 6.5.1, an ideal result would be that every consumer group has exactly one producer group where the language is contained; the closer to this ideal a specific result is, the more confidence the user should have in a "compatible" answer.

The experiments described in this section should be considered preliminary, and there are a number of improvements that could be made to their results. (We had the idea of multiple runs each with single variables late in development, and so have not had time to explore the consequences.)

6.5.1 XFA benefits evaluation

We performed an evaluation designed to test whether the repetition counters tracked by the XFA models provide a benefit over the FA and NWA models built by the original version of PCCA. All tests described in this section were carried out with two bits for each instrumentation variable, but some of the tests have been run with other numbers of bits and the results match. (See Section 6.5.3 for more information on the scaling tests we have performed.)

ICO specification versus ICO specification

To do this, we compared a hand-written XFA specification of the ICO format (modified from the one used in Section 5.6) against *itself*. This may seem like a very strange and not very useful experimental choice, but it provides a measure of evaluation in the following sense. If it came out that the spec-spec comparison was *not* capable of distinguishing between models when the extra repetition information was present (for example, because it was possible for iterations of one loop to "blend into" another loop, similar to the confusion between packets that is discussed in Section 5.4), it would present a significant obstacle to the claim that XFAs are beneficial. Fortunately, this is not the case, and the results are nearly perfect; in other words, tracking the repetition counts really does refine the automaton's *language* as opposed to just affecting the automaton.

As mentioned above, we created an XFA version of the ICO specification discussed in Section 5.6. The specification has four sets of variables,[4] which we refer to as *A* through *D*:

Group *A* tracks the number of actual images within the icon file. (Recall that an icon file can contain multiple images, for example at different resolutions.) It consists of three variables: one to hold the value of the *count* field in the ICO header, one to track the loop count of the loop that reads icon directories, and one to track the loop count of the loop that reads the actual images.

Group *B* tracks the number of colors in an image. It consists of two variables: one to hold the value of the *colors* field in the icon directory, and one to track the loop count of the loop that reads the color palette in the image.

Group *C* tracks the number of rows in an image. It consists of *three* variables: one to hold the value of the *height* field in the icon directory, one to track the loop count of the loop that reads each row in the *or* mask, and one to track the loop count of the loop that reads each row in the *and* mask.

Group *D* is analogous to group *C*, and tracks the number of columns in the image. It contains an analogous set of three variables.

From the variables, we looked at six configurations: for each variable set, a version of the specification containing just that variable set and not tracking the others (referred to as *ico-spec-A* through *ico-spec-D*), a version with no variables (referred to as *ico-spec-none*, which is equivalent to the specification in Section 5.6), and a version that tracks all four sets of variables (*ico-spec-all*).

[4]Note that the choices made in the above specification are not the only possibilities for how to write a program. For instance, the specification conceptually treats each actual image inside the icon file as two images: the *or* mask and the *and* mask. It would be possible (though slightly harder) to write a program that reads each image is if is one larger image with twice the height. What would happen when using such a program as the producer? If h is the value of the height field, k is the trip counter for the loop that reads the rows of the "double-height" image, Snotra/Daikon would likely produce an invariant that says that $k = 2h$. However, this invariant would be ignored during construction of the producer XFA, and the format would, in regular-expression terms, have an uncounted $*$ operator. Checking compatibility with our current implementation (if it finished) versus the specification would probably result in an incompatibility. As discussed in Chapter 8 as a potential future direction, the types of invariants that we could enforce in the XFA is much greater than the equalities that we do now, and linear equalities (like $k = 2h$) is a good example of a type of invariant that I think would actually be quite easy to enforce without adding substantially more to the runtime than pure equalities.

We also tried adding separator characters (in the style of Section 5.4). The locations considered for separators were:

- At the beginning of each icon directory,
- At the beginning of each actual image,
- At the beginning of each color palette entry,
- At the beginning of each row, and
- At the beginning of each pixel.

Note that while these locations are at the beginning of loops that could plausibly be counted by logical variables, where separators are added and what loops are counted are orthogonal issues. For the separators, we looked at the following configurations:

- Separators present in all locations, with a unique separator type for each location,
- Separators present in all locations, but with a single type used throughout,
- Just a row separator,
- Just a column separator,
- Both row and column separators of different types, and
- Row separators.

(We did not try just an image separator, just a color separator, or just a directory separator because the only difficulty seemed to be a blending of the rows and/or pixels within a row. We anticipate those configurations would produce the same results as no separators.)

For each configuration of separators, we checked language containment of every specification version acting as a producer with every version acting as a consumer; the results are given in Table 6.5.

There were three different results; two are shown in the tables and the third is a one-cell difference from Table 6.5(b). In the following situations, we got the ideal results (discussed further below and shown in Table 6.5(a)):

- Separators present in all locations, with a unique separator type for each location,
- Separators present in all locations, but with a single type used throughout,

ICO specification as consumer (rows: ICO spec. as producer)

	-none	-A	-B	-C	-D	-all
-none	✓	✗	✗	✗	✗	✗
-A	✓	✓	✗	✗	✗	✗
-B	✓	✗	✓	✗	✗	✗
-C	✓	✗	✗	✓	✗	✗
-D	✓	✗	✗	✗	✓	✗
-all	✓	✓	✓	✓	✓	✓

(a) The ideal results, achieved if no loop can blend with another loop and our loop repetition counts in the XFAs actually gain additional information over FAs. These results are achieved when there is a separator annotation between each row of the image, including when there is universal separator used everywhere.

ICO specification as consumer (rows: Producer)

	-none	-A	-B	-C	-D	-all
-none	✓	✗	✗	✗	✗*	✗
-A	✓	✓	✗	✗	✓	✗
-B	✓	✗	✓	✗	✓	✗
-C	✓	✗	✗	✓	✓	✗
-D	✓	✗	✗	✗	✓	✗
-all	✓	✓	✓	✓	✓	✓

(b) Results when there is no separator annotation. In addition, the cell marked with a ✗* is incorrectly marked as ✓when a separator is added between pixels within a row but not between rows. Note that even here (with the exception of the last case), if the columns were scrambled and the correspondence with the rows hidden, it would still be possible to determine the correspondence.

Table 6.5: XFA ICO specification vs. specification experiments. ✓ means the combination was reported as compatible, and ✗ means the combination was reported as incompatible.

- Just row separators, and
- Both row and column separators of different types.

We got good, but not ideal, results — shown in Table 6.5(b) — when there are no separators, which corresponds to when there are no annotations in the programs. In this case, there are some non-ideal answers, but there is still potentially enough information to determine that the inputs are compatible without a priori knowledge that the automata were the same or what variables in the producer correspond to what variables in the consumer.

We got OK results when there is a column separator but no row separator; these results are shown in Table 6.5(b) except that the cell marked ✗* was reported as ✓ instead. We lose the ability to determine completely what variables correspond, as the *ico-spec-none* column has the same behavior as the *ico-spec-D* column. However, the other three groups of variables still have perfect results, as does Group *D* on the producer's side. (The ✗* cell is a strange result, and I do not have an explanation for it at this time.)

The results when using separators (Table 6.5(b)) are "ideal" in the sense that it shows that the additional information tracked by the XFA is able to make meaningful distinctions for every tracked loop. Even in the case that arose when no separator annotations were

added, a problem analogous to that discussed in Section 5.4 was largely *avoided*, and the iterations of different loops can be distinguished.

Performance for every test that did not involve an *ico-spec-all* configuration on either the producer or consumer's end was well under a second to perform the containment check. When using *ico-spec-all* as a producer but not as a consumer (the bottom row), times ranged from roughly 10–25 seconds; when using *ico-spec-all* as a consumer but not a producer (the rightmost column), times ranged from roughly 20–50 seconds); when using *ico-spec-all* as both producer and consumer (the bottom-right corner), times ranged from roughly 30 seconds to 2 minutes. Section 6.5.3 gives more information for how certain configurations scale with the number of bits.

Inferred *png2ico* XFA versus inferred *png2ico* XFA

We performed a similar test as that described in the previous subsection, but using automata inferred from *png2ico*. To perform these tests, we generated traces for *png2ico* with two kinds of tests:

- We converted 20 PNG files to ICO files containing just a single image.
- We converted a single PNG file to ICO files containing multiple copies of that image; the number varied between one and five copies.

Following the generation of traces, we generated invariants and then an XFA model for each variable group. The PNG files were selected arbitrarily from files we had lying around on our system. *png2ico* is not able to convert every image to an icon: the ICO format itself limits the image size to 256×256, and *png2ico* additionally limits the size and width to be multiples of 8; our tests were done with successful runs. (Discarding other runs should not actually affect the results, and was done for expediency.) The runs where *png2ico* was given multiple images are not ideal in the sense that it would have been better to use different images to better match realistic usage. Our use of multiple counts of a single image was designed to give results similar to what progressive limiting would likely be able to achieve (see Section 6.4), but using what we had implemented and with a single run.

We modified *png2ico* to replace a call to *fwrite* with a loop that wrote each byte. This is a mechanical replacement that could be done automatically. We also tried a version with separators — for this, we added a call to a separator function at the start of each loop in the

portion of the program that performs I/O; these locations are similar to those in which separators were added in the ICO specification. (We only tried a version with separate types for each separator.)

To determine variable groups, we put them into partitions based on what variables are compared at a loop exit. If two variables are compared, they are put into the same group. For both the version with and without separators, there were four variable groups inferred:

Group α was created because of a "spurious" invariant. There is a loop in the program that outputs some padding at the end of each row of the image (in particular, the *or* mask), but this loop is never actually run, most likely because of the separately imposed constraint that the dimensions of the image are multiples of 8. Consequently Snotra/Daikon picks up an invariant that this loop is always run 0 times. It also picks up an invariant that the first byte of output (part of PNG's magic number) is 0. Because the trip counter and first byte of output are implied by the invariants to equal, PCCA tracks both variables; but PCCA is not smart enough to make sure that both of them are actually 0. It would be possible to add support for checking values against constants, which would actually be faster at runtime than checking them against each other.

Group β consists of an I/O variable for the count field, a trip counter for the loop that reads icon directories, and a counter for the loop that reads images. This is essentially the same as Group *A* of the ICO specification, but inferred by Snotra/Daikon.

Group γ consists of an I/O variable for the height field of the directory entry, a trip counter for the loop that reads rows in the *or* mask, and a trip counter for the loop that reads rows in the *and* mask. This is the same as Group *C* of the ICO specification, but inferred.

Group δ consists of an I/O variable for the width field of the actual image and a trip counter for the loop that reads columns in the *or* mask. (This loop was one of the ones introduced as a replacement for `fwrite`.) There are two differences between this group and Group *D* of the ICO specification. First, it used a different location for the width field. As mentioned in Footnote 4, the height and width of each image appear in two different locations, and PCCA used a different one. (PCCA does not insert checks for invariants between two I/O variables, although it would certainly

png2ico as consumer

png2ico as producer	-none	-α	-β	-γ	-δ
-none	✓	✓	✗	✗	✓
-A	✓	✓	✗	✓	✓
-B	✓	✓	✓	✓	✓
-C	✓	✓	✗	✓	✓
-D	✓	✓	✗	✓	✓

(a) *png2ico* as producer and consumer, no separators.

png2ico as consumer

png2ico as producer	-none	-α	-β	-γ	-δ
-none	✓	✗	✗	✗	✗
-A	✓	✓	✗	✗	✗
-B	✓	✗	✓	✗	✗
-C	✓	✗	✗	✓	✗
-D	✓	✗	✗	✗	✓

(b) *png2ico* as producer and consumer, with separators.

Table 6.6: *png2ico* vs *png2ico* experiments. ✓ means the combination was reported as compatible, and ✗ means the combination was reported as incompatible.

be possible for extra runtime cost.) Second, Snotra/Daikon did not infer an invariant with the trip counter for the loop in the *and* mask. The *and* mask data may be encoded with run-length encoding (RLE), and this would mean that the number of columns in the *and* mask actually does not need to match the width field. In that sense, the fact that Group δ does not match Group *D* is really a deficiency of the specification rather than the inference process. It is not clear how to make PCCA support encoding methods like RLE in a generic fashion.

In addition to the caveats with Groups α and δ given above, Snotra/Daikon did not infer equality invariants between the count field and the number of iterations of the color palette loop (Group *B* of the ICO specification). I did not have time to investigate that issue further.

Table 6.6 reports the compatibility results for the *png2ico* tests. As can be seen, the version with the separators gives the ideal results ("ideal" is explained in the previous subsection). The version without the separator does not work as well — there are a lot of "compatible" answers that we would like to see as "incompatible." It is not clear why that is.

6.5.2 Synthetic performance-scaling evaluation

To see how the runtime is affected by the number of bits used for each logical variable, we ran the XFA version of PCCA on the synthetic benchmark described in Section 6.3.1, looking at the effect of the number of loops concatenated together, the number of bits dedicated to each logical variable, and the optimizations described in the previous section.

We look at the impact of all three variables. The number of bits was varied between 1 and 7. The number of cycles was varied between 1 and 7, and for very low numbers of bits, in powers of 2 up to 256 cycles. (The following subsection discusses scaling for the ICO specification.)

Experiments in this section were run on an dual quad-core Intel Xeon 2.27 GHz processor with 12 GB of RAM, but the XFA version of PCCA is single-threaded. For a BDD implementation we used version 2.4 of the BuDDy library [50]. The process was configured so that approximately 8 GB is spent in total on a BuDDy node table with 60 million entries and a BuDDy operator cache with 15 million entries; the time to initialize these tables, roughly three to five seconds, is discounted in all benchmarks. (The timed portion of the benchmarks consists of reading in the automata, constructing the transformers, performing the enabled optimizations, and performing the inclusion test.)

First, we look at the effect on performance of the number of cycles (along with the optimizations). Figure 6.7 shows, when 3 bits are used, the effects of the number of cycles. (3 bits was chosen because it is high enough that we see an interesting climb in the unoptimized version, but low enough that it does not essentially go from "very low time" to timeout.) Figure 6.8 shows what happens with just 1 bit.

As can be seen from the figures, there are two behaviors. For the versions that use the reset-killed-variables optimization discussed in Section 6.3.1, the time grows roughtly linearly with the number of cycles. This can be seen best in the right-hand portion of the graphs with a vertical log axis. (In the right-hand portion, the graph is basically a log-log graph, in which straight lines correspond to linear growth.) Note that in Fig. 6.7(a), because of the points used on the X axis, there is not actually exponential growth at that part of the graph.

For the versions that do not use the reset-killed-variables optimization, the time grows exponentially. This is best seen in the (left portion of) the chart for 3 bits. (In the left-hand portion, the graph is basically a semi-log graph, in which a straight line correpsonds to exponential growth.)

The experiments substantiate two claims. First, in the un-optimized version, the time taken is exponential in the number of repeated loops. Second, the reset-killed-variables optimization eliminates this exponential to produce linear growth in the number of loops instead.

Second, we look at the effect on performance of the number of bits. Figure 6.9 shows the effect of varying the number of bits when there are 2 cycles in a row, and Figure 6.10 shows the effect for 3 bits. Here the story is essentially what we expect: the time taken goes up exponentially with the number of bits. The reset-killed-variables optimization appears to help some, but really that is just a manifestation of the effect that we already saw: the exponential increase is being "applied" fewer times in the optimized version.

Finally, note that turning on ε chains does not seem to have much effect.

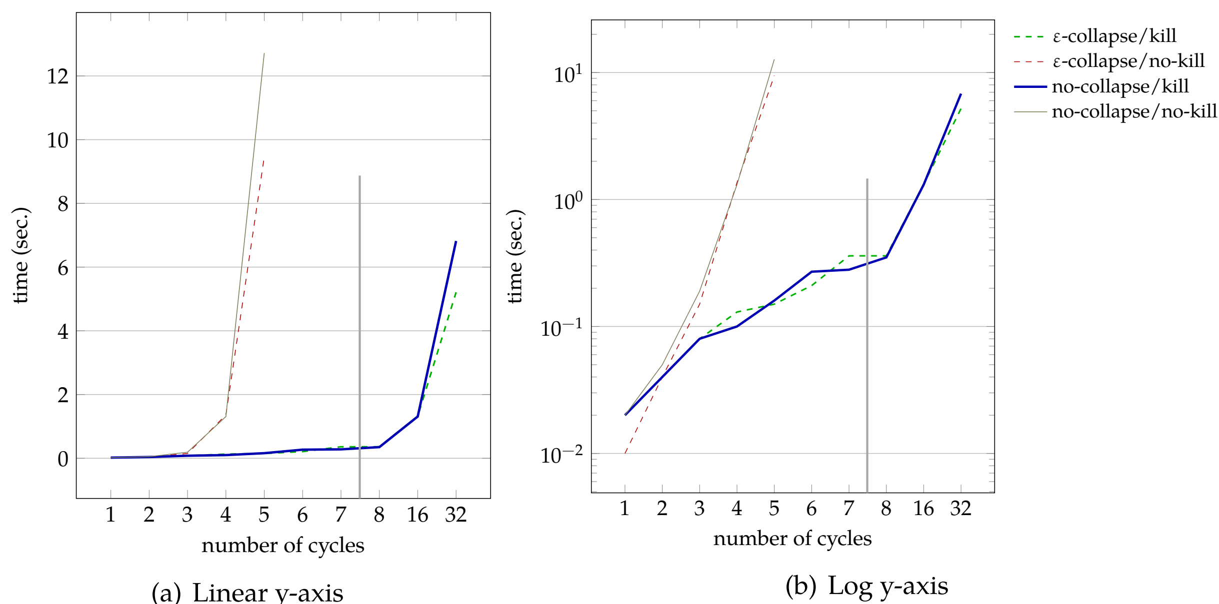
6.5.3 ICO specification performance scaling evaluation

We also tested how PCCA scales on some of the ICO tests described in Section 6.5.1. We tested the following pairs of configurations:

- *ico-spec-all* as both producer and consumer,
- *ico-spec-none* as producer and *ico-spec-A* as consumer,
- *ico-spec-A* as producer and *ico-spec-none* as consumer,
- *ico-spec-A* as both producer and consumer, and
- *ico-spec-A* as producer and *ico-spec-B* as consumer.

These configuration pairs were chosen as follows. *ico-spec-all→ico-spec-all* was chosen because it is the most demanding. Following that, Group A was chosen uniformly at random from the four groups of variables. *ico-spec-none→ico-spec-A* and *ico-spec-A→ico-spec-none* are the two least-demanding tests involving A, and both are included to show the differing effects of variables in the producer and consumer. *ico-spec-A→ico-spec-A* was chosen because there is correlation between the value of the A variables in the producer and consumer automata, and thus it is likely to be less demanding than other tests involving variables in both automata but still be a step up from the previous two tests. Finally, Group B was chosen at random to form *ico-spec-A→ico-spec-B*, to show what happens with different sets of variables in each automata.

For the *ico-spec-all→ico-spec-all* and *ico-spec-A→ico-spec-B* pairs, we increased the number of bits each run by one until they either ran out of time with a five-minute timeout or ran out of memory (enough to start making my machine page, which happens at 7.5–8 GB



(a) Linear y-axis

(b) Log y-axis

Figure 6.7: Fixing the size of each logical variable to 3 bits, a chart of the effect of the number of concatenated loops. The heavy grey vertical line at 8 cycles divides the chart into the left portion, which uses a linear X-axis, and the right portion, which uses a log axis. Both charts display exactly the same data; the right chart uses a log scale on the Y axis.

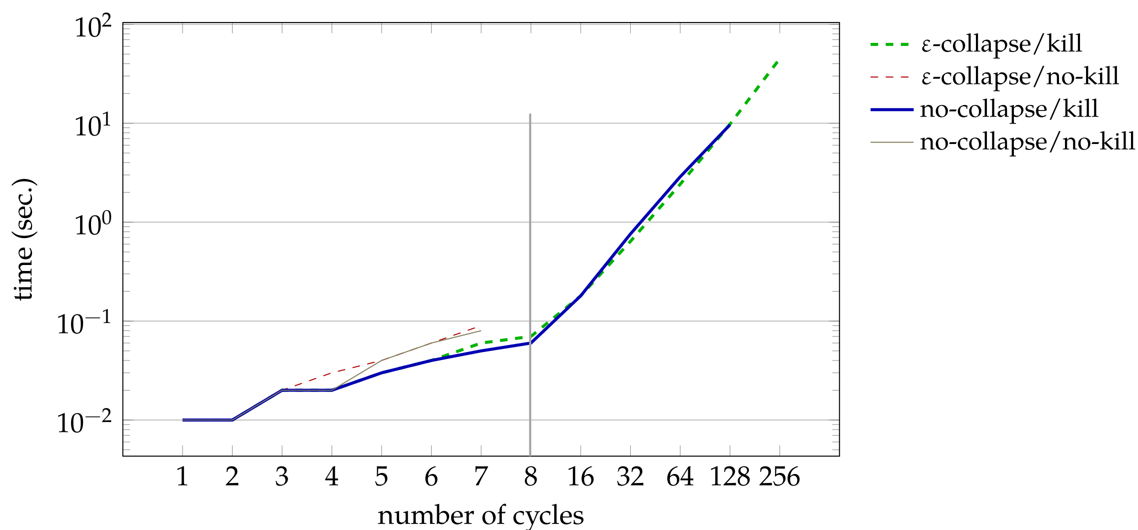
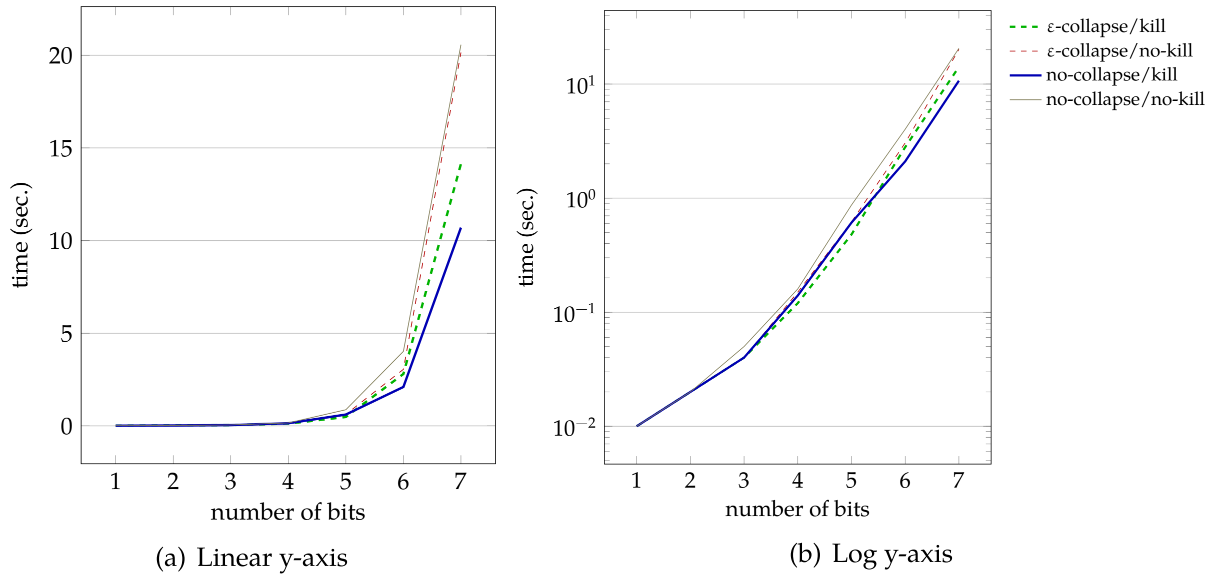


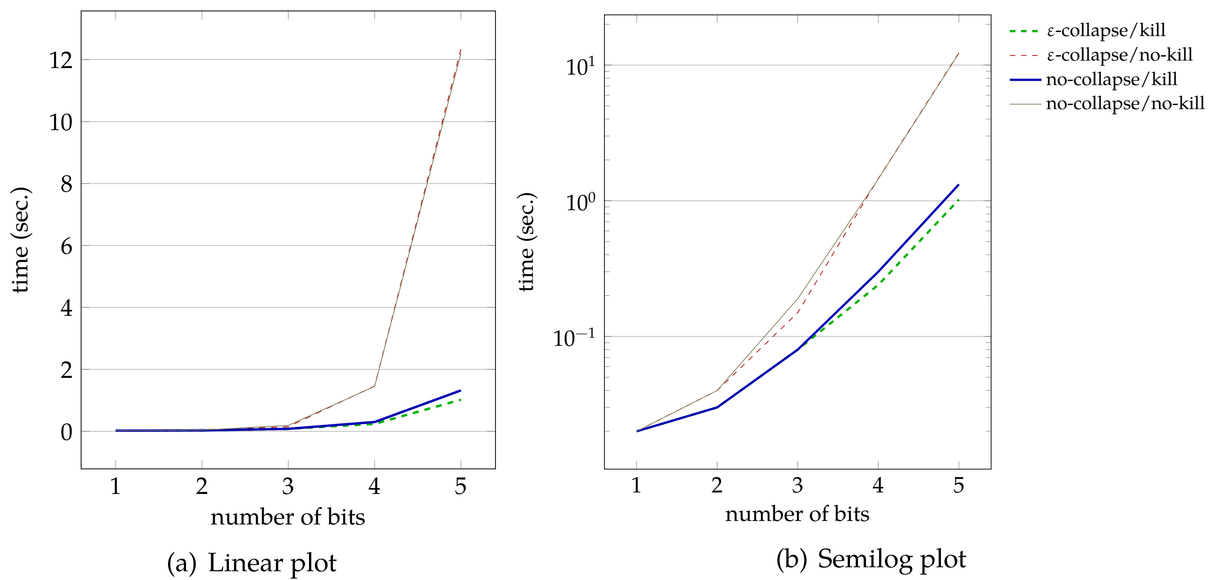
Figure 6.8: Fixing the size of each logical variable to just 1 bit (essentially, tracking whether each loop was taken an even or odd number of times), a chart of the effect of the number of concatenated loops.



(a) Linear y-axis

(b) Log y-axis

Figure 6.9: Fixing the number of concatenated cycles to 2, a chart of the effect of the number of bits. Both charts display exactly the same data.



(a) Linear plot

(b) Semilog plot

Figure 6.10: Fixing the number of concatenated cycles to 3, a chart of the effect of the number of bits. Both charts display exactly the same data.

ico-spec versions	Number of bits per logical variable										
P → C	1	2	3	4	5	6	7	8	9	10	11
-all → *-all*	0.448	25.4	mem								
-A → *-B*	0.133	0.349	2.53	29.8	time						
-A → *-A*	0.104	0.152	0.338	1.10	4.61	21.8	mem				
-A → *-none*	0.096	0.128	0.250	0.700	2.77	12.8	54.6	56.7	240.0	time	
-none → *-A*	0.094	0.104	0.134	0.157	0.284	0.693	2.16	1.95	10.7	58.2	254

Table 6.11: Time in seconds taken for each ICO specification compatibility result. An entry of "—" means that test was not run because it would likely not have been interesting, an entry of "time" means the test took more than five minutes, an entry of "mem" means that the test caused my system to start paging (at about 7.5–8 GB of memory use by PCCA), and a blank entry means the test was not run because an earlier test timed out or ran out of memory.

of use by that process). For the other tests, we increased the number of bits by two each time (because they scale to larger numbers) and, when encountering a timeout, backed off 1 bit to get the previous time.

The results are shown in Table 6.11. As can be seen, the time goes up exponentially. In addition, including all variables means that the test times out extremely quickly; one hypothesis for this is that the variable-killing optimization does not improve things very much because many variables are "in scope" for almost the entire specification.

A user's confidence that a ✓result is returned only because of arithmetic wrap-around in the model (but not the program) should go up as more bits are added to the model. As a result, Table 6.11 shows that, for this example, it is possible to use 4–6 bits in each test (and thus arithmetic would be modulo 16–64) and still complete all combinations of variables within minutes.

7 Related Work

Lim et al. [48] describe the original File-Format Extractor technique, and created an implementation for binary code. Lim's work was a significant inspiration to my beginning steps and to an old version of PCCA. However, the exact techniques she used are different from mine. In addition, the dissertation has discussed several extensions to Lim's work throughout. Further, Lim did not address the format-compatibility problem, but rather used the inferred format to manually check against a specification, and she did not talk about input formats.

A line of work by Fisher et al. [35] aims to infer a grammar for the messages that one component sends another, but takes a very different approach. Instead of analyzing *programs*, it examines example output in an attempt to find its structure. The input to Fischer et al.'s tool is a (fairly large) number of messages. After tokenizing the messages, it looks at a histogram of the number of times each token appears in each of the messages. If a token appears in all or most of the messages about the same number of times, then the algorithm infers that the token in question is part of a struct. If it appears a widely-varying number of times across the messages, it is assumed to be an element of an array. The tool evaluates the inferred format using an information-theoretic computation, and possibly refines the format using some rewriting rules to improve its score.

In contrast to Fisher's work, our approach looks at the program itself. In addition to taking into account infrequently-executed paths and not being as subject to the (in)completeness of the test suite that produces the examples (my techniques have a dependence on the test suite in terms of what invariants Snotra/Daikon are able to find, but as discussed previously, we could use a static approach instead), it is not clear how to use their technique to get a good approximation to the *input* language of a component. Thus it would be hard to base a complete format-compatibility tool such as PCCA off of Fisher's work.

Similarly, Cui, Kannan, and Wang's *Discoverer* [23] attempts to pull off a somewhat similar feat with network communications. It examines network traces that contain messages using *multiple* protocols in an attempt to both split up the messages by protocol and

infer information about the format. The algorithm proceeds by repeated application of a clustering technique, which groups messages that share the same format. (The goal is to arrive at a clustering that is sound in the sense that any two messages in the same cluster do, in fact, use the same protocol format; having two messages in different clusters that use the same protocol is acceptable.) Similar clusters are then merged to obtain a more concise description. The initial clustering is computed by dividing up each message into tokens that are either text or binary, and grouping messages that agree in their format; textual fields are split at delimiters such as whitespace, while each byte in a binary sequence is considered to be its own field. (These may be merged later.) To determine subclusters, the algorithm infers whether each field contains the size of a variable-length field or the offset to the end of it. Messages that disagree in this respect are split into different clusters. Next it finds potential *format descriptor* fields (FD),[1] which can be thought of as the "tag" part of a tagged union, by looking for fields that take on a small number of values across all messages. These are assumed to be FD fields, and the cluster is split into one subcluster per unique value. (Again, these may be merged later.) These specialized clusters may allow the algorithm to infer that more fields are length fields. Finally, the algorithm attempts to merge clusters to avoid overspecialization, by using a subsequence alignment algorithm across the inferred formats. The final output from each cluster consists of information about how many fields are present, which fields are length fields, which fields are FDs, and which fields are binary vs. text.

Saxena et al. [73] introduced a technique they call Loop-Extended Symbolic Execution (LESE). LESE introduces loop trip counters, which makes it possible for the execution engine to reason about the number of executions of a loop. Saxena et al.'s goal is rather different than ours, but part of their process involves finding relationships between input fields and trip counts, much as our goal is for the XFA version of PCCA. While we have not investigated this possibility too much, it is possible that LESE could be used as an alternative to Snotra and Daikon. However, Saxena et al. do not address compatibility between programs, and models inferred in part using LESE would still need to be checked for compatibility.

[1]For instance, often messages from one protocol are encapsulated in messages in an another, outer protocol; the outer protocol will have an FD field specifying what the inner protocol is. A popular example of this is tunneling insecure protocols, such as remote desktop, over SSH. Note that in Cui's technique, it is very likely that HTTP tunnelled over SSH would be considered a different protocol than FTP tunnelled over SSH, and messages using each would be clustered differently.

Devaki and Kanade [28] have the closest piece of related work to this dissertation: they also statically check compatibility of a producer and a consumer by inferring a model of both and then checking compatibility of the models. However, Devaki and Kanade's models are very different from ours, and are incomparably as powerful. One limitation of their technique is that they are only considering inferring information about data headers — they only consider fixed-size messages (or a fixed-size portion of a message) — while our format models incorporate information about the entire format. In addition, because of the fixed-size nature of their models, they do not infer anything like loop-repetition counts. On the other hand, they are able to infer more information about the region of the format they *do* consider. Their models consist of a set of "guarded layouts," where each layout is a mapping from offsets within the header to the type at that position (like a structure layout). Thus they are able to handle constructs like type tags, which we leave for future work. In addition, they do not assume a streaming model, and instead analyze the actual memory reads and writes that the program makes to a certain region.

There is also a fair bit of work on dynamic instrumentation to perform inference. Often, the goal is protocol inference; in its broadest terms, protocol inference includes not just the type of format inference that I discuss, but also inferring a state machine that describes the protocol. Dynamic techniques, such as those described below, seem to provide good results for single tests, but they do not necessarily generalize well beyond the tests that were performed and cannot perform verification. The following paragraphs provide a survey of such dynamic techniques. It is worth noting that a fundamental difference between these approaches and ours is in the use of dynamic analysis; the techniques described below have dynamic analysis at their core, while our use of Snotra and Daikon is one component that could be easily swapped out for a static technique.

Lin and Zhang [49] describe a dynamic analysis that, given a program that parses its input using either a recursive-descent predictive parser or an automatically-generated bottom-up parser *and an input*, infers the parse tree for that input. (Note that the program under analysis need not actually build a parse tree itself.) The analyses are built on dynamic taint-analysis techniques. For top-down parsers, conceptually the analysis proceeds as follows; the authors describe significant changes to make the analysis scale better. First, the analyzer constructs the dynamic control-dependence graph (DCDG) for the execution

in question. If a program point consumes an input value, then its corresponding node in the graph is labeled with that input value as a terminal. Each dynamic occurrence of an instruction corresponds to one node in the DCDG, which means that the graph has no cycles; a portion of this graph is isomorphic to the parse tree. To find it, the analysis removes any node that is not an ancestor of a labeled node. For bottom-up parsers, the analysis watches the parse stack for the push and pop operations that the parser performs. (It is unclear how much manual effort is needed to identify the location of the stack, but the authors' claim is that they analyze stripped executables, and that the parser stack has some mostly unique characteristics.) The authors leave inferring the grammar as a whole for future work. Clearly, the techniques presented in the paper only apply to the program's input, not output.

Caballero et al. discuss *Polyglot* [17], a tool that infers the format of protocols a program communicates with. After running a program under a dynamic taint-analysis monitor, it performs four analyses offline to determine field separators, "direction fields" (e.g., length or pointer fields), "keywords" (e.g., the `ACCEPT` header in an HTTP request), and finally the boundaries between fields. Conceptually these analyses proceed as follows. Field separators are found by looking for a character (or string, such as `\r\n`) that is compared to most or all of the characters in a field. Direction fields are found in one of two ways: either the program uses tainted data to compute an address in the message, or it uses tainted data in the condition of a loop header that walks over a portion of the message. Keywords are found by looking at where the program compared tainted and untainted data and found a match. Finally, field boundaries are found in one of two ways. Boundaries of variable-length fields are determined when the corresponding direction fields are found. Boundaries of fixed-length fields are determined by merging adjacent bytes any time those bytes are used by an instruction at the same time. (This method would generally prevent the technique from finding fixed-length fields wider than a machine word.) Again, the techniques presented by Caballero et al. only work for received messages, not ones the program emits.

Other papers that use dynamic taint analysis followed by a post-processing step include Wondracek et al.'s independent creation of a technique similar to Caballero's Polyglot, the main contribution of which is a method for automatically generalizing from multiple examples; Cui et al.'s *Tupni* [24], the main contribution of which is the handling of repeated

records; and Caballero et al.'s *Dispatcher* [18], the main contribution of which is the ability to infer *output* formats as well as inputs. (Dispatcher's technique for inferring output formats is to taint sources of known information, such as system calls that return information about the environment, with information about the source; if that information shows up in the output, it is included in the format description.)

Komondoor and Ramalingam developed an analysis to recover an object-oriented data model from a program written in weakly-typed languages, such as Cobol [46]. It is capable of recovering information about the record structure of entities that occur in a file, as well as information about subtyping relationships between such entities.

Rajamani and Rehof [69] developed a way to check that an implementation model I extracted from a message-passing program conforms to a specification S. Their goal was to support modular reasoning; they established that if I conforms to S and P is any environment in which P and S cannot starve waiting to send or receive messages, then P and I also cannot starve.

There have also been many papers on session types, starting with [38, 39, 63]. In some sense, this body of work has the same goal that we have—helping to ensure that different components communicate properly—but their approach is far different. Session types, at a high level, convey much the same information as our inferred languages. (For instance, in the syntax of Honda [39], "`↑int; (↑char & ↑double)`" is the type of a component that emits an `int` followed by either a `char` or `double`.) Some recent work, e.g., Hu et al.'s [42], is integrating session types into common programming languages.

In most of this literature, session types need to be incorporated into the language being used to write the components, which means they cannot be applied to legacy software without rewriting it. In contrast to these papers, our work analyzes existing C/C++ code for compatibility by inferring the format. In return for the re-engineering effort, session types support richer interactions than we currently do; most notably, it can specify bidirectional communications.

Recently there have been advances in *inferring* session types, which is much closer to our goal. Mezzina [56] and Collingbourne and Kelly [22] each developed such an algorithm. Collingbourne's is particularly related to PCCA, as they implemented their technique in a source-to-source translator for C++. However, neither paper really gives enough information on how it performs in practice to compare to PCCA.

In a similar vein — and actually of equivalent power — are channel contracts from the Singularity OS [34]. Channel contracts specify a protocol between two endpoints as a state machine, where each state specifies messages that each endpoint can send or receive. Fähndrich et al. describe an analysis that verifies certain memory-safety properties in programs that use channel contracts. More recent work has analyzed channel contracts with respect to deadlocks [79] and developed formal type theories for channels [14].

Behavioral subtyping

There is a lengthy literature on the concept of *behavioral subtyping*. This line of work attempts to describe when it is possible to substitute one component for another, taking into account whether the behaviors (rather than just the interface, as in traditional subtyping) are compatible. A statement of the requirement of behavior subtyping is expressed in Liskov and Wing [51]: "Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T."

Techniques developed in this field apply to problems of the following form: given a pair of communicating components P and C, is it possible to substitute a different component, P', for P (resp. substitute C' for C)? The answer is "yes" if P' is a subtype of P (resp. C' is a subtype of C). This question is of interest if one party is trying to determine whether to upgrade a component they use, or whether there is a non-backwards-compatible change.

This problem is something like a *component-upgrade* question — is it possible to upgrade P or C to a new version? This is related to my format-compatibility goal, but is still a different question. The question my research tries to answer is "do these two components cooperate?", while the question that behavioral subtyping is aimed at is "I have two components that cooperate; can I replace one of the components with another?"

As an example of work in this area, McCamant and Ernst [53] discuss using Daikon [32] to infer invariants that apply to each of the old and new versions of the component being upgraded. The inference of invariants used in the old version would be performed by the party deciding whether to upgrade, and the invariants produced indicate the properties of the old component that the system relies on. The inference of invariants in the new version could be performed by the party that created it in the first place, using their own test suite. McCamant's technique infers pre- and post-conditions for each version of the components. Denote the old version as A, the new version as B, and the pre- and post-

conditions as A_{pre}, A_{post}, B_{pre}, and B_{post}. The upgrade is considered okay if and only if $(A_{pre} \rightarrow B_{pre}) \wedge ((A_{pre} \wedge B_{post}) \rightarrow A_{post})$. The authors then assume that $B_{pre} \rightarrow B_{post}$ because it held in every test case. The second conjunct establishes that, under these assumptions, the post conditions the system relied on with the old version will continue to hold.) This formula is tested by submitting it to the Simplify theorem solver [27, 61]. Later, the same authors extended and modified the technique [54] to be more robust in the face of real-world problems.

8 Conclusions

This dissertation has presented techniques for analyzing programs to determine input and output formats, and then for determining the compatibility of such formats across programs. Along the way, we developed a number of algorithms for nested-word automata, weighted-finite automata, and extended finite automata.

Chapter 2 presented some new observations and algorithms for nested-word automata:

- Several new observations regarding ε transitions in NWAs: (1) the very natural extension of NWAs that allows ε internal transitions, (2) a description of why allowing ε on call and return edges is not possible, and (3) an alternative interpretation of ε transitions that can change (positively or negatively) the number of transitions needed to recognize a given language,
- A reformulation of Alur and Madhusudan's determinizattion algorithm that makes clearer how the algorithm works (along with a description of how to incorporate the alternative ε semantics),
- The discovery of the fact that Alur and Madhusudan's Kleene star algorithm was flawed, and a patch to correct it, and
- A previously-unpublished algorithm for reversing NWAs in our setting.

In addition, the dissertation describes the OpenNWA library, which I had a significant hand in creating.

The determinization algorithm and observations about the impossibility of ε calls and returns played a significant role in the development of PCCA. Our formulation of determinization was needed so that we could perform language containment (as remarked in Section 2.1.4, Alur and Madhusudan's two papers were contradictory in what happens when handling call transitions, and we had to understand which version was correct). The

fact that ε transitions cannot label calls and returns led directly to our use of $\langle$ and $\rangle$ in the NWA version of PCCA, which in turn led to our need of developing the Enrich operation.

Meanwhile, the alternative ε interpretation arose because of work by Prabhu, Turetsky, and Reps regarding a particular NWA construction that would benefit from the ability to use the broader interpretation. There is no support for this interpretation in OpenNWA (it is primarily helpful to reduce clutter in diagrams of the NWA in question), but the fact that such an interpretation is consistent is very interesting and helps to fill in knowledge about NWAs. Similarly, the new reversal algorithm and Kleene star help fill in or correct knowledge about NWAs, though are not explicitly used in any of our applications.[1]

Future directions: The alternative ε interpretation is the least-explored of the topics addressed in Chapter 2. While it certainly seems like existing NWA algorithms such as intersection, concatenation, and even Kleene star continue to work with the alternative ε interpretation, we have not explicitly proven these facts. In would also be interesting to investigate whether actually supporting the alternative interpretation is helpful or not in practice.

Chapter 3 presented several algorithms and operations for XFAs and WFAs:

- Discussions and comparisons of three methods for performing ε closure of a WFA (Mohri's algorithm, the standard iterative WPDS algorithm, and Tarjan's algorithm via FWPDS),
- Partially generalizing the XFA determinization algorithm to WFAs, which provides a choice of two WFA determinization algorithms with different requirements on the weights: my algorithm, which for a transition (p, σ, q) with weight w requires computing $w \odot \{p \to q\}$, and Mohri's algorithm [58, §3.2], which requires taking the multiplicative inverse of each weight,
- A description of how to compute language containment of two WFAs (subject to the $\odot$ restriction) that, in the context of XFAs, differs from the approach of fully-determinizing the XFAs involved using Smith et al.'s algorithm [76], complementing, intersecting, and checking emptiness,

[1] OpenNWA supports both operations of course, but they are present for completeness rather than because we added them on-demand.

- A description of how to use the idea of De Wulf et al.'s antichains algorithms to improve containment checking.

In addition, we briefly described our implementation of XFAs in WALi that use BDDs.

All of our algorithms were guided by what we perceived as the real needs of the XFA version of PCCA at the time. The BDD-based operations turned out to be more expensive than anticipated, unfortunately, and it is unclear how well the algorithms work in practice.

Future directions: One very interesting topic that we have not looked into is how much our WFA state-determinization algorithm *actually* generalizes Smith et al.'s XFA description. We found the formalisms useful for two purposes: (1) it helps guide implementations of XFAs that are built upon our existing WFA classes, and (2) it allows for some simple proofs of correctness. However, the state-determinization process imposes a condition that there be a tensor-product operation that, given a weight w, allows computing $w \odot \{\mathsf{q} \to \mathsf{p}\}$. It is not clear what such an operation would be for weight domains that are not relations. Can that operation be made useful outside of relational domains?

Chapter 4 presented Snotra, a binary front end for Daikon. Combined with the Daikon back end, Snotra provides a means for obtaining invariants (or at least candidate invariants) for programs. It is currently targeted toward the instrumentation that is needed for the XFA version of PCCA, which instruments I/O function calls and loops. In that context, the invariants it finds correspond to invariants that the value of a field in the message equals the repetition count of another group of fields.

Future directions: Snotra is a new tool, and there is a lot of potential for it to be used in other contexts. As mentioned in Chapter 4, it is built to be highly extendable, allowing reasonably easy specifications about what program points and what "variables" are of interest. There are multiple tools in our group that have the potential to be able to make use of candidate invariants to decrease analysis time, including McVeto [82, §3.2]. Junghee Lim reported that Snotra appeared to be helpful for early attempts at McTreeIC3.

Chapter 5 presented PCCA, a tool that analyzes programs for compatibility. The basic version performs a control-flow abstraction of the program, using finite automata to model each program. Even this simple model was able to detect an inconsistency in how GZIP reads and writes part of the files' headers as well as find unintentional errors in a manually-crafted

specification for the ICO format as it was being created. PCCA marked as compatible a modified version of GZIP, PNG2ICO against the manually-crafted specification, and other programs.

An NWA version of PCCA adds context sensitivity to the producer's automaton, at a modest increase in running time. Context sensitivity was not important for our real-world test cases, but showed its utility in a synthetic example. To make compatibility checking with NWAs actually return a useful answer, in most cases it is necessary to Enrich the consumer's automaton first so that the two components are not required to have the same internal call/return structure.

Future directions: On the theoretical side, a question that I think would be very interesting would be to investigate the properties of Enrich. We just sort of use it and remark that it is somewhat analogous to a regular approximation of the original language, but it would be interesting to characterize exactly what Enrich does. There are other techniques that have been suggested for producing a regular approximation to a context-free language, for example by Mohri and Nederhof [43] and Pereira and Wright [65]; can Enrich be compared to and contrasted with those? Does such a comparison make sense, and if so, which is better? (An alternative approach would be to use a pushdown automaton for the producer in place of the NWA, and then use a slightly more traditional CFL-to-RL approximation in place of Enrich.)

On the practical side, perhaps the biggest complication to using PCCA is dealing with how programs actually perform I/O in practice. A lot of programs do not read or write data in a way that is easy to adapt into the PCCA model, at least under the current implementation. They do block reads of larger amounts data and read out parts of interest (probably solvable with a modest amount of engineering effort), perform seeks in a file to get to the portion of interest (would require a completely new technique), or otherwise violate PCCA's assumptions. These limitations have made finding good test cases somewhat difficult. Loosening the assumptions to cover a wider range of programs would be highly useful.

Another idea for a future extension is to deal properly with error-handling code. For example, if the consumer tries to read some data but fails because the input is malformed and enters some error code and then exits, that path will still correspond to a string in the program's language — but it really should not, because that string was not accepted by the

consumer in a meaningful sense. Taking error-handling code into account was something we planned to do "on demand" — if we found an example for which PCCA was producing the wrong result because of error-handling code, we could handle it at that point. My rough idea was to use an existing technique to detect code that is likely to be handling errors, and then simply disconnect it from being able to reach the automaton's accepting state. The disconnection would prevent strings that need to traverse the error handling code from being accepted. We never ran into the apparent need to do anything special with error-handling code, but it is something that would be a possibility if this research is carried forward.

Chapter 6 described a version of PCCA that is able to incorporate information involving certain kinds of data flow within the program. We focus on inferring and checking compatibility of invariants that the value of one field in the message equals the number of repetitions of a different field or groups of fields. We instrument the programs of interest, find invariants using Snotra and Dakion, and then incorporate those invariants into the model of the programs. XFAs are used to store information about the values read or written and loop trip counts, which form the basis of the invariants that the model enforces.

Future directions: The obvious future work here is to find a way to adapt the ideas developed by this section in a way that works fast enough to run on real programs.

If that can be solved, then there are a number of very interesting avenues that can be followed. In particular, we focus on counted fields in the format. There are a number of other format features that one could capture. For instance, another format feature that is used a lot is something like a type tag: there is one field that can take on any of a small number of choices, and the value of that field controls which of a set of options is read later. For example, consider the following code:

```
1  type = read_char()
2  if type == 'A':
3      x = read_double()
4  else:
5      y = read_string()
```

PCCA would infer the following regular expression:

$$\texttt{char}\ (\texttt{double} \mid \texttt{string})$$

However, a technique much like our way of dealing with counted fields could determine that, for instance, `double` is only read if the first `char` is "A". This could be denoted by something like

$$\tau : \texttt{char}\ (\textsc{a} \rightarrow \texttt{double} \mid_\tau \text{(other)} \rightarrow \texttt{string})$$

where the $|_\tau$ specifies that the choice of alternative depends on the value that was read at τ. This relationship could be found and enforced almost identically to the field counts: Snotra/Daikon would determine that at Line 3, `type` has the value "A" (or more specifically, that the value read on Line 1 was "A"), and then that constraint would be enforced in the XFA much like the exit constraint from a cycle.

The idea in the previous paragraph can potentially be extended to many data behaviors. If there is some invariant that Daikon is able to find that seems like it could help with format compatibility, all that needs to be done is to make sure that the XFA's transformers mimic the program's behavior with regards to relevant portions of the program. If $y > 5$ is an invariant that Snotra/Daikon found for some program point and there is reason to believe that the invariant is important, then the XFA could potentially track changes to y and then enforce that $y > 5$ at that program point. These enforcement constraints can become arbitrarily difficult as memory is manipulated, but there may be a large set of facts that are relatively easy to model.

Finally, a wild out-there idea would be to incorporate a CEGAR loop [20] into PCCA. In particular: if a counterexample s to $P \subseteq C$ is found, use symbolic execution or another technique to determine whether s is actually feasible in P. If not, then refine P to exclude s. How to actually perform the refinement is an open question; perhaps it would use something related to the previous paragraph. In addition, the technique as just explained only accounts for one of the two kinds of errors; it is not clear what to use as a basis for refinement if PCCA finds that $P \subseteq C$ holds.

Bibliography

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, & Tools*. Addison-Wesley, second edition, 2006.

[2] Jürgen Albert and Jarkko Kari. *Digital Image Compression*, chapter 11. In Doste et al. [29], 2009. Online edition.

[3] Cyril Allauzen, Michael Riley, Johan Schalkwyk, Wojciech Skut, and Mehryar Mohri. OpenFst: A general and efficient weighted finite-state transducer library. In *Conf. on Impl. and Applications of Automata*, 2007.

[4] Rajeev Alur. personal communication, August 2011.

[5] Rajeev Alur. Nested words, 2011. `http://www.cis.upenn.edu/~alur/nw.html`.

[6] Rajeev Alur and P. Madhusudan. Visibly pushdown languages. In *Symp. on Theory of Comp.*, 2004.

[7] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. In *Developments in Lang. Theory*, 2006.

[8] Rajeev Alur and P. Madhusudan. Adding nesting structure to words. *J. ACM*, 56(3), 2009.

[9] Christel Baier, Marcus Größer, and Frank Ciesinki. *Model Checking Linear-Time Properties of Probabilistic Systems*, chapter 13. In Doste et al. [29], 2009. Online edition.

[10] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Comp. Construct.*, pages 5–23, 2004.

[11] Arati Baliga, Vinod Ganapathy, and Liviu Iftode. Automatic inference and enforcement of kernel data structure invariants. In *Annual Computer Sec. Applications Conf.*, December 2008.

[12] Beate Bollig and Igno Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Trans. on Computers*, 45(9), 1996.

[13] Benedkit Bollig, Joost-Pieter Katoen, Carsten Kern, Martin Leucker, Daniel Neider, and David R. Piegdon. libalf: The automata learning framework. In *Computer Aided Verif.*, 2010.

[14] Viviana Bono, Chiara Messa, and Luca Padovani. Typing copyless message passing. In *European Symp. on Programming*, 2011.

[15] Víctor Braberman, Federico Fernández, Diego Garbervetsky, and Sergio Yovine. Parametric prediction of heap memory requirements. In *Int. Symp. on Mem. Mgmt.*, pages 141–150, Tucson, AZ, USA, June 7–8, 2008.

[16] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. on Computers*, C-35(8), 1986.

[17] Juan Caballero, Heng Yin, Zhenkai Liang, and Dawn Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Conf. on Comp. and Commun. Sec.*, 2007.

[18] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Conf. on Comp. and Commun. Sec.*, 2009.

[19] Raymond Chen. The evolution of the ICO file format (parts 1–4), Oct 2010. `http://blogs.msdn.com/b/oldnewthing/archive/2010/10/18/10077133.aspx`, `http://blogs.msdn.com/b/oldnewthing/archive/2010/10/19/10077610.aspx`, `http://blogs.msdn.com/b/oldnewthing/archive/2010/10/21/10078690.aspx`, `http://blogs.msdn.com/b/oldnewthing/archive/2010/10/22/10079192.aspx`.

[20] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), September 2003.

[21] codesurfer. CodeSurfer system, 2013. www.grammatech.com/products/codesurfer.

[22] Peter Collingbourne and Paul H J Kelly. Inference of session types from control flow. *Electr. Notes Theor. Comp. Sci.*, 238(6), 2010.

[23] Weidong Cui, Jayanthkumar Kannan, and Helen J. Wang. Discoverer: Automatic protocol reverse engineering from network traces. In *USENIX Sec. Symp.*, 2007.

[24] Weidong Cui, Marcus Peinado, Karl Chen, Helen J. Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Conf. on Comp. and Commun. Sec.*, 2008.

[25] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Computer Aided Verif.* 2006.

[26] M. De Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Slides for Antichains: A new algorithm for checking universality of finite automata [25], 2006. `http://www.lsv.ens-cachan.fr/~doyen/antichains/slides/antichains.pdf`.

[27] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: A theorem prover for program checking. *J. ACM*, 52(3), 2005.

[28] Pranavadatta Devaki and Aditya Kanade. Static analysis for checking data format compatibility of programs. In *Found. of Softw. Tech. and Theoretical Comp. Sci.*, 2012.

[29] Manfred Doste, Werner Kuich, and Heiko Vogler, editors. *Handbook of Weighted Automata*. Springer Berlin Heidelberg, 2009. Online edition.

[30] Evan Driscoll, Amanda Burton, and Thomas Reps. Checking conformance of a producer and a consumer. In *Found. of Softw. Eng.*, 2011.

[31] Evan Driscoll, Aditya Thakur, Amanda Burton, , and Thomas Reps. WALi: Nested-word automata. TR-1675R, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, September 2011.

[32] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. on Softw. Eng.*, 27(2), February 2001.

[33] Zoltán Ésik and Werner Kuich. *Finite Automata*, chapter 3. In Doste et al. [29], 2009. Online edition.

[34] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi. Language support for fast and reliable message-based communication in Singularity OS. In *European Conf. on Comp. Sys.* ACM, 2006.

[35] Kathleen Fisher, David Walker, Kenny Q. Zhu, and Peter White. From dirt to shovels: Fully automatic tool generation from ad hoc data. In *Princ. of Prog. Lang.*, 2008.

[36] Oliver Friedmann, Felix Klaedtke, and Martin Lange. Ramsey goes visibly pushdown. In *Intl. Colloq. on Automata, Lang., and Prog.*, 2013.

[37] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Softw.*, 12(6), November 1995.

[38] Simon Gay, Vasco Vasconcelos, and Antonio Ravara. Session types for inter-process communication. TR-2003-133, Dept. of Computing Sci., Univ. of Glasgow, March 2003.

[39] Kohei Honda. Types for dyadic interaction. In *Conf. on Concurrency Theory*. 1993.

[40] John E. Hopcroft, Rajeeve Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, second edition, 2001.

[41] John Hornick. Icons, Sept. 1995. `http://msdn.microsoft.com/en-us/library/ms997538.aspx`.

[42] Raymond Hu, Dimitrios Kouzapas, Olivier Pernet, Nobuko Yoshida, and Kohei Honda. Type-safe eventful sessions in java. In *European Conf. on Obj.-Oriented Prog.* Springer, 2010.

[43] Jean-Claude Junqua and Gertjan van Noord, editors. *Robustness in Language and Speech Technology*. Kluwer Academic Publishers, 2000.

[44] N. Kidd, A. Lal, and T. Reps. WALi: The Weighted Automaton Library, 2007. www.cs.wisc.edu/wpis/wpds/download.php.

[45] Kevin Knight and Jonathan May. *Applications of Weighted Automata in Natural Language Processing*, chapter 14. In Doste et al. [29], 2009. Online edition.

[46] Raghavan Komondoor and G. Ramalingam. Recovering data models via guarded dependences. In *Working Conf. on Rev. Eng.*, 2007.

[47] Akash Lal and Thomas Reps. Improving pushdown system model checking. In *Computer Aided Verif.*, 2006.

[48] Junghee Lim, Thomas Reps, and Ben Liblit. Extracting output formats from executables. In *Working Conf. on Rev. Eng.*, 2006.

[49] Zhiqiang Lin and Xiangyu Zhang. Deriving input syntactic structure from execution. In *Found. of Softw. Eng.*, 2008.

[50] Jørn Lind-Nielsen. BuDDy: A BDD package.

[51] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. on Prog. Lang. and Syst.*, 16(6), 1994.

[52] P. Madhusudan. Visibly pushdown automata – automata on nested words, 2009. `http://www.cs.uiuc.edu/~madhu/vpa/`.

[53] Stephen McCamant and Michael D. Ernst. Predicting problems caused by component upgrades. In *Found. of Softw. Eng.*, 2003.

[54] Stephen McCamant and Michael D. Ernst. Early identification of incompatibilities in multicomponent upgrades. In *European Conf. on Obj.-Oriented Prog.*, 2004.

[55] K. McMillan. *Symbolic Model Checking*. Kluwer Acad., 1993.

[56] Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In *Coordination Models and Lang.* 2008.

[57] Mehryar Mohri. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2), June 1997.

[58] Mehryar Mohri. Generic ε-removal algorithm for weighted automata. In *Conf. on Impl. and Applications of Automata*. Springer, 2000.

[59] Mehryar Mohri, Fernando C. N. Pereira, and Michael D. Riley. AT&T FSM library – finite-state machine library, 2002. `http://www2.research.att.com/~fsmtools/fsm/`.

[60] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2010. `http://www.brics.dk/automaton/`.

[61] Charles Gregory Nelson. *Techniques for program verification*. PhD thesis, Standford University, Stanford, CA, USA, 1980.

[62] H. Nguyen. Visibly pushdown automata library, 2006. `http://www.emn.fr/z-info/hnguyen/vpa/`.

[63] Oscar Nierstrasz and Michael Papathomas. Viewing object as patterns of communicating agents. In *Obj.-Oriented Prog., Sys., and Applications*, 1990.

[64] Fernando Pereira, Michael Riley, and Richard Sproat. Weighted rational transductions and their application to human language processing. In *Workshop on Human Lang. Tech.*, 1994.

[65] Fernando C. N. Pereira and Rebecca N. Wright. Finite-state approximation of phrase structure grammars. In *Assoc. for Comp. Linguistics*, 1991.

[66] Jeff H. Perkins, Sunghun Kim, Sam Larsen, Saman Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst, and Martin Rinard. Automatically patching errors in deployed software. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, October 12–14, 2009.

[67] MIT program analysis group. The Daikon invariant detector, 2013. `http://groups.csail.mit.edu/pag/daikon/`.

[68] Paradyn Project. Dyninst: Putting the performance in high performance computing. `http://www.dyninst.org`.

[69] Sriram K. Rajamani and Jakob Rehof. Conformance checking for models of asynchronous message passing software. In *Computer Aided Verif.*, 2002.

[70] G. Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Princ. of Prog. Lang.*, 1999.

[71] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Princ. of Prog. Lang.*, 1995.

[72] Thomas Reps, Stefan Schwoon, Somesh Jha, and David Melski. Weighted pushdown systems and their application to interprocedural dataflow analysis. *Sci. of Comp. Prog.*, 58(1–2), October 2005.

[73] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. Loop-extended symbolic execution on binary programs. In *Int. Symp. on Softw. Testing and Analysis*, 2009.

[74] David Schuler, Valentin Dallmeier, and Andreas Zeller. Efficient mutation testing by checking invariant violations. In *Int. Symp. on Softw. Testing and Analysis*, July 2009.

[75] Michael Sipser. *Introduction to the Theory of Computation*. Thompson Course Technology, second edition, 2006.

[76] Randy Smith, Christan Estan, and Somesh Jha. XFA: Faster signature matching with extended automata. In *Symp. on Sec. and Privacy*, 2008.

[77] Randy Smith, Cristian Estan, Somesh Jha, and Shijin Kong. Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In *SIGCOMM*, 2008.

[78] Varun Srivastava, Michael D. Bond, Kathryn S. McKinley, and Vitaly Shmatikov. A security policy oracle: detecting security holes using multiple api implementations. In *Prog. Lang. Design and Impl.*, 2011.

[79] Z. Stengel and T. Bultan. Analyzing Singluarity channel contracts. In *Int. Symp. on Softw. Testing and Analysis*. ACM, 2009.

[80] Robert Endre Tarjan. Fast algorithms for solving path problems. *J. ACM*, 28(3), 1981.

[81] Robert Endre Tarjan. A unified approach to path problems. *J. ACM*, 28(3), 1981.

[82] A. Thakur, J. Lim, A. Lal, A. Burton, E. Driscoll, M. Elder, T. Andersen, and T. Reps. Directed proof generation for machine code. TR 1669, UW-Madison, April 2010. Abridged version published in CAV 2010.

[83] MIT theory of computation group. IOA language and toolset, 2013. `http://groups.csail.mit.edu/tds/ioa/`.